Software Engineering:

Principles and Practices

Ravi Sethi

The University of Arizona

January 12, 2021

Version 7.10

This working draft is intended for students enrolled in Computer Science 536, *Software Engineering*, Spring 2021, at the University of Arizona.

Preface

> "Because so much of what is learned will change over a student's professional career and only a small fraction of what could be learned will be taught and learned at university, it is of paramount importance that students develop the habit of continually expanding their knowledge."
>
> — *The joint IEEE-ACM software engineering curriculum guidelines stress the importance of continued learning.*[1]

About this Draft

This book is under iterative construction. The main content sections have been classroom tested over the past six years. The Conclusion sections are early drafts and the book could use a concluding chapter. This draft also needs more exercises for more even treatment across chapters.

Focus on Basic Principles and Best Practices

The selection of content for this book has been guided by the question: What do developers really need to know about software engineering to be productive today and relevant tomorrow? The discipline of software engineering continues to change, driven by new technologies, applications, and development methods. There is every indication that such changes will continue as the developers' careers unfold. For example, continuous deployment requires every aspect of software development and deployment to be reimagined.

The basic principles remain constant, amidst changes in the discipline: software is complex; requirements change; defects are inevitable; software is a team sport (software systems are generally built by teams); and so on. If anything, software is getting more complex. Testing may get automated, but the notion of test coverage continues to be a measure of goodness for a test suite.

The focus of this book is on basic principles and best practices based on those principles. A firm grounding in basic principles will help developers adapt as software engineering evolves. The book includes real-world examples and case studies, where possible. For perspective, some classic examples are included to illustrate the durability of the principles.

Use of this Book

This book is intended for a junior-senior level introductory course in software engineering. It is about organizing a development team; working with users/customers; architecting a system; and validating and verifying a system.

The primary use of the book is expected to be in a course with a team project. The book assumes that students have enough programming maturity to contribute to o such a project. Note that the ACM-IEEE computer science curriculum guidelines strongly recommend that a software-engineering course include a significant team project.[2]

For students with some development experience, the book has also been used in courses without projects.

Content Organization and Coverage

The chapters in this book can be grouped as follows:

Part I: Getting Started

1. Introduction
2, Development Processes

Part II: What to Build?

3. Requirements
4. User Requirements
5. Use Cases

Part III: Useful Techniques

6. Estimation and Prioritization
7. Goal Analysis

Part IV: Decomposing the Problem

8. Software Architecture
9. Architectural Patterns

Part V: Delivering Quality

10. Static Validation and Verification
11. Testing
12. Software Quality

Getting Started. Chapter 1 introduces key topics that are explored in the rest of the book: requirements, iterative processes, software architecture, and

testing. The chapter also has a section on social responsibility and professional conduct.

The selection of a development process is one of the earliest decisions during a software project. Processes affect every aspect of a project. Processes are therefore discussed early, in Chapter 2.

What to Build? Chapter 3 provides an overview of requirements development. With plan-driven processes, requirements development is a major early phase; it is significant enough that it is called requirements engineering, and is considered a sub-field of software engineering. With agile development, requirement development blends into overall software development. Dependencies on the development process are confined largely to Chapter 3.

Chapter 4 introduces a model of user needs. The chapter discusses levels of user needs; how to access them; and how to record them. What users say relates to just the first level of needs. What users do relates to the next level. Chapter 5 deals with use cases.

Useful Techniques. The techniques in Chapters 6 and **??** have applications beyond requirements analysis. Estimation is a key element of project management; it is needed for predicting costs and schedules and for work assignment. (Kano analysis is specific to requirements analysis.)

Goal analysis is useful enough that it appears to have been reinvented in three different contexts: requirements analysis; security, in the form of attack trees; and goal-directed actions and measurement.

Decomposing the Problem. Software architecture, Chapter 7, is key to dealing with the complexity of software. Architectural patterns, Chapter 8, provide design guidance, distilled from previous solutions to similar problems. Chapter 8 includes a section on client-server architectures. Such architectures enable new software to be added to a production system: a load balancer can direct some of the incoming traffic to the new software on a trial basis, until the new software is ready to go into production. The ability to add new software to a production system is needed for continuous deployment.

Delivering Quality. The combination of architecture plus code reviews, static analysis, and testing is much more effective for defect detection than any of these techniques by themselves. Chapter 9 discusses reviews and static analysis. The focus of the chapter is on static or compile-time techniques. Chapter 10 is on testing, which is done by running code on specific inputs. The chapter includes MC/DC and Combinatorial testing.

Chapter **??** explores views of quality, such as product quality and post-delivery operational quality. Operational quality refers to quality after a system is installed at a customer site.

Organizing a Course with a Team Project

A team project can help students relate to the concepts in a course. Software engineering addresses problems of complexity and scale. Working with a real

customer on a project suitable for a team of 4, or perhaps 3, allows students to experience the benefits of engineering methods.

In a course with a concepts track and a project track, one challenge is that students need help making connections between the concepts presented in the classroom and the project they are working on. In terms of a learning hierarchy, the students are at a familiarity or guided-usage level, so they are not yet ready to assess the applicability of the concepts in another context. Clearly, it helps if the connections between the tracks are made e explicit, say, by using examples from the student projects. In addition, it helps to align the concepts track with the project as much as possible.

Example. This example is for a course that uses an iterative or agile software development process. Each course has its own context and objectives, so the following suggestions may need to be adapted to suit local needs. Hopefully, they can serve as a starting point.

1. Settle on an iterative/agile development process for the project. Suggestion: Use Scrum.

2. Create a timeline for the course and mark the constraints; e.g., exam dates, holidays. external events.

3. Fit in the iterations, allowing for start-up and wrap-up (including final project demos). Suggestion: Use roughly 3-week iterations, with an initial iteration for requirements. The initial iteration is also a good time for the teams to have a rough plan for what they want to accomplish in each iteration.

4. Align classroom content with the iterations, to the extent possible.

In the first few weeks, the focus in the classroom is on core concepts that the students need for their projects, concepts related to processes, requirements, and architecture. Once the student projects are underway, there is more flexibility for covering topics that may not fit within the constraints of a student project. □

Acknowledgments

Ravi Sethi
Tucson, Arizona

Contents

January 12, 2021

Chapter 1

Introduction

> "Software, during the early days of [the Apollo space] project, was treated like a stepchild and not taken as seriously as [hardware] ... I fought to bring the software legitimacy so that it (and those building it) would be given its due respect"
>
> — *Margaret Hamilton began using the term software engineering to put software on a par with hardware during the early days of the Apollo project, around 1963-64. The term was coined independently in multiple contexts in the early 1960s.*[1]

---

a

1.1 What is Software Engineering?

Products and Processes

Simply stated, software engineering is the application of engineering methods to the development of software products. Examples of products include software systems, services delivered from the cloud, test suites, and documentation. In general, *product* is a convenient term for any deliverable from a software engineering project. The term product does not apply to programs intended for personal use. Software engineering is aimed at products intended for use by others.

Let us turn now from what is developed (the product) to how it is developed. A *process* is a set of rules and guidelines for organizing the activities of a development team. Software products are typically built teams. Processes are used to define team member roles and coordinate their activities. The choice

of a suitable development process is one of the most significant decisions for a software project.

Definition of Software Engineering

Software engineering is a rich subject, too rich to be characterized by a simple definition. Definitions therefore tend to focus on some specific aspect of the subject. Over the years,, most definitions have focused on the application of processes to the development of products. Consider, for example, the IEEE definition:

> *Software engineering* is the "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."[2]

A process is "a systematic, disciplined, quantifiable approach." Meanwhile, product is a convenient term for the outputs of software-related activities such as "development, operation, and maintenance."

Since the IEEE definition of software engineering is widely available through the Software Engineering Body of Knowledge (SWEBOK), we shall use it as the working definition in this book.

Let us also touch on what software engineering is not. It is not aimed at single-person programming of single-person programs for personal or throwaway use. Engineering practices may well be adapted to write programs for personal use, but the primary purpose of the practices is to create robust products that will be used by others.

Customer Needs and External Constraints

Engineering projects have a purpose: they produce something useful for someone at an acceptable cost by an agreed-upon time. We refer to the someone as the *user* or as the *customer*. Technically, the customer is the one who commissions or pays for the product and a user is someone who uses the product. If there is no reason to distinguish between them, we use the term customer broadly to include users.

Example 1.1. This example illustrates customer responsiveness as a driving force that influences all aspects of software development and deployment in a successful technology company.

In order to respond rapidly to customer suggestions and feedback, the company updates its software frequently: it makes hundreds of changes a day to its main servers. Each change is small, but the small changes add up. The practically "continuous" updates enable the company to respond quickly to customer feedback.

Each change is made by a small team that is responsible for "everything" about the change: design, coding, testing, and then deploy deployment directly

on customer-facing servers. The change cycle from concept to deployment takes just a few days. There are multiple teams working in parallel. Together, they make hundreds of updates a day.

The cost of a misstep is low, since the company can trial an update with a few customers and quickly roll back the update if there is a hitch. After a short trial period, the updated software can be put into production for the entire customer base.

Every aspect of the above description is influenced by the company's desire to respond rapidly to its customers. Rapid response implies that the cycle time from concept to deployment must be very short. The developers must develop and deploy changes directly on production servers; there is no time for a handoff to a separate deployment team. The short cycle time also means that changes must be small, so work must be broken down into small pieces that are developed in parallel. Management support is essential. These items and more illustrate what it means to say that customer responsiveness can be driving force for software projects. □

As noted above, engineering projects produce something at an acceptable cost by an agreed-upon time. Cost and time are examples of *constraints* on projects. These constraints might be imposed on the project by customers or by the management of the development team. Either way, the constraints are outside forces on the project. Other examples of external constraints include legal and social constraints.

Example 1.2. Regulatory constraints are the driving force in this example. As we shall see, heavy regulations on large banks ripple through the banks to their suppliers. Regulations result in an emphasis on quality, which prompts multiple levels of testing. The time and cost of testing influences software release schedules.

A supplier of business software releases software semi-annually, on a schedule that is set by its large customers, including highly regulated investment banks. Some of the bigger customers conduct their own rigorous acceptance tests in a lab, before they deploy any supplier software. The trial-deploy cycle takes time and resources, so the customers do not want small updates as soon as they are available; they prefer semi-annual releases.

The supplier's development projects are geared to the customers' preferredrelease cycle. Since releases are relatively infrequent, the cost of a misstep is high. Dedicated product managers therefore stay in close touch with major customers. They share product plans and status and bring back valuable customer feedback. Projects are organized so feedback can be incorporated on a regular basis, while a project is underway. □

Key Project Drivers

Some of the many driving forces (or simply drivers) of software engineering projects appear in Fig. 1.1. Each project must balance these drivers based on

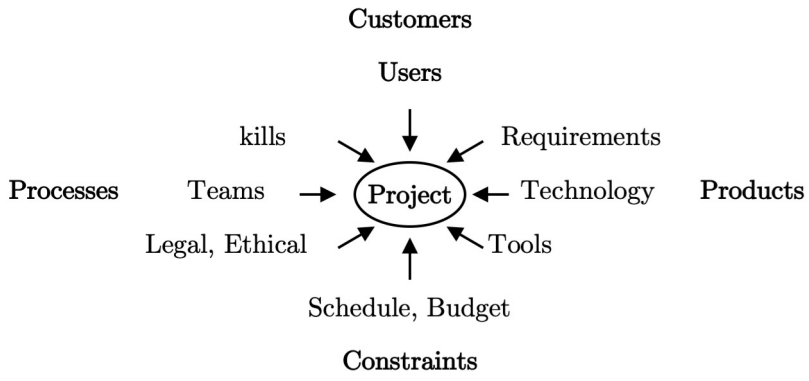


Figure 1.1: Some of the many driving forces that projects must balance. Key drivers are in bold.

its own situation. In Example 1.1, customer needs were the primary driver; in Example 1.2, the underlying driver was regulatory constraints.

Customers, processes, products, and constraints are key representatives of the drivers that projects must balance. A greatly simplified view of their relationships appears in Fig. 1.2. Customers are at the top in the figures—without them, there would be no project. Customer needs and requirements drive the process—they drive the process activities that build a product. Once built, the product must meet customer needs and satisfy any constraints. Constraints are at the bottom in the figures.

The drivers in Fig. 1.2 are tightly linked through their relationships. A change to any aspect of a project ripples through to affect the rest of the project.

In general, a ten-fold change in any aspect of a project is a time to revisit the existing solution: it may no longer work as well, or it may no longer work at all due to the change. A hundred-fold change may require an entirely new solution approach, perhaps even a new organizational structure. Note that there is more than a thousand-fold difference in the pace at which the companies in Examples **??**-**??** deploy software.

Chapter Overview

The rest of this chapter briefly discusses the implications for software engineering of the following statements about software:

- Software systems are intrinsically complex.
- User requirements change during the life of a project.

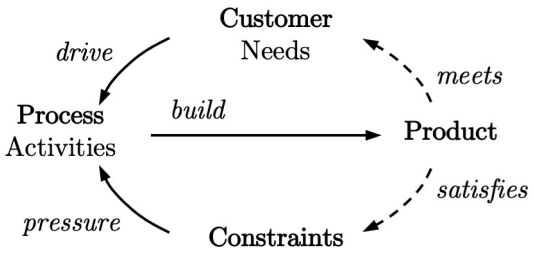


Figure 1.2: The arrows represent relationships between four key drivers of software projects: Customers, Processes, Products, and Constraints. The dashed arrows indicate feedback loops.

- Defects are inevitable.
- Scope. Cost. Time. Pick any two! (You can't simultaneously control a project's functionality, budget, and schedule, so pick any two to fix The other will vary.)

These statements are generally true, based on observation and empirical evidence—such statements are called *ground truths*. Despite occasional exceptions, a ground truth can be treated as a true for all practical purposes.

As we shall see, the intrinsic complexity of software motivates modular design: complexity is easier to deal with if we decompose a complex system into simpler subsystems. Volatility of user requirements motivates iterative development processes, which incorporate frequent user feedback to keep the project on track. The inevitability of defects motivates testing. This chapter concludes with a brief discussion of social responsibility.

1.2 User Requirements Change

Consider the following scenario. A development team talks to users to identify their requirements for a product. For now, let *user requirements* be what users want from a product. The developers then build a product to meet the requirements, only to find that there is a mismatch between user expectations and the product they have built.

When such a mismatch occurs, we say that the requirements have changed. There are two issues:

a) User needs and expectations may indeed have changed between the time the requirements were identified and the time the product was completed.

b) There may have been a gap in the first place between the initial user needs and the requirements that were identified.

In either case, the effect is the same: from a developer's viewpoint, it is convenient to say that requirements have changed.

For some insights into requirements changes, this section examine the following questions:

- Who is the customer? There will likely be multiple classes of users with differing needs.
- What do they want? They may not know what they want until they see a prototype/product.
- Are there any external factors? Changes in external conditions may have repercussions on the project.

Any one of these situations can result in a mismatch between customer expectations and the initial requirements at the start of a project.

Multiple Stakeholders

A *stakeholder* is someone who has a stake or some involvement in a software project. The stakeholders in a flight-control system include not only the airline executives who sign the contract for the system, but the pilots who rely on it to fly the plane, the engineers who must maintain it, not to mention the passengers who are served by the airline. Although developers and marketers also have a stake in the project, we will use term stakeholder to refer to customers and users, unless stated otherwise.

Since different stakeholders can have different needs, it can be a challenge to come up with a single coherent set of requirements. There are two issues: missing requirements and conflicting requirements. Requirements can be missing if some stakeholders are overlooked and their needs not even considered. The issue with conflicting requirements is that it may not be possible to satisfy all stakeholders. Analysis and prioritization of requirements is discussed in Chapter 6. Either missing or conflicting requirements can result in a product that does not meet expectations.

In the following example, the developers found a creative way of satisfying stakeholders with differing needs.

Example 1.3. The makers of a speech-therapy program had a problem. Market testing with a prototype revealed that parents really wanted the program for their children, but found it frustratingly hard to use. Children, meanwhile, had no trouble using the program, but could not be bothered with it: they found it annoyingly like a lesson.

The stakeholders in the speech-therapy program included two groups: frustrated parents and annoyed children. The differing needs of these stakeholders were addressed by changing the program so it was easier to use for parents and was more like a game for children.[3] □

User Uncertainty

What customers say they want can differ from what will satisfy them. There may also be needs that they are unaware of. Even if they did know their needs, they may have only a general "I'll know it when I see it" sense of how a product can help them.

Example 1.4. Netflix is known for its video and on-demand services. Their experience with their recommender system is as follows:

> "Good businesses pay attention to what their customers have to say. But what customers ask for (as much choice as possible, comprehensive search and navigation tools, and more) and what actually works (a few compelling choices simply presented) are very different."[4] □

Uncertainty refers to a known unknown. In this case, we know that there is a requirement, but have yet to converge on exactly what it is. Uncertainty can be anticipated when proposing a solution or design.

External Factors

Unexpected changes are unknown unknowns: we do not even know whether there will be a change. Here are some examples:

- A competitor suddenly introduce an exciting new product that raises customer expectations.
- The customer's organization changes business direction, which prompts changes in user requirements.
- Design and implementation issues during development lead to a reevaluation of the project.
- The delivered product does what customers asked for, but it does not have the performance that they need.
- Customers simply decide that what they asked for is not really what they wanted.

Requirements changes due to external factors can lead to a project being redirected or even canceled.

Dealing with Requirements Changes

Changes in customer needs (hence, in requirements) have repercussions for the development process and for he product, because of the relationships shown in Fig. **??**. We have no control over changes in customer needs, but there are two things we can do.

1. Do as good a job as possible of identifying and analyzing user requirements. Requirements development is a subject in its own right; see Chapter 3.

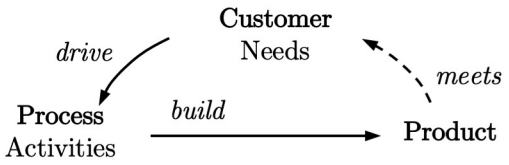


Figure 1.3: The cycle represents the iterative nature of product development based on customer needs.

2. Use a development process that accommodates requirements changes during development. See below for a brief overview of iterative processes.

Iterative and agile process evolve a product, using customer feedback to guide the evolutionary development of the product. Development consists of a sequence of steps, called iterations. Each iteration begins with customer interactions to get feedback on the current version of the product; validate existing requirements; and elicit any new requirements. The developers then build an evolutionary increment to the product and loop back for the next iteration. The continuing feedback helps to keep the project on track, so that the completed product will meet customer expectations.

Iterative and agile processes are discussed in Chapter 2.

1.3 Software is Intrinsically Complex

Let us call a piece of software *complex* if it is hard to understand, debug, and modify. Complexity due to poorly written code is *incidental* to the problem that the code is supposed to address. Meanwhile, if well-written code is hard to understand, its complexity is deeper: it is not due to the code; the complexity is due to the problem addressed by the code. This deeper complexity will remain even if we rewrite the code or use another programming language, because it is *intrinsic* to the problem; that is, it is due to the nature of the problem.[5]

We focus on intrinsic complexity. Architecture is a primary tool for managing software complexity. A software architecture partitions a problem into simpler subproblems. Layered architectures are used often enough that we introduce them in this section.

Sources of Complexity

The two main sources of complexity are scale and the structure/behavior distinction.

- *Scale.* Program size is an indicator of the scale of a problem. A large software system can have hundreds of thousands of lines of code. Sheer size can make a system hard to understand.
- *Structure versus Behavior.* Here, structure refers to the organization of the code for a system. Behavior refers to what the code does when it is run. The challenge is that behavior is invisible, so we do not deal directly with it. We read and write code and have to imagine and predict how the code will behave when the code is run.

Example 1.5. As a toy example of the distinction between structure and behavior, consider the following line from a sorting program:

```
do i = i+1; while ( a[i] < v );
```

In order to understand the behavior of this loop, we need to build a mental model of the flow of control through the loop at run time. The behavior depends on the values of `i`, the elements of the array `a`, and `v` when the loop is reached. Control flows some number of times through the loop before going on to the next line. If any of these values changes, the number of executions of the loop could change.

The structure of this loop identifies the loop body and the condition for staying in the loop. The behavior of the loop is characterized by the set of all possible ways that control can flow through the loop. □

The single well-structured `do-while` loop in the above example was convenient for introducing the distinction between program structure and run-time behavior. Complexity grows rapidly as we consider larger pieces of code; it grows rapidly as decisions are added, as objects are defined, as messages flow between parts of a program, and so on.

To summarize, scale and the predictability of run-time behavior are significant contributors to software complexity.

Software Architecture: Dealing with Program Complexity

Informally, a *software architecture* defines the parts and the relationships between the parts of a system. In effect, an architecture partitions a system into simpler parts that can be studied individually, separately from the rest of the system. (See Chapter 7 for a definition of architecture.)

In this section, the parts are *modules*, where each module has a specific responsibility and a well-defined interface—modules interact with each other only through their interfaces. The complexity of understanding the whole system is therefore reduced to that of understanding the modules and their interactions. For the interactions, it is enough to know the responsibilities of modules and the services they provide through their interfaces. The implementation of the responsibilities and services can be studied separately, as needed.

What is inside a module? The program elements in a module implement its responsibility and services.

support its services that the module provides through its support its interface. As long as its interface remains the same, the internal code for the module can be modified without touching the rest of the system. A module can be anything from the equivalent of a single class to a collection of related classes, methods, values, types, and other program elements. This concept of module is language independent. (Modules are closer to packages than they are to classes.)

For more information about modules, see Chapter 7. In particular, modules can be nested; that is, a module can have submodules. For example, a user-interface module may have submodules for handling text, images, audio, and video—these are all needed for a user interface.

Layered Architectures

For examples, let us turn from a general discussion of architecture to a specific form: layered architectures, which are widely used. In a *layered architecture*, modules are grouped into sets called *layers*. The layers are typically shown stacked, one on top of the other. A key property of layered architectures is that modules in an upper layer may use modules in the layer immediately below. Modules in a lower layer know nothing about modules in the layers above them.

Example 1.6. The layered architecture in Fig. 1.4 is a simplified version of the architecture of many apps. At the top of the diagram is the Presentation layer, which manages the user interface. Below it is the Domain layer, which handles the business of the app. In a ride-sharing app, the Domain layer would contain the rules for matching riders and drivers. Next is the Service Access layer, which accesses all the persistent data related to the app; e.g., customer profiles and preferences. At the bottom, the Platform layer is for the frameworks and operating system that support the app.

The dashed arrows represent who-may-use-who. Modules in the Presentation layer may use modules in the Domain layer, but not vice versa; that is, modules in the Domain layer may not use modules in the Presentation layer. Similar comments apply to the other layers. By design, all the arrows are down arrows. □

The arrows in Fig. 1.4 are included simply to make the may-use relationships explicit. Vertical stacking of layers can convey the same information implicitly. From now on, such down arrows will be dropped from layered diagrams. By convention, if layer A is immediately above layer B in a digram, modules in the upper layer A may use modules in the lower layer B.

The next example is about the modules in a specific app that has a layered architecture like the one in Fig. 1.4.

Example 1.7. A local pool-service company has technicians who go to customer homes to maintain their swimming pools. Each technician has a specific route that changes by day of the week. At each home along their route, the technicians jot down notes for future reference.

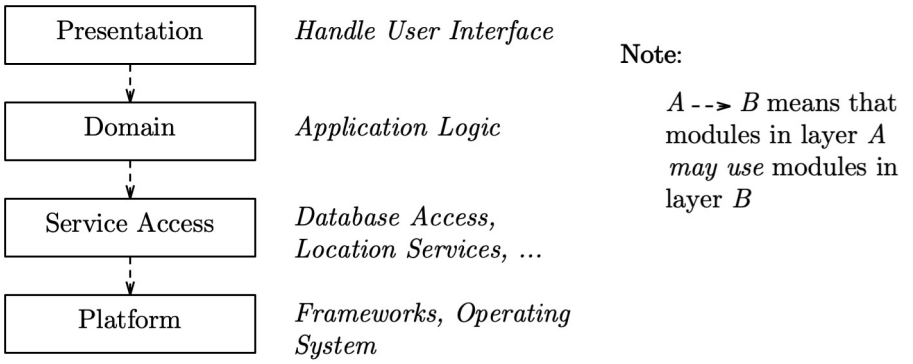


Figure 1.4: A simplified version of the layered architecture of many apps. Boxes represent modules, dashed arrows represent dependencies.

The company wants to replace its current paper-based system by a software system that will support a mobile app for technicians and a web interface for managers. Managers would use the web interface to set and assign routes to technicians. Technicians would use the mobile app for individualized route information, data about each pool along the route, and some customer-account information.

The main modules in the solution correspond to the layers in Fig. 1.4:

- *Data Access Module.* Supports create, read, update, and delete operations related to customers, routes, technicians, and pool-service history. The database schema is private to this module, as is the nature of the data repository (it happens to be in the cloud).
- *Domain Module.* Serves as an intermediary between the user interfaces and the data repository.
- *Mobile Interface Module.* Provides technicians with the data they need along their assigned route; e.g., customer account information and pool-service history. Also supports note-taking.
- *Web Interface Module.* Supports route management, work assignments, access to pool service history, and customer service.
- *Presentation Module.* The presentation module has submodules for the mobile app and the web interface. □

1.4 Defects are Inevitable

In a series of experiments, the psychologist George A. Miller found that people can accurately "receive, process, and remember" about seven chunks or units of

information. The units in his experiments included bits, words, colors, tones, and tastes.[6] Beyond about seven chunks, confusion and errors set in. Software involves much much larger numbers of statements, variables, functions, messages, objects, control-flow paths, ... Hence the assertion that defects are inevitable.

Miller's experiments also have implications for how to deal with complexity: design software systems in chunks that a developer can understand in relative isolation from the rest of the system. Such chunks correspond to modules;; see Section 1.3.

Testing for defects is a very important part of software development. By testing and after every change, developers can catch and fix errors early, thereby increasing confidence in the correctness of the change. Testing early and often can actually shorten overall development time. Regrettably, testing does not always get the priority it deserves. In the rush to complete coding, developers put off thorough testing until later. Later, as a deadline looms, time runs out before testing is complete.

Programs Have Faults, Their Computations Have Failures

Defect, fault, bug. Failure, malfunction, crash. These are all terms to indicate that something is wrong with a program.

Defect and fault are synonyms: *fault* refers to something that is wrong in the program text (source code), the documentation, or some other system artifact. The term fault also applies to any deviations from programming style guidelines. Typically, fault refers to a flaw in the static program text.

Failure refers to unexpected behavior when a program is run. The incorrect behavior may occur only some of the time, under certain conditions, or on certain inputs.

Saying, "programs have faults, their computations have failures," is equivalent to saying, "code can have faults, its runs can have failures." A *computation* is the sequence of actions that occur when a program is run on given inputs. The next example illustrates the distinction between faults and failures. It also illustrates the distinction between programs and their computations.

Example 1.8. There had been 300 successful trial runs of the airborne guidance system for NASA's Atlas rocket. Then, an Atlas Athena rocket was launched, carrying Mariner 1, an unmanned probe to the planet Venus. Shortly after launch, signal contact with the ground was lost. The rocket wobbled astray and was destroyed by the safety officer 293 seconds after launch.

The developers had planned for loss of signal contact, but the code had a fault. The airborne guidance system was supposed to behave as follows:

if not in contact with the ground **then**
do **not** accept course correction

Due to a programming error, the **not** was missing. The guidance system blindly steered the rocket off course, until the rocket was destroyed.

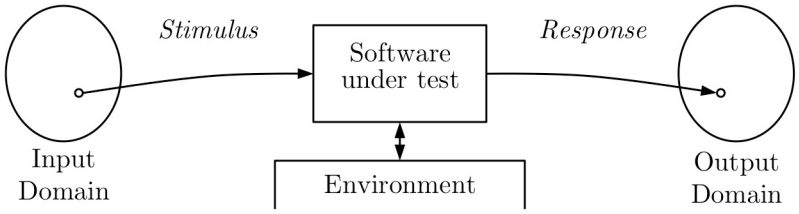


Figure 1.5: Software can be tested by applying an input stimulus and evaluating the output response.

The missing **not** was a fault in the source code. This fault lay undetected through 300 successful trial runs. During these trials, the fault did not trigger a failure. The failure occurred when contact with the ground was lost and control reached the fault in the code. Stated differently, the program had a fault; 300 computations did not trigger a failure.[7] □

Introduction to Testing

Software testing refers to running a program in a controlled environment to check whether its computations behave as expected. A test input *fails* if its computation fails. If a test fails, then the source code must have a fault.

This introduction to testing is in terms of the four main elements in Fig. 1.5:

- *Software Under Test.* The software under test can be a code fragment, a module, a subsystem, a self-contained program, or a complete hardware-software system.
- *Input Domain.* The input domain is the set of possible test inputs. A test input is more than a value for a single variable–=it provides values for all the relevant input variables for the software under test. The input need not be numeric; e.g., it could be a click on a web page or a signal from a sensor.
- *Output Domain.* The output domain is the set of possible output responses or observable behaviors by the software under test. Examples of behaviors include the following: produce text outputs; display an image on a screen; send a request for a web page.
- *Environment.* Typically, the software under test is not self contained, so an environment is needed to provide the context for running the software. For example, the software under test may use a package or call a function external to the software. The role of the environment is to provide a (possibly dummy) package or external function.

The following questions capture the main issues that arise during testing:[8]

- How to stabilize the environment to make tests repeatable?
- How to select test inputs?
- How to evaluate the response to a test input?
- How to decide when to stop testing?

The main barrier to testing is test selection. Once tests are selected, automated tools make it convenient to re-run all tests after every change to the program. See Chapter 10.

1.4.1 Black-Box and White-Box Testing

During test selection, we can either treat the software under test as a black box or we can look inside the box at the source code. Testing that depends only on the software's interface is called *black-box testing*. Testing that is based on knowledge of the source code is called *white-box testing*.

Typically, white-box testing is used for smaller units of software and black-box testing is used for larger segments that are built up from the units. Black-box testing is the only option if the source code is not available and all we have is an executable version of the software. Chapter 10 has more information on testing.

1.5 Scope. Cost. Time. Pick Any Two!

A *project* is a set of activities with a start, a finish, and deliverables. The deliverables can be an artifact or an event; e.g., a product like an app; a demo at a conference; a service delivered from the cloud; an assessment report. *Project management* consists of planning, organizing, tracking, and controlling a project from initial concept through final delivery[9]

The Iron Triangle

All projects balance competing priorities and face resource constraints. The *Iron Triangle* (also known as the *Project Management Triangle*) illustrates the typical constraints faced by projects: scope, time, and cost; see Fig. 1.6.[10]

Scope refers to two things: (1) the functionality to be delivered or the customer requirements to be met by the project; and (2) the quality of the product that is delivered. Quality is interpreted broadly to include not only free of defects, but also attributes of the product such as security, performance, and availability. The two vertices at the base of the triangle represent time (schedule) and cost (budget) constraints.

The edges of the triangle represent the connections between the constraints. For example, if scope increases, then time and/or cost are bound to be affected. Similarly, if time or cost are reduced, then scope is bound to be affected.

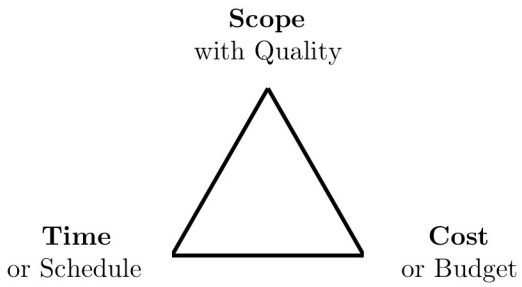


Figure 1.6: The Project Management Triangle, with constraints at the vertices. A variant of this diagram attaches the constraints to the edges of the triangle, instead of the vertices.

Schedule overruns have typically been accompanied by cost overruns or scope reductions.

In practice, there is rarely enough time or budget to deliver the full scope with quality. The challenge of meeting the triple constraints simultaneously accounts for the quip, "Fast. Cheap. Good. Pick any two!"

Example 1.9. The Iron Triangle is convenient for illustrating the experience of a company that will remain nameless. In a rush to get new products to market, the executives of Company X pressed the development teams for a 15% reduction in project schedules, compared to similar past projects. The budget remained the same. The teams responded by spending fewer days, on average, on every activity: design, coding, testing.

Once the new products were released, it became evident that functionality and quality had suffered. Early customers complained about missing features and product defects. The company reacted to the trouble reports by issuing upgrades to improve the products that had already been delivered. Eventually, the problems with the products did get fixed, but the company's reputation had been tarnished.

In a bid to repair its reputation, the company prioritized quality over schedule for its next set of it. □

Project management is a big subject, consisting of planning, organizing, monitoring, and delivering a project, from start to finish. The discussion of the Iron Triangle in this section touches on a small part of the subject. To a large extent, the entire book is relevant to software project management since a project manager needs to know about software development in order to manage it.

1.6 Social Responsibility: A Cautionary Tale

In 1986, a patient died after radiation treatment by a software-controlled medical device called Therac-25. It was one of six known accidents with the device.[11] The Therac-25 accidents are a cautionary tale of the societal impact of software projects.

The professional societies ACM and IEEE have published a Software Engineering Code of Ethics and Professional Practice. The Code recommends that software engineers act in the public interest, in the best interests of clients and employers, and maintain integrity and independence in their professional judgment.

This section uses the Therac-25 case to illustrate the public and product principles of the Code. The Therac-25 case is the first known case of a software-related fatality. Based on a thorough investigation, basic software engineering principles had apparently been violated during the development of the software.

The ACM-IEEE Code of Ethics and Professional Practice

The short version of the code appears in Fig. 1.7. The preamble to the short version warns that the aspirational principles in Fig. 1.7 are to be taken with the examples and details in the full version of the code. Neither the aspirations nor the details stand on their own:

> Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.[12]

The issue of social responsibility is very much with us. The Code does not deal with true ethical dilemmas; e.g., if an accident seems inevitable, should a self-driving car prioritize the life of the driver or that of a pedestrian? The Code does apply to negligence and misconduct; e.g., to malware.

1.6.1 Case Study: Therac-25

When the patient came to the East Texas Cancer Center for his ninth treatment on March 21, 1986, more than 500 patients had been treated on the Therac-25 radiation therapy machine, over a period of two years. The planned dose was 180 rads. Nobody realized that the patient had actually received between 16,500 and 25,000 rads over a concentrated area, in less than 1 second. He died five months later due to complications from the massive overdose.

Malfunction 54

On March 21, when the technician pressed the key for treatment, the machine shut down with an error message: “Malfunction 54,” which was a “dose input 2” error, according to the only documentation available. The machine’s monitor

1. PUBLIC. Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER. Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT. Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT. Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT. Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION. Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES. Software engineers shall be fair to and supportive of their colleagues.
8. SELF. Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Figure 1.7: Short version of the ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice.

showed that a substantial underdose had been delivered, instead of the actual overdose. The machine was shut down for testing, but no problems were found and Malfunction 54 could not be reproduced. The machine was put back in service within a couple of weeks.

Four days after the machine was put back in service, the Therac-25 shut down again with Malfunction 54 after having delivered an overdose to another patient, who died three weeks later. An autopsy revealed an acute high-dose radiation injury.

Synchronization Problems and Coding Errors

After the second malfunction, the physicist at the East Texas Cancer Center took the Therac-25 out of service. Carefully retracing the steps by the technician in both accidents, the physicist and the technician were eventually able to reproduce Malfunction 54 at will. If patient treatment data was entered rapidly enough, the machine malfunctioned and delivered an overdose. With experience, the technician had become faster at data entry, until she became fast enough to encounter the malfunction.

The accidents in Texas were later connected with prior accidents with Therac-25, for a total of six known accidents between 1985 and 1987. After the 1985 accidents, the manufacturer made some improvements and declared the ma-

chine fit to be put back into service. The improvements were unrelated to the synchronization problems that led to the malfunctions in Texas in 1986.

On January 17, 1987, a different software problem led to an overdose at the Yakima Valley Memorial Hospital. The Yakima problem was due to a coding error in the software.

1.6.2 Lessons for Software Projects

The following discussion explores what software projects can do right, not just what went wrong with Therac-25. The idea is to highlight a few clauses from the Code of Ethics and Professional Practice that are relevant for any project. Note that the clauses in the Code are not meant to applied individually, in isolation. The Code provides fundamental principles for guiding thoughtful consideration of the merits of a situation. "The Code is not a simple ethical algorithm that generates ethical decisions."

Act in the Public Interest

The full version of the Code includes

> '1.03. [Software engineers shall, as appropriate] approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good."

With Therac-25, before the massive overdose in the East Texas Cancer Center, there were reports of unexpected behavior from other sites. Each time, the manufacturer focused narrowly on specific design issues, without getting at the root-cause of the problem or the overall safety of the medical device.

Lesson. Act in the public interest at every stage of a software project, from requirements, to design, to coding and testing, even to maintenance. The software needs to be safe not only when it is first delivered, but also whenever it is modified. Get to the root cause of any problem.

Design Defensively

The product principle of the Code begins with

> 3. "Software engineers shall ensure that their products and related modifications meet the highest professional standards possible."

Could the accidents in East Texas have been prevented? Possibly. Therac-25 did not have a hardware interlock to prevent an accidental overdose due to a software problem. The software was reused from an earlier machine, called Therac-20. A related software problem existed with Therac-20, but that earlier machine had a hardware interlock to guard against an overdose

Lesson. Design for fault tolerance, so a failure in one part of the system is contained rather than cascaded.

Test the System in its Environment

The product section of the Code includes

> "3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work."

For Therac-25, testing was inadequate. Documentation was lacking. The first safety analysis did not include software.

The failure in Texas was due to an interaction between the human technician and the machine. Specifically, the Therac-25 software had concurrent tasks and Malfunction 54 was due to a task synchronization problem. The malfunction occurred only when the technician entered data fast enough.

Lesson. A system can fail due to interaction between its components or between the system and its environment. Test the system in its environment.

1.7 Conclusion

Key Drivers of Software Engineering Projects. Software engineering is the application of engineering processes to the building of software products. Engineering projects have customers; their needs drive what is built. Projects also have constraints, such as budget, schedule, business, legal and social/ethical constraints. Customers, processes, products, and constraints are therefore key drivers of software projects.

Ground Truths About Software. One measure of the wide diversity of software projects is development interval, which is the time between initial concept and final delivery or deployment of a product. Development intervals can range from a few months to a few days, depending on a project's unique situation, Tech companies that have control over their servers typically have short development intervals, measured in days. Regulated companies, on the other hand, may set longer development intervals, measured in months, to allow for extensive lab trials prior to deployment. The development interval influences everything about a project, from level of automation to organizational structure.

Given the diversity of projects, we might well ask: what are the underlying principles or ground truths that apply across projects? A ground truth is a statement that is generally true, based on empirical evidence. This chapter explores the following ground truths:

- Requirements change during the life of a project.
- Software systems are intrinsically complex.

- Defects are inevitable.
- Scope. Cost. Time. Pick any two! (You can't simultaneously control a project's functionality, budget, and schedule, so pick any two to fix—the other will vary.)

Requirements Changes. User requirements change for three reasons. First, there may be multiple users with diverging or conflicting needs. Second, users may not know what they want until they experience a working version of the product. Third, conditions can change, with external factors then leading to a reappraisal of user needs. In any of these cases, a product built to the initial requirements may fail to meet customer expectations upon delivery. From a developer's perspective, requirements will have changed. The way to deal with changing requirements is to use an iterative process that incorporates ongoing customer feedback, as discussed in Chapter 2.

Software Complexity. Software complexity is due largely to the distinction between a program and its behavior. Program text is visible, but understanding is based on behavior, which is invisible—behavior refers to actions that occur inside a machine at run time. The separation between programs and computations has implications for both program design and testing. Modular designs partition a program into subprograms that are easier to understand; see the discussion of software architecture in Chapter **??**.

Defect Removal. People make mistakes that lead to defects in programs. Mistakes occur because software complexity outstrips our limited human ability to deal with complexity. Defects are therefore inevitable.

Testing is a key technique for finding and removing defects; see Chapter 10. Testing by itself is not enough to ensure product quality. The combination of testing with reviews and static analysis (Chapter 9) is very effective for defect removal.

Resource Constraints. A project has a start, a finish, and deliverables. In practice, there is never enough time or budget to deliver the full product functionality, with quality (the scope). The Iron Triangle, also known as the Project Management Triangle, has scope, time, cost, and scope at its three vertices. The message is that project can control any two, but cannot simultaneously control all three: scope, cost, and time.

]subsubSocial Responsibility Finally, software projects have a social and ethical responsibility to do no harm. The first known software-related fatalities were due to a malfunction in a medical device called Therac-25; the device delivered fatal doses of radiation. A careful investigation concluded that the makers of the device did not follow generally accepted software engineering practices.

Origins of the Term Software Engineering

Software engineering emerged as a distinct branch of engineering in the 1960s. The term software engineering dates back to (at least) 1963-64. Margaret Hamilton, who was with the Apollo space program, began using it to distinguish software engineering from hardware and other forms of engineering. She recalls,

> "Software, during the early days of [the Apollo] project, was treated like a stepchild and not taken as seriously as [hardware] ... I fought to bring the software legitimacy so that it (and those building it) would be given its due respect"[13]

At the time, hardware was precious: an hour of computer time cost hundreds of times as much as an hour of a programmer's time.[14] Code clarity and maintainability were often sacrificed in the name of efficiency. As the complexity and size of computer applications grew, so did the importance of software.

Software engineering was more of a dream than a reality when the term was chosen as the title of a 1968 NATO conference. The organizers chose the title to be

> "provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering."[15]

Since then, much has changed.

Exercises for Chapter 1

Exercise 1.1. Most definitions of software engineering boil down to the application of engineering methods to software. For each of the following definitions answer the questions:

- What is the counterpart of "engineering methods" in the definition?
- What is that counterpart applied to?
- Does the definition have any elements that are not covered by the above questions? If so, what is it and how would you summarize its role?

The definitions are as follows.[16]

a) *Bauer*: "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works on real machines."

b) *SEI*: "Engineering is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind. Software engineering is that form of engineering that

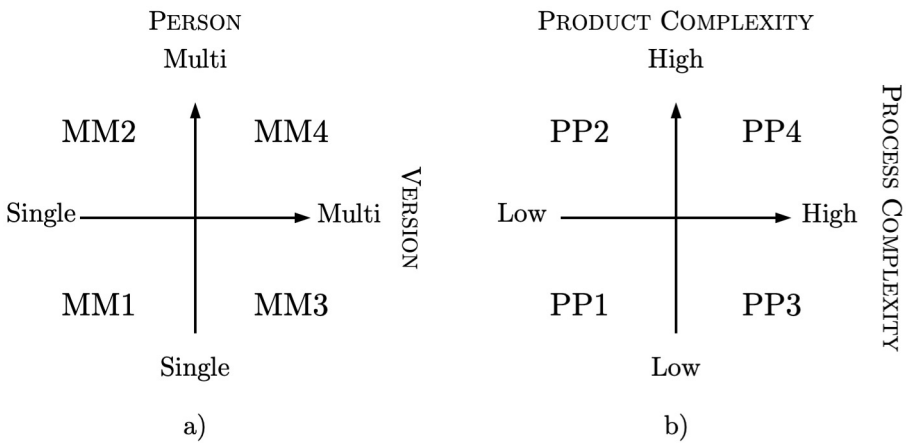


Figure 1.8: Diagrams for Exercise 1.2.

applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems."

c) *IEEE* Software engineering is the "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

Exercise 1.2. In order to distinguish software engineering from programming alone, consider the following characterization:

> Software engineering is multi-person or multi-version development of software.[17]

By contrast, programming by itself is single-person development of single-version programs; see Fig. 1.8(a). Assume that "multi version" implies that the versions are built one after the other, and that each version builds on the last.

a) Relate the diagram in Fig. 1.8(a) to the diagram of product versus process complexity in Fig. 1.8(b). Use the MM and PP labels to briefly describe the relationships, if any, between the quadrants in the two diagrams.

b) Give examples to illustrate your answer to part (a).

Exercise 1.3. For each of the following sixteen driving forces on software projects, choose the "best fit" with the four key drivers in Fig. 1.2: Customers, Processes, Products, and Constraints. The drivers are listed in priority order. If a force seems to fit under more than one driver, choose the earliest driver

in the priority order. Briefly justify your grouping of each force under a key driver.[18] (The forces are in alphabetical order.)

Compatibility	Functionality
Complexity	Legal
Context	Mission
Cost	Performance
Deployment	Reliability
Development	Safety
Ethical	Schedule
Evolution	Security

Exercise 1.4. This exercise deals with the distinction between programs and computations. The program is the following C code for rearranging the elements of an array:

```
int partition(int i, int j) {
   int pivot, x;
   pivot = a[(i+j)/2];
   for(;;) {
      while( a[i] < pivot ) i = i+1;
      while( a[j] >= pivot ) j = j-1;
      if( i >= j ) break;
      x = a[i]; a[i] = a[j]; a[j] = x;
   }
```

Consider the function call `partition(0,4)`, where the relevant array elements are as shown below. In each case, what are the computation paths through the function body?

a) `53`, `53`, `53`, `53`, `53`.

b) `53`, `58`, `53`, `84`, `59`.

Exercise 1.5. The Python code in this exercise is deliberately complex. It computes 24, the factorial of 4, but it does so in a way that can be hard to understand. How does it work?[19]

```
def f(g, m):
    if m == 0:
        return 1
    else:
        return m*g(g,m-1)

def main():
    print f(f,4)
```

Exercise 1.6. The pool-service company in Example 1.7 has two user interfaces: a mobile interface for technicians and a web interface for managers. Both technicians and managers also rely on an external map service to display routes. Modify the architecture diagram in Fig. 1.4 to show the two user interfaces and both an external data repository and a mapping service.

Exercise 1.7. Based on a thorough investigation of the Therac-25 accidents, the following software engineering practices were violated:[11]

- Specifications and documentation should not be an afterthought
- Establish rigorous software quality assurance practices and standards
- Keep designs simple; avoid dangerous coding practices
- Design audit trails and error detection into the system from the start
- Conduct extensive tests at the module and software level
- System tests are not enough
- Perform regression tests on all software changes
- Carefully design user interfaces, error messages, and documentation

How would each of these practices have helped avoid the Therac-25 accidents? Provide 2-3 bullet items per practice.

Chapter 2

Software Development Processes

> "Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them."
>
> — *Doug McIlroy, Elliot Pinson, and Berkley Tague, 1978. Maxims from their foreword to a collection of papers about Unix and its programming culture.*[1]

2.1 Introduction

The selection of a development process is one of the earliest decisions during the life of a software project. A process orchestrates the workings of a team: it guides what they do to build a product; how they interact with customers; and so on. A process touches every aspect of a software project.

A development team's values and culture can be as important to the success of a project as "who does what." Values complement processes. Processes provide rules for some aspects of development and leave other aspects unspecified. Values like the following can guide decisions that are not covered by an explicit process:

- Have every program do one thing well.
- Build in days and weeks to get early customer feedback.

- Have clean working software at the end of each iteration.

After an overview of processes, this section introduces the values and culture of agile software development. The section concludes with the concept of risk. Projects face many risks, such as the risk that the requirements for the project might change; or the risk that a design might fail to deliver the desired performance. Processes can be tailored to manage risks.

2.1.1 An Overview of Processes

In simple terms, a process guides "who will do what by when and why" to achieve a goal.[2] Processes can be defined for any aspect of software development, ranging from building an entire product to conducting a code review, to deploying a system update. Unless stated otherwise, we deal with processes for building a product from start to finish.

Definition of Process

Often, a process provides rules and guidelines only for what a team must do (their activities) and for what they produce (the artifacts). Software development activities include design, coding, and testing. Examples of artifacts include: code; a software architecture produced by a design activity; and a report produced by a code-review activity.

In addition to activities and artifacts. the following definition includes additional aspects of a process. In practice, these additional aspects may be left implicit or unspecified. Here, implicit means that they are known to the team even though they are not mentioned in the written or stated process. Unspecified means that they are left for the team to handle, as it see fit.

Here is the definition. A *process* is a systematic method for meeting a goal by doing the following:

- identifying and organizing a set of *activities* to achieve the goal;
- defining the *artifacts* (deliverables) produced by each activity;
- determining the *roles* and skills needed to perform the activities;
- setting *criteria* for progressing from one activity to the another; and
- specifying *events*, say, for planning and tracking progress;
- all with the context of the values and culture of the team.

Processes are typically represented by "box-and-line" diagrams of the activities and the order in which the activities are performed. Boxes represent activities. Arrows represent the progression from one activity to the next; see Fig. 2.1. (Scrum diagrams, on the other hand, focus on artifacts and events; see Section 2.3.)

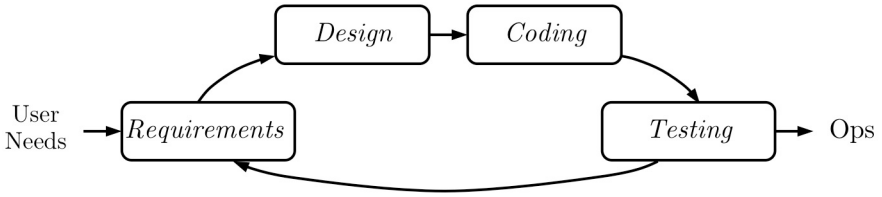


Figure 2.1: An iterative development process. Arrows represent the progression through the activities.

Example 2.1. The diagram in Fig. **??** is for an iterative process. The process starts with *Requirements*, progresses through to *Testing*, and then either completes or cycles back to begin the next iteration through the activities. Such cycling is a characteristic of iterative processes, which build a product incrementally; see Section 2.2.

For completeness, here are one-line descriptions of the activities in Fig. 2.1:

- *Requirements*. Identify user requirements for what to build.
- *Design*. Outline or refine a solution.
- *Coding*. Implement some part of the design.
- *Testing*. Run test cases to verify that the implementation is correct.

Activities are specific to a process, so a different iterative process might have slightly different interpretations of terms like requirements, design, coding, and testing; e.g., it might specify the activities in greater detail. □

The definition of process allows activities to overlap. For example, coding of a module may begin before the design of other modules is complete. Coding and testing may overlap or be integrated; tests can be run as soon as some code is written or a change is made.

Furthermore, activities can be either manual or automated. Design is manual; testing is often automated. Continuous deployment relies heavily on automated tests to rapidly verify a software update before it is deployed.

Terminology: Models, Frameworks, Methods

Classes of processes are sometimes called *models* or *frameworks*; e.g., "iterative model" for the class of iterative processes; or "Scrum framework" for processes that follow the rules of Scrum.

Models and frameworks may provide the core of a process, as opposed to providing a complete process. The core provides guidelines for tailoring a process to suit a given project. For example, the diagram in Fig. 2.1 does not

specify the length of an iteration: Is it a week? Is it a month? Several months? The length is up to the project.

In practice, he term *method* may either refer to a process class or a specific process. For example, Scrum is billed as a framework and is also referred to as an agile method.

2.1.2 Development Culture and Values

Values take center stage with agile methods. Values can guide decisions about what to do when during development. Processes are in the background. A small team of skilled individuals, with a strong set of values, can successfully develop software with a minimum of formal process.

As an early example (1970s), the Unix operating system and related software tools were developed in a culture that relied on values rather than on processes. Development was guided by *maxims*—values we might call them—like the following:

- Design and build software for early user feedback.
- Refactor by rebuilding the "clumsy parts."
- Make every program do one thing well.
- Expect programs to be used in combination with others; e.g., in a pipeline.[3]

Echoes of the Unix maxims resonate in agile values (below) and dataflow architectures (Chapter 8).

The Agile Manifesto

The term *agile* was coined by a group of self-described "independent thinkers about software development.' They met in 2001 to explore "an alternative to document driven, heavyweight processes." The group included proponents of a range of existing process models, including Scrum and Extreme Programming (XP); Scrum is discussed in Section 2.3, XP in Section 2.4. They were all technical people with extensive development experience. They found common ground in a set of values embodied in the Agile Manifesto, reproduced in Fig. 2.2.[4]

The group put "Individuals and interactions" first: he team comes first, before process and technology. The underlying assumption is that the team consists of "good" people, who value technical excellence and work together on a human level.

> "The point is, the team doing work decides how to do it. That is a fundamental agile principle. That even means if the team doesn't want to work in an agile way, then agile probably isn't appropriate in that context, [not using agile] is the most agile way they can do things .... "[5]

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	*over*	processes and tools
Working software	*over*	comprehensive documentation
Customer collaboration	*over*	contract negotiation
Responding to change	*over*	following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Figure 2.2: The Agile Manifesto.

What is an Agile Method?

The agile emphasis on values does not preclude the use of processes, as we shall see in Sections **??** and 2.4. Let the term *agile method* or *agile process* apply to any software development process that conforms with the values in the Agile Manifesto and the principles behind it.[6] Thus, agile process emphasize

- self-organizing skilled teams;
- satisfying customers through collaboration;
- delivering working software frequently (in weeks, not months);
- accommodating requirements changes during development; and
- valuing simplicity and technical excellence.

This characterization covers all of the processes that were represented when the Manifesto was created; e.g., Scrum and XP, which date back to the mid-1990s. The characterization also allows for other pre-Manifesto agile processes; that is, projects that we would now call agile. As the Agile Alliance notes,

> "A lot of people peg the start of Agile software development, and to some extent Agile in general, to a meeting that occurred in 2001 when the term Agile software development was coined.
>
> However, people started working in an Agile fashion prior to that 2001 meeting."[7]

2.1.3 Project Risks

Processes can be customized or *tailored* to suit a project's unique needs and challenges. For example, if the project faces a customer problemthat is new to the team, one option is to add a quick prototyping activity, so the development

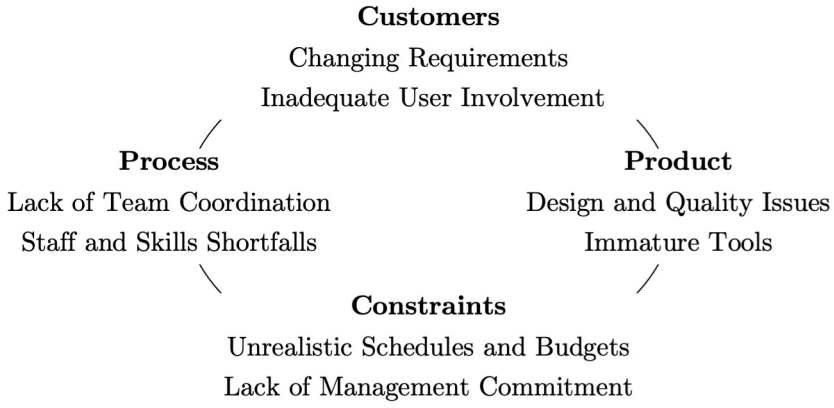


Figure 2.3: Sample project risks.

team can explore potential solutions. As another example, if there are concerns that a design may fail to deliver the desired performance, a design review might be added to bring in expert opinions.

Before making any adjustments to the development process, we need to ask: What indeed are the challenges or risks associated with this project? Even a rough risk assessment can help expose aspects of a project that merit special attention. The process can then be tailored to manage the risks. In the examples above, one process was tailored by adding a prototyping step; the other was tailored by adding a design review. Each project has its own risks and risk tolerance. The risks associated with a movie recommender are surely different from the risks associated with a a medical device.

An intuitive notion of risk will suffice for our purposes. At the start of a project, risk predictions are likely to be guesstimates, at best Furthermore, software developers seldom use formal models of risk. A rough risk assessment can be done as follows:

1. Identify potential risk factors. Note that any aspect of a project could be subject to risk. The sample risks in Fig. 2.3 provide a starting point.

2. For each risk factor, ask questions to classify the risk as high, medium, or low. See below for examples of questions.

Here are some sample questions to help with a rough risk assessment:

- On requirements, are users and developers close to agreement, getting to agreement, or far from agreement? Risk increases if a project starts before agreement is reached with all key stakeholders.

- How critical is the application: high, medium, or low? With critical projects, design, quality, and regulations merit special attention.
- Is the technology, known to the team, new to the team, or new to the world? Known technology carries lower risks.

Similar questions arise during the prioritization of user requirements; see Chapter 6.

Quantifying Risk as a Cost

If needed, risk factors such as the ones in Fig. 2.3 can be quantified and compared by treating each risk as a cost: it is the cost of something going wrong. For example, there is a potential cost associated with missing a deadline or with having to rework a design. Let us refer to something going wrong as an event. , The risk of an event is as follows:

$$\textit{Risk} \text{ of event} = (\textit{Probability} \text{ of event}) \times (\textit{Cost} \text{ of handling event})$$

For a predictable or known event, its probability can be estimated, based on intuition and experience with similar projects. Unforeseen or unknown events can be accounted for by building in a margin of error, again guided by experience.

2.2 Iterative Processes

- *Highlights.* Iterative processes evolve a product, using design-build-test cycles that are called iterations. Each iteration incorporate customer feedback, so the finished product will meet customer expectations upon delivery.
- *Risks Addressed.* Regular customer feedback addresses the risk of requirements changes, which is perhaps the biggest risk during software development.
- *Caveats.* Requirements risk is reduced, but not eliminated. To avoid redesign and rework, the design must be flexible enough to accommodate changes due to customer feedback. It is up to individual projects to manage other risks, such as a brittle code base or inadequate testing.

2.2.1 Overview of Iterative Processes

An *iterative process* is characterized by

1. a sequence of iterations that develop increasingly functional versions of a product; and
2. customer feedback during each iteration to guide the evolutionary development of the product.

The idea is to start by building an early working version that provides just enough functionality for customers to extrapolate what they will get from the finished product. With each iteration, the software does more of what customers want. Their feedback on previous iterations guides what gets added or changed in subsequent iterations. Iterative processes can manage requirements changes. Each iteration revisits requirements and includes design, coding, and testing activities; see Fig. 2.1.

Agile development is iterative, so the above comments carry over to agile processes; see Sections 2.3-2.4. The defining difference between agile and iterative processes is the agile emphasis on team culture and values.

Short iterations and continuing user feedback are important but not sufficient for ensuring the success of a project. This section considers a pair of projects: both used the same development processes; both were done by essentially the same team. The first was wildly successful; the second failed o live up to its promise. The second illustrates that iterative processes reduce the risk of changing requirements, but they do not eliminate it. Furthermore, projects can succumb to other risks related to design, quality, and skills shortfalls. The pair of projects therefore serves as a good case study. The lessons remain relevant today because projects are not always run under perfect conditions.

2.2.2 Netscape 3.0: A Successful Iterative Project

Mozilla was the code name for the Netscape browser project. The Mozilla Foundation and the Firefox browser descend from Netscape's code base.

In the early days of the Web, Netscape Navigator was the dominant browser, with 70% market share. Microsoft appeared to have missed the Internet "Tidal Wave" until it unveiled its Internet strategy on December 7, 1995. The company compared itself to a sleeping giant that had been awakened, and launched an all-out effort to build a competing browser, Internet Explorer 3.0.[8]

Example 2.2. In the race with Microsoft, time to market was paramount for the Navigator 3.0 project. The development process emphasized quick iterations; see Fig. 2.4, where the shaded boxes represent iterations. There were six beta releases between the start of the project in January 1996 and the final release of the product in August.

The project faced both user uncertainty and a competitive threat:

- *User Uncertainty.* The initial requirements were based on extensive interactions with customers; however, there was uncertainty about whether customers would like the design and usability of the features.[9]
- *Competitive Threat.* The team carefully monitored beta releases of Microsoft's Explorer 3.0, ready to change requirements dynamically to keep Navigator 3.0 competitive with Explorer 3.0.[10]

The Navigator 3.0 project began with a prototype that was released internally within the company as Beta 0. The prototype was followed by quick

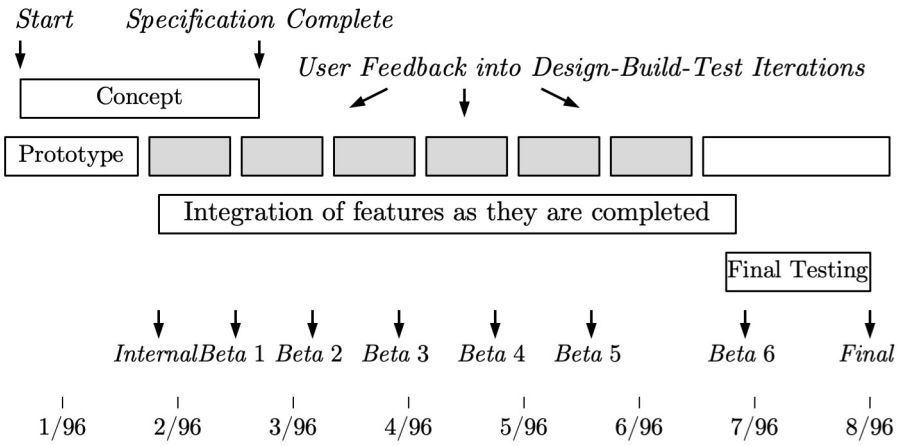


Figure 2.4: A stylized version of the iterative process for the Netscape Navigator 3.0 web browser project.

design-build-test iterations. Each iteration led to an external beta release, available for public download. The beta releases of working software elicited valuable user feedback that guided the content of the next beta version.

Navigator 3.0 and Explorer 3.0 both hit the market in August 1996. □

2.2.3 Netscape 4.0: A Troubled Iterative Project

Following the success of its Navigator 3.0 web browser, Netscape launched a project to build the next version, called Communicator 4.0. Both 3.0 and 4.0 used the same iterative process. While the 3.0 project went well, the 4.0 project produced a poor quality product.

Example 2.3. The Netscape Communicator 4.0 project faced multiple risks.

Customers. With 4.0, Netscape shifted its business strategy from focusing on individual consumers to selling to enterprises—enterprises include businesses, non-profits, and government organizations. The senior engineering executive later described the shift as "a complete right turn to become an airtight software company." Enterprises have much higher expectations for product quality.[11]

The team took on new features, including email and groupware. Well into the 4.0 project, the requirements for the mailer changed significantly: the company changed the competitive benchmark. As the engineering executive put it, "Now that's an entire shift!"

The groupware features were unproven and were not embraced by customers. Three quarters of the way through the project, a major new feature was added.

Product. The project had technology issues related to both the code for the system and with the tools to build the system. Communicator 4.0 was built on the existing code base from Navigator 3.0. The existing code base needed to be re-architected to accept the new features, but the schedule did not permit a redesign.

With respect to tools, the team chose to use Java, so the same Java code would run on Windows, MacOS, and Unix. Java was relatively new at the time and did not provide the desired product performance. (Since then, Java compilers have improved significantly.)

Process. The team faced skills shortages. With multiple platforms to support (Windows, MacOS, Unix), the team did not have enough testers.

When Communicator 4.0 was released in June 1997, it faced quality problems. Ongoing customer feedback addressed some of the risks related to requirements, but there were other risks that were not addressed, related to the shortcomings of the existing code base, the performance issues with Java, a tight schedule, and insufficient testing. □

2.3 The Scrum Framework

- *Highlights.* The rules of Scrum specify roles, events, and artifacts: roles for team members; events for planning and reviewing development work; and artifacts that the team works with. The rules are geared to iterations, called sprints, of one month or less. Within the structure provided by the rules, developers have the flexibility to choose how they develop software.[12]
- *Risks Addressed.* Scrum's rules address risks related to team coordination. Sprints (iterations) the risks related to requirements changes.
- *Caveats.* Note that Scrum is a framework that structures team interactions, not a complete process. Projects that adapt the rules of Scrum to their needs remain responsible for managing other risks; e.g., related to design, coding and testing.

2.3.1 Overview of Scrum

Scrum is by far the most widely practiced of all agile methods. Surveys of agile projects have shown that over half of all projects use Scrum by itself. Another 25% use a hybrid of Scrum with another process.[13]

Scrum is the work of Ken Schwaber and Jeff Sutherland. The first public presentation of Scrum was at a conference in 1995, long before the Agile Manifesto.

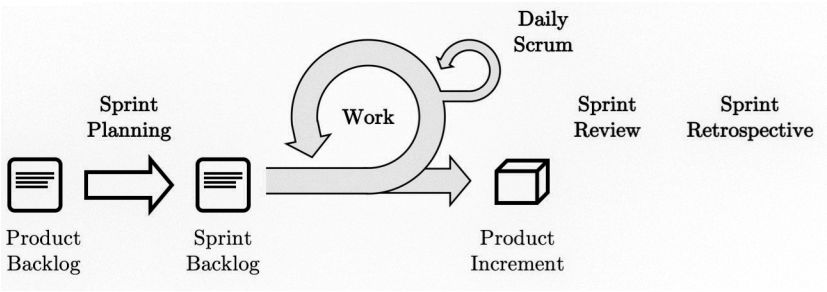


Figure 2.5: Elements of Scrum.

Sprints

With Scrum, software products are developed iteratively. As in Section 2.2, iterative process aim for a potentially deliverable product at the end of each iteration. Iterations are called *sprints* in Scrum. Sprints are relatively short: one month or less.

Scrum specifies three things: roles for team members; planning and review events during a sprint; and artifacts related to product development. The specified events help to keep sprints on track. Figure 2.5 illustrates the four events and the three artifacts specified by Scrum:

Events	Artifacts
Sprint Planning	Product Backlog, Sprint Backlog
Daily Scrum	
Sprint Review	Product Increment
Sprint Retrospective	

A sprint begins with the current *product backlog* of potential work items. Think of the product backlog as a "wish list" of requirements items.

The first event in a sprint is sprint planning, represented by the first arrow from the left. The purpose of *sprint planning* is twofold: (a) to craft a goal for the work to be done during the sprint; and (b) to select work items for implementation during the sprint. The work items are selected from the product backlog. The selected items are called the *sprint backlog*.

The big circular arrow represents the work to develop the sprint backlog. The small circular arrow represents an event called a daily scrum. A *daily scrum* is a brief (15 minute) event to review the work being done by the developers. During the event, developers review what was done the previous day and what is planned for the given day.

The output from the development work is called a *product increment*; it consists of the functionality completed in this sprint. The product increment

is inspected during a *sprint review* with stakeholders. The sprint ends with a *sprint retrospective* to explore process improvements, in readiness for the next sprint.

Scrum: Structure and Flexibility

Scrum is a framework, not a complete process. A framework specifies some aspects of a process and leaves other aspects to be filled in for a given project. Scrum specifies sprints and events and lets developers choose how they implement work items in a sprint backlog.

In other words, Scrum provides a combination of structure and flexibility: sprints, events, roles and artifacts structure team interaction; and developers have the flexibility to choose how they design and build the work items in a sprint backlog. To keep a project on track, development work is reviewed regularly, during daily scrums, sprint reviews, and sprint retrospectives.

About 10% of agile projects use a Scrum/XP hybrid, which integrates team structure from Scrum with development practices from Extreme Programming (XP). XP is discussed in Section 2.4. Individual Scrum practices, such as daily scrums, have been widely adopted by agile projects.

The rest of this section explores Scrum roles, events, and artifacts.

2.3.2 Scrum Roles

Scrum defines three roles: product owner, developer, and scrum master. Together, a product owner, a development team, and a scrum master form a *scrum team*.

Product Owner. The *product owner* is responsible for the content of the product. The owner must be a person; ownership is not to be spread across team members. In all team events, the product owner serves as the voice of the customer, and sets prioritizes for what the team implements.

subsubDevelopers *Developers* have sole responsibility for implementation. The development team comes up with estimates for the time and effort needed to implement a work item.

Scrum Master. The *scrum master* acts as coach, facilitator, and moderator. The scrum master is responsible for arranging all events and for keeping them focused and on time. The scrum master does not tell people how to do their jobs. Instead he or she guides then by highlighting the purpose and ground rules of an event.

The scrum master also takes responsibility for removing any external impediments for the team.

2.3.3 Scrum Events

Scrum defines four kinds of time-boxed events: sprint planning, daily scrums, sprint reviews, and sprint retrospectives. All of these events occur during a sprint; see Fig. 2.5. One reason for Scrum events is to plan , review, and steer development work during a sprint. Another reason for specified or scheduled events is to facilitate communication within the scrum team. The events are time-limited, to keep down the time spent in meetings.

Sprint Planning.

The purpose of *sprint planning* is to set a sprint goal and to select a sprint backlog. The product owner represents customer needs and priorities. The development team provides estimates for what it can accomplish. Sprint planning is time-limited to 8 hours or less for a one-month sprint. The attendees are the entire scrum team.

- *Sprint Goal*. The product owner proposes a goal for the functionality to be implemented during the sprint. The owner leads the entire scrum team in converging on the *sprint goal*. Once set, the sprint goal may not be changed during the sprint.
- *Sprint Backlog*. Given the sprint goal, the development team has sole responsibility for selecting product-backlog items to achieve the goal. The developers are accountable for explaining how completion of the selected items during the sprint will achieve the sprint goal. to achieve the goal. What the development team can accomplish during a sprint can be renegotiated with the product owner.

Daily Scrum. The name Scrum comes from the sport of rugby, where a scrum is called to regroup and restart play. The att During a sprint, a *daily scrum* is a short 15-minute meeting to regroup and replan development work for the day. The attendees are the scrum master and the development team. The scrum master facilitates the meeting. The developers address three questions:

- What did I do yesterday toward the sprint goal?
- What will I do today?
- Are there any impediments to progress?

These questions keep the whole development team informed about the current status and work that remains to be done in the sprint. The scrum master takes responsibility for addressing any external impediments; that is, with impediments that are external to the scrum team.

The 15-minute time limit works for small teams. The process can be scaled to larger teams by grouping the teams into smaller subteams responsible for subsystems. The daily scrum for the larger team is then a *scrum of scrums*,

with representatives from the subteams. The representatives are typically the scrum masters of the subteams.

Sprint Review. A *sprint review* is an at most 4-hour meeting at the end of a one-month sprint. The purpose of the review is to close the current sprint and prepare for the next. The review includes stakeholders and the whole scrum team.

Closing the current sprint includes a discussion of what was accomplished, what went well, and what did not go well during the sprint. The development team describes the implemented functionality and possibly gives a demo of the new functionality.

A sprint review sets the stage for the planning meeting for the next sprint. Preparing for the next sprint is like starting a new project, building on the current working software. The product owner updates the product backlog based on the latest understanding of customer needs, schedule, budget, and progress by the development team. The group revisits the priorities for the project and explores what to do next.

Sprint Retrospective. A *sprint retrospective* is an at most 3-hour meeting for a one-month sprint. In the spirit of continuous improvement, the purpose of the retrospective is for the scrum team to reflect on the current sprint and identify improvements that can be put in place for the next sprint. The improvements may relate to the product under development, the tools used by the team, the workings of the team within the rules of Scrum, or the interactions between team members.

2.3.4 Scrum Artifacts

Scrum specifies three artifacts: product backlog, sprint, backlog, and product increment. The artifacts are visible to all team members to ensure that everyone has access to all available information.

Product Backlog. The *product backlog* is an evolving prioritized list of items to be implemented. It is maintained by the product owner. It represent the requirements for the product, based on stakeholder needs, market conditions, and the functionality implemented during previous sprints. At the end of each sprint, the product backlog may be updated, during the sprint review with stakeholders.

Sprint Backlog. A *sprint backlog* consists of the work items to be implemented during the current sprint. The work items are selected from the product backlog during sprint planning, by the development team.

The sprint goal drives the selection of work items for the sprint backlog. The goal cannot be changed during the sprint. The sprint backlog may, however, be changed by the developers work progresses and new information comes to light.

Changes to the sprint backlog may be made only by the developers, and the intent of the changes is to keep the sprint backlog focused on the sprint goal. Any requirements changes have to wait until the next sprint planning event.

Product Increment. As with any iterative process, the product evolves with every sprint. The deliverable from a sprint is called a *product increment*. The added functionality in the product increment is driven by the sprint goal.

The product increment is reviewed with stakeholders during the sprint review. The intent is to have a potentially releasable product at the end of each sprint.

2.4 XP: Agile Development Practices

- *Highlights*. The premise of XP is "an always-deployable system to which features, chosen by the customer, are added and automatically tested on a fixed heartbeat."[14]
- *Risks Addressed*. Extreme Programming (XP) addresses risks related to requirements and product quality.
- *Caveats*. Potential design risk from deferring design until the last "responsible" moment.

2.4.1 Overview of XP

Extreme Programming (XP) is more about agile values and culture than it is about specific software-development practices. The practices in XP are refinements of best practices that predate XP, in some cases, by decades. What is of interest is how the practices work together. Development is iterative. The outer loop in Fig. 2.6 represents an iteration.. Starting at the top, an iteration proceeds as follows:

- review customer needs and identify what to build in this iteration;
- define tests before coding;
- write just enough code and integrate it into a working system; and then
- clean up the design and code so that it is ready for the next iteration.

Such iterations are common in agile development; they are not limited to XP.

Given the XP focus on the culture of development, this section is organized around the values in the Agile Manifesto. "Individuals and interactions" relates to culture, which is addressed in Section 2.1.2. The practices in this section relate to customer collaboration; responding to change; and working software.

XP was introduced by Kent Beck in the late 1990s.[15]

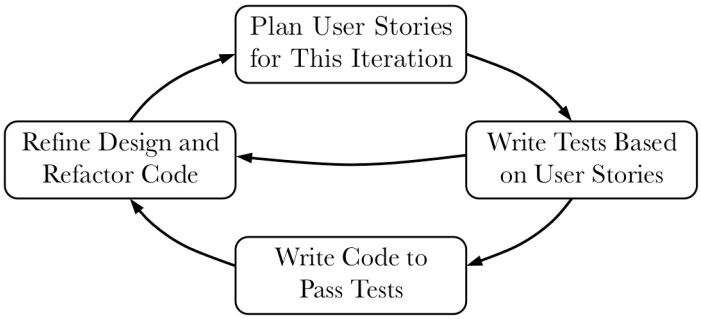


Figure 2.6: Software development using XP.

2.4.2 Customer Collaboration: User Stories

At the start of a project and after each iteration, the development team engages users and other stakeholders to review their needs and wants. This engagement results in a set of stories that have three aspects:

- *Brief Description.* In simple language, user story contains a brief testable description of a user need. A user story is written from the user's perspective.
- *Acceptance Tests.* Each user story is accompanied by one or more acceptance tests to validate the implementation of the story.
- *Estimates.* A rough estimate of the development effort for a user story is essential for cost-benefit tradeoffs across stories. Estimates are provided by the development team.

The acronym *3C* highlights the intent of a user story. 3C stands for Card, Conversation, and Confirmation: fit on an index card; spark a conversation; confirm understanding through an executable acceptance test.[16] A software product of any size may have dozens or perhaps hundreds of user stories.

A template for writing user stories appears in Fig. 2.7.[17] The three main elements of the template are

1. the role or stakeholder who wants the story to work;
2. the need to be met by the implementation; and
3. the business value of the functionality to the stakeholder.

Example 2.4. XP was motivated by a payroll system for Chrysler, so here is a payroll example:

As a	payroll manager
I want to	produce a simple paycheck
so that	the company can pay an employee

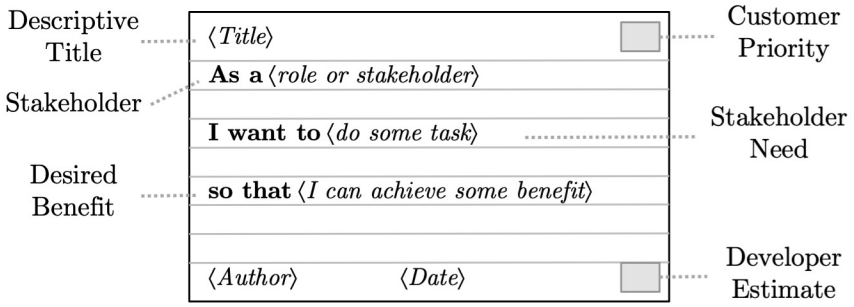


Figure 2.7: A rendering of a user story template created by Connextra in 2001.

Stories get clarified and sharpened during conversations between stakeholders and developers. For example, in this payroll story, what does "simple paycheck" mean? Paychecks can be very complicated, between various forms of compensation and various deductions for taxes, savings, medical insurance, ... Such questions can be addressed during a stakeholder conversation. □

Example 2.5. A ride-sharing app has at least two classes of stakeholders: riders who want rides and drivers who provide rides. Here are some examples of user stories for riders:

- As a rider, I want flexible on-demand transportation so that I can get to my destination whenever I want.
- As a rider, I want a trustworthy driver, so I will feel safe getting into the driver's car.
- As a rider, I want the app to remember my preferred drivers so that I can request their services in the future.

The number of stories will grow as additional needs are identified. □

See Section 4.4 for guidelines for writing user stories.

2.4.3 Responding to Change: Iteration Planning

The collaboration between stakeholders and developers extends beyond reviewing needs and revising the backlog of user stories to be implemented. The collaboration includes prioritizing the backlog and planning what gets implemented in the next iteration.

Agile iterations are *time boxed*, which means that the time interval of an iteration is fixed. A time box constrains what the development team can accomplish during an iteration. Any functionality that does not fit into a time-boxed iteration is either dropped or added to the backlog of functionality to be addressed during a future iteration.

Prioritization of stories for an iteration is done by the stakeholder or the customer representative. Prioritization is based on a rough cost-benefit analysis, where the cost is the developers' estimate of the implementation effort needed for a story and the benefit is the value of the story to the stakeholder.

A team's *velocity* is the sum of the estimates of the stories that a team can implement during an iteration. Velocity determines how many of the highest priority stories are selected for the current iteration. Velocity can also be used to revise estimates as a project proceeds.

Planning of iterations and projects is discussed in Section **??**.

2.4.4 Working Software: Testing and Refactoring

With agile processes, the ideal is to have simple working software all the time, not just at the end of an iteration. Three forms of testing play an essential role in maintaining a state of clean working software: automated tests; continuous integration; and test-driven development.

The combination of constant unobtrusive automated testing and periodic refactoring increases confidence in new code and in changes to existing code. By extension, the combination increases confidence in the overall system under development. The combination therefore promotes the creation of correct code from the outset, instead of untested code that will later be fixed. This shift in approach represents a cultural change in how software is developed, whether or not customer needs are expressed as user stories and whether or not tests are written before code.

Automated Regression Testing

With any change to a system, there is a risk that the change will break something that was working. The risk of breakage can be significantly reduced by ensuring that the system continues to pass all tests that used to work. This process of running all tests is called *regression testing*.

The burden of regression testing can be significantly reduced by automating it. Automated regression testing provides a safety net while developers are making changes. The more complete the regression tests, the greater the safety net provided by automated tests.

Continuous Integration

The goal of working software applies to the overall system, not just to the components. Continuous integration means that overall system integration is done several times a day; say, every couple of hours.

System integration includes a complete set of system tests. If a change causes system tests to fail, then the change is rolled back and reassessed. The complete set of tests must pass before proceeding.

Test-Driven Development

With test-driven development, each new feature, each new piece of functionality, begins with a test of what the feature should do. The test should fail, since the code has not yet been written. Furthermore, the test should fail for the expected reason. If, however, the test passes, then either the feature already exists, or the test is inadequate.

In addition to acceptance tests, developers may write tests that are relevant to the proposed implementation.

Once the tests are written, the idea is to write just enough code so the software passes all tests. Regression testing acts as a safety net during test-driven development, since it runs all tests to verify that the new code did not break some existing feature.

Periodic Refactoring

Refactoring consists of a sequence of correctness-preserving changes to clean up the code. Correctness-preserving means that the external behavior of the system stays the same. Each change is typically small. After each change, all tests are run to verify the external behavior of the system.

Without refactoring, the design and code of the system can drift with the accumulation of incremental changes. The drift can result in a system that is hard to change, which undermines the goal of working software.

ical debt

Pair Programming

Pair programming is perhaps the most controversial aspect of XP.F *Pair programming* is the practice of two developers working together on one task, typically at the same machine. The claimed benefit of pair programming is that it produces better software more effectively. Two people can share ideas, discuss design alternatives, review each others' work, and keep each other on task. As a side benefit, two people know the code, which helps spread knowledge within the team. So, if one person leaves or is not available, there is likely someone else on the team who knows the code.

One of the concerns with pair programming is that having two people working on the same task could potentially double the cost of development. An early study with undergraduate students concluded that pair programming added 15% to the cost, not 100%. Further, the 15% added cost was balanced by 15% fewer defects in the code produced by a pair.[18]

A decade later the controversy remained:

> "We are no longer in the first flush of pair programming, yet the gulf between enthusiasts and critics seems as wide as ever."[19]

2.5 When and How Much to Design?

Successful project managers tailor (customize) their development processes by borrowing best practices, without regard to the sources of the practices. Short iterations, daily standups, testing after every change, and refactoring have crossed the borders of specific development methods. Customer involvement has become part of the culture.

There is one area, however, where what to do is not clear cut. That area is design and architecture. The key questions include: How much to invest in design up front, at the start of a project? How to spread design investment over the life of a project; in short, when to design?

Looking ahead, each project has to come up with its own answers. There is no one-size-fits-all answer. If the stakes are low enough, little or no design may suffice. If the stakes are high, however, significant design investment may be warranted. Product attributes like security, performance, and reliability have to be designed in from the start; they are hard to retrofit. As they say, if you're building a dog house in your back yard, just about any design will do. But, if you're building a skyscraper, ...

The tradeoff is as follows:

- *Effort Risk.* Too much design too early and there may be wasted effort on planning for something that will not be needed. In short, too much design carries effort risk.
- *Architecture Risk.* Too little design too late may result in an unwieldy architecture that does not scale or has to be fixed to accommodate new requirements. In short, too little design can carry architecture risk.

This section addresses the above questions from two perspectives: the cost of making a change now rather than later; and the level of risk associated with the project. The focus of the discussion is on insights that can help projects make informed decisions about their specific situations. But first, let us briefly consider approaches to design and architecture.

2.5.1 Architectural Approaches

When requirements change, the design and architecture of the product must evolve to accommodate the changes. The following are two approaches to evolving or maintaining a design:

- *Flexible Initial Design.* Start with a design that is flexible enough to handle later changes without major rework.
- *Incremental Design.* Start with a minimal plausible design, add to it as needed, and refactor at the end of each iteration to incrementally clean up the design.

Both approaches involve some up-front design. The difference between them is in their point of view. Flexible design invests in anticipation of a later need.

Incremental design defers investment until there is a need. The balance between up-front flexible design and just-in-time incremental design depends on the goals and risks of the project.

Flexible Design

Flexible design was identified as a success factor in a study of 29 iterative projects. Major initial investments in software design and architecture correlated with higher quality. All of the projects used short iterations, comparable to those of agile processes.[20] The short iterations were to cope with uncertain and dynamically changing requirements. (From Section 1.2, uncertain requirements are "known unknowns," whereas dynamically changing requirements are "unknown unknowns.")

Flexibility involves product-related tradeoffs, in addition to the risk of effort wasted on functionality that is never needed. Architectures that are modular are more likely to be flexible than architectures that are optimized for a specific purpose, such as performance. In other words, there may be tradeoffs between flexibility and attributes such as performance and scale.

Incremental Design

The premise of the incremental design approach is that the overall cost can be reduced by doing just-in-time design. Automated regression testing and refactoring are essential to incremental design. Regression testing ensures that design changes are safe, and that the features that used to work continue to work. Refactoring keeps the design clean, which lowers the cost of making a change.

The risk with incremental design is that of running into a dead end that might have been avoided with some preplanning.

Design and Agile Development

In the early days of agile development, some teams deferred design until the last possible moment and ended up with poorly designed brittle systems. Their approach to design is summed up by the acronym *yagni*, which comes from "you aren't going to need it."[21] Yagni came to mean: don't anticipate; don't implement any functionality until you need it.

However, yagni does not have to mean no design up front. On design, Kent Beck's view is as follows:

> "The advice to XP teams is not to minimize design investment over the short run, but to keep the design investment in proportion to the needs of the system so far. The question is not whether or not to design, the question is when to design."[22]

He recommends deferring design until the last "responsible" moment.[23]

What is the last responsible moment?

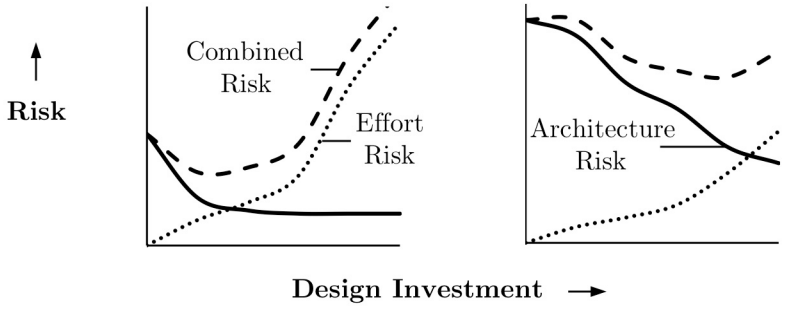


Figure 2.8: Tradeoffs between

2.5.2 A Cost-of-Change Perspective

We might expect the cost of making a change to rise as a project progresses. Some changes are easy, some are hard. On average, the later the fix, the greater the cost. As more decisions are made, more code is written, and the product takes shape, changes become harder and costs go up.

How much does the cost of a change rise over time? Hard data is scarce.

Agile methods embrace changing requirements on the premise that the cost of a change rises slowly. The reasoning is that cost can be kept low incremental design, by spreading planning, analysis, and design across iterations and doing them a little at a time.[24]

If the cost of a change is relatively constant, then design investment can be deferred until it is needed, because the added cost of a design change stays relatively low over the life of a project.

2.5.3 A Risk Perspective

For some guidance on when and how much to design, consider the combination of the architecture and effort risk. As noted above, with low design investment, architecture risk can be high; and, with high design investment, effort risk can be high.

In the two scenarios in Fig. 2.8, design investment increases along the horizontal axis and risk increases along the vertical axis. Architecture risk, depicted by solid lines, falls as design investment increases. Effort risk, depicted by dotted lines, increases with design investment: with no up-front design, effort risk is zero, since there is no investment, so no waste,

The sum of the two risks is depicted by dashed lines. The difference between the two scenarios is in the rates at which these risks fall and rise.

In the first scenario, on the left, low design investment is adequate, since the combined risk quickly reaches a minimum and then starts to rise. Perhaps, the development team is familiar with the application, so architecture risk quickly levels off—the initial architecture may be adequate to the task. Architecture

never goes away entirely, since there is always the possibility of unexpected requirements changes. Meanwhile effort risk rises with further investment. With an adequate architecture, further investment is likely to be unnecessary.

In the second scenario, on the right, higher design investment is recommended, since the combined risk continues to fall with investment. Architecture risk starts out high. Perhaps, the application is a critical one, or has performance and security requirements that cannot be bolted on later. Meanwhile, effort risk is slow to rise. The minimum combined risk tracks design investment.

The scenarios in Fig. 2.8 are hypothetical. A risk-based approach, however, can inform decisions, even with only rough estimates of architecture and effort risk.

2.6 Waterfall Processes

- *Jighlights.* Organize software development as a sequence of phases; e.g., planning, analysis, design, coding, and testing. Each phase completes before the next one begins.
- *Risks Addressed.* Careful up-front planning addresses design risk; see also caveats, below.
- *Caveats.* The biggest risk is requirements risk, since the product is planned and then built according to the the starting requirements. Late-stage testing results in design and quality risks, Since testing comes at the end, defects can lie undetected until the end. , Rework to fix defects can be accompanied by cost and schedule overruns.

2.6.1 Overview of Waterfall Processes

A *waterfall* process is characterized by a linear sequence of activities that are performed one after the other. The name waterfall comes from diagrams like the one in Fig. 2.9, where the sequential flow from one activity to the next looks a little like water streaming over a series of drops in a waterfall[25]

Waterfall processes provide useful lessons in what not to do: they lack customer involvement during development and a working product emerges with a big bang at the end. Customer collaboration has already been stressed in the discussion of iterative and agile processes. This section explores some of the consequences of deferring testing until the end. Problems related to late-stage testing continue to trip software projects; `healthcare.gov` is a recent example. This section concludes with brief historical notes on how a couple of major projects managed the risks with waterfall processes.

Grow Versus Build Analogy

By itself, customer collaboration is not enough to address the risk of changing requirements. Equally important is a working software that they can comment

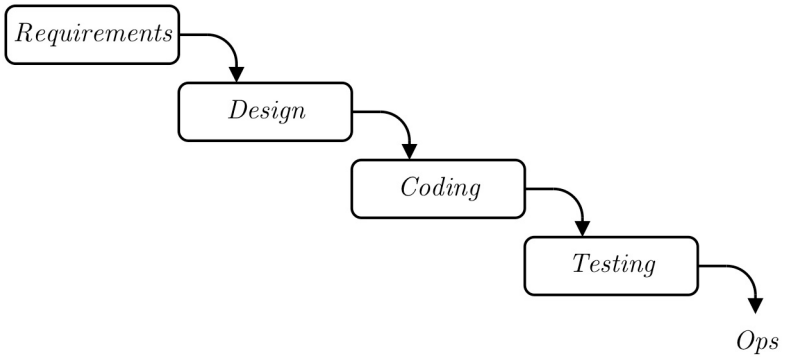


Figure 2.9: A waterfall process. Boxes represent activities.

on.

With waterfall processes, software is built like a physical building is built: neither is ready of use until the end of the project. A building goes through phases: an architect prepares a blueprint; a contractor then builds according to the blueprint; an inspector finally certifies that the completed building is fit for occupancy. With each phase, the building takes shape, but it is not usable until the end.

Informally, an iterative process *grows* a product in functionality. The grow analogy is with living organisms. The capabilities of the organism grow as it evolves.[26]

A growing working system makes it easier for customers to visualize what they will get from a completed system.

2.6.2 The Perils of Big-Bang Integration and Testing

Waterfall processes were supposed to make software development orderly and predictable. The implicit assumption behind them is that everything will go according to plan. In other words, we can specify what a system must do before we design it; that we can fully design it before we code it; that we can code it before we test it; and that there will be no serious issues along the way. This assumption is very risky. c Testing that is deferred until the end of a project is called *big-bang testing*. With it, design and coding issues are discovered late in a project's life. Design issues include performance, security, reliability, and scalability issues.

In practice, late discovery of major issues has resulted in replanning and rework, leading to significant cost and schedule overruns. Late projects can also suffer from quality problems. When projects run late, testing can get squeezed. Inadequate testing can result in the delivery of projects with defects.

There continue to be cases of testing getting squeezed when software projects

run late. The public project in the next example had problems that were due, in part, to big-bang integration and testing of components from multiple contractors.

Example 2.6. The Affordable Care Act has enabled millions of people to get healthcare coverage. The software system to implement the law was the result of a two-year project, with components built by several contractors. The system includes a website, `healthcare.gov`, for people to enroll for health insurance.

The October 1, 2013 launch of the website did not go well. Integration and testing came at the tail end of the rollout of the site, too late to address usability and performance issues.

The chairman of the oversight committee opened a congressional hearing on October 24 with

> "Today the Energy and Commerce Committee continues our ongoing oversight of the healthcare law as we examine the many problems, crashes, glitches, system failures that have defined open enrollment [for health insurance]."

When questioned, one of the contractors admitted that integration testing for the website began two weeks before launch. Another contractor admitted that full end-to-end system testing did not occur until "the couple of days leading up to the launch."

At the time, $118 million had already been spent on the website alone.[27]
□

2.6.3 The Waterfall Cost of Change Curve

As noted in Section 2.5, we expect the cost of a making a change to rise as a project progresses and the product takes shape. For waterfall processes, the cost of a change rises exponentially. There is data to back up the intuition that the later the fix, the greater the cost.

A Proxy for the Cost of a Change

The data that is available is for the cost of fixing a defect. Making a change and fixing a defect are closely related, since both require an understanding of the code for a system. Let us therefore use defect data to support conclusions about the cost of a change.

The curves in Fig. 2.10 show changes to the cost of fixing a severe defect as a waterfall project progresses. The cost is lowest during the requirements and design phases. The solid curve is based on a chart published in 1976 by Barry W. Boehm. The underlying data was from three companies: GTE, IBM, and TRW. Boehm added data from smaller software projects in a 2006 version of the chart.[28]

For large projects, the relative cost of fixing a severe defect rises by a factor of 100 between the initial requirements and design phases and the final phase

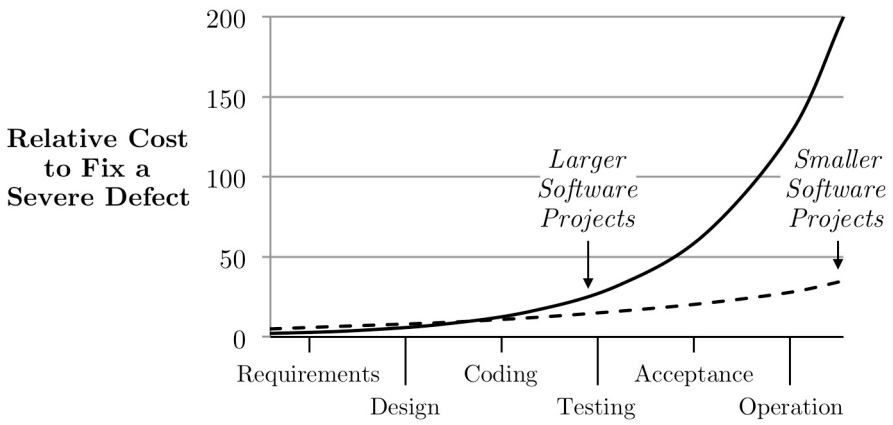


Figure 2.10: For severe defects, the later the fix, the greater the cost. (The original diagram had a log scale for the relative cost).

where the system has been delivered and put into operation. The cost jumps by a factor of 10 between requirements and coding and jumps by another factor of 10 between coding and operation.

The dashed curve for smaller projects is much flatter: the cost of a fix during operation is 7 times the cost of a fix during the requirements phase. For non-severe defects, the ratio may be 2:1, instead of the 100:1 ratio for severe defects.

The following table summarizes the above discussion comparing the cost of late fixes (during operation, after delivery) to the cost of early fixes (during initial requirements and design):

Project Size	Defect Severity	Ratio
Large	Severe	$\sim 100:1$
Small	Severe	$\sim 7:1$
Large	Non-Severe	$\sim 2:1$

(The symbol "$\sim$" stands for "roughly.")

Similar ratios were reported during a 2002 workshop by projects from IBM Rochester, Toshiba's software factory, and others.[29]

2.6.4 Managing the Risks of Waterfall Processes

In the 1970s, waterfall was the dominant process model, to the point where the Space Shuttle project felt the need to justify their use of an iterative rather than a waterfall process:[30]

> "From an idealistic viewpoint, software should be developed from a concise set of requirements that are defined, documented, and

> established before implementation begins. The requirements on the Shuttle program, however, evolved during the software development process."

Thousands of useful software products were built using variants of waterfall processes, but too many projects failed to live up to their promise or failed entirely.[31]

What did successful projects do right; hat is, how did they manage the significant risks? As the SAGE case study illustrates, up-front design can be effective when requirements are stable and the design issues are well understood. The SAGE project's approach to testing is discussed in Section 2.7.

Case Study: SAGE

SAGE (Semi-Automated Ground Environment) was an ambitious distributed system that grew to 24 radar-data collection centers and 3 combat centers spread across the United States. The SAGE development team addressed both requirements and design risks by gaining a deep understanding of the problem to be solved and of possible workable solutions. They did so by building a prototype with all of the functions of air defense. Then, they successfully built SAGE, as described in the following example.

Example 2.7. SAGE development began with an initial prototyping phase to explore the tradeoffs between performance, cost, and risk. Herbert Benington writes,[32]

> "The experimental prototype ... performed all the bare-bones functions of air defense. Twenty people understood in detail the performance of those 35,000 instructions; they knew what each module would do, they understood the interfaces, and they understood the performance requirements."

Having done a realistic prototype, the team was well prepared to build the final system, which was roughly three times the size of the prototype: 100,000 lines. □

2.7 V Processes: Levels of Design and Testing

- *Highlights.* V-processes emphasize testing. They pair specification and testing at multiple levels of granularity, from the whole system down to individual modules (units). The teststing phases check that the corresponding specification was implemented correctly. Levels of testing—unit, functional, integration, system, acceptance—came with V-processes, in the 1950s.
- *Risks Addressed.* Thorough testing addresses the risk of defects due to design and implementation.

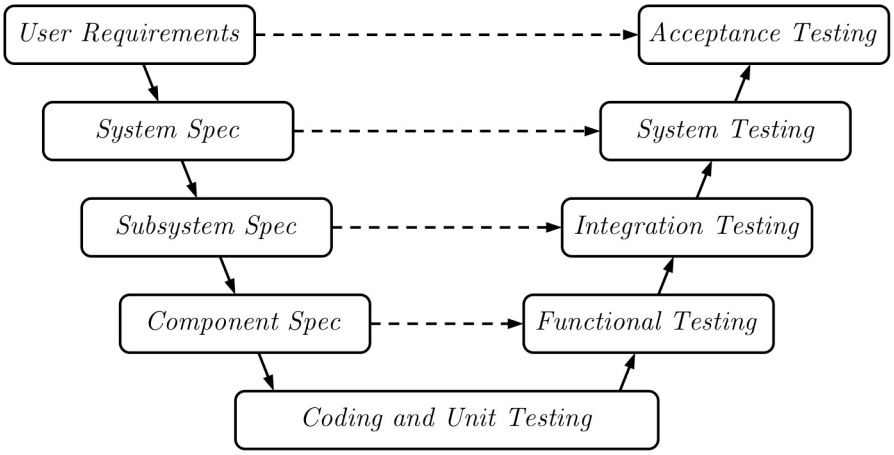


Figure 2.11: A V process with levels of specification and testing.

- *Caveats.* V processes are a variant of waterfall processes, so they remain subject to requirements-related risks. The concepts of unit, functional, integration, system, and acceptance testing come from V processes.

2.7.1 Overview of V Processes

Does testing have to come late in a development process? No! Testing can begin as soon as there is something to test. The earlier a defect is found, the less it costs to fix; see Section 2.6.3. Test planning can begin even earlier—before there is anything to test. In fact, test-driven development is based on the idea of starting with tests and then writing code so it passes the tests.

A *V process* has levels of design and testing: the higher the level, the larger the software under design and test. The number of levels varies from project to project. An example of a V process appears in Fig. 2.11.

Diagrams for V processes resemble the letter V. The design phases are drawn going down and to the right. Coding and any testing that occurs during coding are at the bottom. The testing phases are drawn going up and to the right. The dashed arrows in Fig. 2.11 link a specification phase with its corresponding testing phase. Design and testing are thus paired, like opening and closing parentheses, surrounding a coding phase.

At each level, the result of the design activity is a specification of what to build and test at that level. The testing activity at each level checks that its corresponding specification is implemented correctly.

Test planning can begin in the down part of the V, before coding begins. Test execution must follow coding, so test execution is in the up part of the V.

Including relevant tests with a specification strengthens the specification.

V processes are based on the development process for the SAGE air defense system, which was built in the 1950s.[33] In the SAGE project, test planning was done in parallel with coding.

V processes are essentially waterfall processes. Note that the solid arrows in Fig. 2.11 trace the sequential flow from customer requirements to acceptance testing. A specific V process may have different levels of specification and testing.

2.7.2 Levels of Testing, From Unit to Acceptance

The upper levels of a V process focus on validation; the lower levels focus on verification, as described below. Both validation and verification refer to forms of error checking::[34]

Validation: "Am I building the right product?"
Verification: "Am I building the product right?"

Validation testing checks whether a product has the correct functionality. Verification testing checks for implementation defects. (See Chapter 9 for more on validation and verification.)

The following are typical levels of testing; see Fig. 2.11:

- *Acceptance Testing.* Performed by customers to validate that the system meets their requirements for its use and operation.
- *System Testing.* Done by developers to validate the functionality and performance of the system as a whole.
- *Integration Testing.* Done by developers to verify that the modules worked together.
- *Functional Testing* Black box testing to verify the design of the modules. Testing for functionality also occurs during system testing.
- *Unit Testing.* Primarily white-box testing of modules or units, where *unit* is short for unit of implementation.

For more on these levels of testing, see Chapter 10.

2.8 The Spiral Risk-Reduction Framework

- *Highlights.* In each cycle of a Spiral Framework, the team revisits stakeholders, requirements, solution approaches, and risks, before picking a development process for that cycle. Commitment to the project increases with each cycle
- *Risks Addressed.* The Spiral Framework is used to identify and address risks.
- *Caveats.* The Spiral Framework is often misinterpreted as either an iterative process or as a sequence of waterfall processes

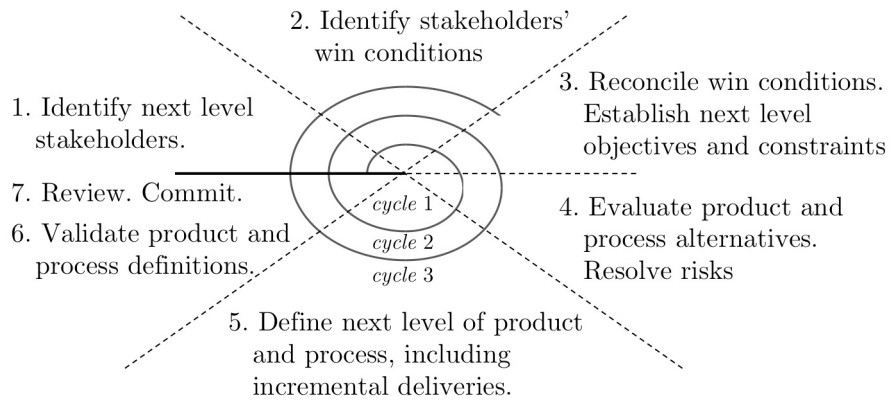


Figure 2.12: Spiral Risk-Reduction Framework.

2.8.1 Overview of the Spiral Framework

The *spiral framework* is a cyclic approach to risk reduction during software development. With each cycle, risk shrinks and the system grows. Each cycle is like a mini-project. Unlike iterations in iterative processes, in each cycle, the team can use a different development process. Barry Boehm introduced the spiral framework in the 1980s.[35]

The framework takes its name from diagrams like Fig. 2.12, where cycles are depicted by the loops of a spiral. The first cycle is at the center. The widening cycles reflect the increasing levels of investment and commitment to the project. The greater the risk, the more careful the investment. The level of effort and investment in a cycle is guided by the risk reduction planned during the cycle. For example, a first-of-its-kind project might have a prototyping cycle to investigate whether a proposed approach will work well. Better to invest a little up front in prototyping than to launch a full implementation, only to have it fail.

Each cycle corresponds to a pass through the numbered actions in the diagram. (Although the actions are numbered, they are expected to be concurrent, where possible.) The pass begins with identifying the next level of stakeholders and proceeds clockwise all the way around to the commitment to proceed with the next cycle.

The upper half of the diagram relates to the customer problem, the bottom half to the evolving solution:

- *Problem Definition.* Identify the stakeholders and their objectives. Reconcile any conflicts and respect the constraints.
- *Solution Progress.* Resolve risks through benchmarking, modeling, and/or

prototyping. Build the next increment for the next level of risk-reduction. Review with stakeholders and get a commitment to proceed.

The framework accommodates any methods or processes for carrying out the actions listed in Fig. 2.12. In fact, the reason for calling it a "framework" rather than a process model is that the framework has been adapted for use with various process models. For example, risk resolution in Action 4 can be through analysis, prototyping, or by building an increment to the system. In Action 5, the process for the next level of system definition and development can be different from the process used during the previous cycle.

For a simple project, some of the actions may be combined and the number of cycles reduced. A small project may have just one cycle. For a complex project, a more formal approach with more cycles might be helpful.

Example 2.8. This example illustrates the flexibility of the spiral risk-reduction framework. It deals with a very large contract for a system to control remotely-piloted vehicles (drones). The contract had several risk-reduction cycles.

The challenge was to improve productivity by a factor of 8, from each drone piloted remotely by 2 people to 4 drones controlled by one person.[36] In other words, improve eightfold from a 1:2 ratio to a 4:1 ratio of drones to pilots.

The project started with four competing teams in the first risk-reduction cycle. Each team was awarded $5M (M is for million). With an additional $5M for evaluation, the initial investment for the four teams was $25M. The review at the end of the first cycle concluded that a 4:1 ratio was not realistic, but that some improvement was possible.

Three competing teams remained in the second risk-reduction cycle. Each was awarded $20M to build a scaled-down system. With an additional $15M for evaluation, the incremental investment for the three teams was $75M. The review at the end of the second cycle concluded that a ratio of 1:1 was possible.

For the final cycle, one team was selected to build a viable system that achieved a ratio of 1:1, resulting in a twofold productivity improvement.

Each of the competing teams could choose its own process for a given cycle. □

2.9 Conclusion

Software development projects can be grouped roughly into iterative and agile on the one hand and waterfall and plan-driven on the other. These groupings represent two distinct development cultures. Agile projects share the values in the Agile Manifesto: respect for individuals and team interactions; customer collaboration; working software; and responsiveness to change. *Plan-driven* projects stress careful up-front planning, design, and documentation, prior to coding and testing. There is little, if any, customer involvement after the initial design.

The Spiral Risk-Reduction Framework (Section 2.8) spans the two groupings. First the Spiral framework is not a development process; it leaves that

choice up to developers. The framework has a sequence of cycles: in each cycle, the team identifies the next level of stakeholders, their requirements, and the current risks to the project. The team then selects a development process for the rest of that cycle.

Contrasting Plan-Driven and Agile Priorities

Plan-driven projects are risky. The biggest risk is that requirements typically change while a project is underway. While many successful products have been built using plan-driven processes, there have been many failures as well.

With their emphasis on customer collaboration and working software, agile processes address the risks related to requirements and product quality. They slice development into short iterations that build increasingly functional working versions of the product. The functionality added in an iteration is guided by customer feedback. To this vision of deployable working versions, the Scrum Framework adds rules for team coordination. The rules define roles for team members; event, in the form of meetings and reviews; and artifacts to be produced. Extreme Programming (XP) complements Scrum, by providing development practices that mesh well with the Scrum Framework.

The contrast between plan-driven and agile processes extends to how they prioritize the scope, time, and cost of a project. These contrasting priorities are illustrated by using variants of the Iron Triangle, introduced in Section 1.5. The triangle in Fig. 2.13(a) is for a traditional plan-driven process that fixes the scope and allows the time and cost to vary. Such a project is run until the full functionality (scope) can be delivered, even if there are schedule and cost overruns. The inverted triangle in Fig. 2.13(b) is for a time-boxed iterative/agile process. With time-boxed iterations, time and cost are fixed; scope varies. Such a project drops lower priority features that cannot be completed within the allotted time.

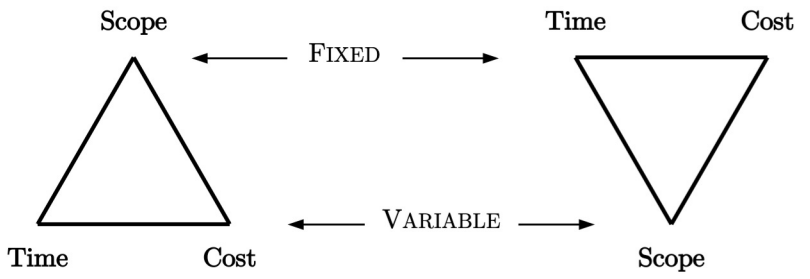
With time-boxing, might some higher priority features be dropped because time runs out? Potentially, yes. However, since agile projects replan for each iteration, presumably, planning will provide early warnings if it appears that higher priority features might not be completed in time. Since agile processes involve customers in iteration planning, the project team can renegotiate the deliverables with customers, if needed. (In practice, plan-driven project deliverables also get renegotiated, if needed.)

An inverted triangle that fixes time and cost, as in Fig. 2.13(b), is called an *Iterative* or *Agile Iron Triangle*.

Tailoring a Process

The progression of topics in this chapter suggests an approach to customizing a software-development process for a project:

- commit to a culture and values for the team. See, for example, the values in the Agile Manifesto (Section 2.1).



(a) Plan-Driven Processes (b) Iterative Processes

Figure 2.13: Project Management (Iron) Triangles for plan-driven and iterative processes. Note that agile processes are iterative, so the inverted triangle applies to them as well.

- Define How the team will work together. Two thirds of agile projects use Scrum for team coordination (Section 2.3). It defines team roles and the relevant artifacts for planning and review events.
- Select development practices. XP emphasizes testing to increase confidence in the code and refactoring at the end of each iteration to clean up the code (Section 2.4).

Here is some more detailed guidance:

- When and how much to design? If the stakes are low, an incremental just-in-time approach to design may suffice. But, if the stakes are high, significant up-front design effort may be warranted. Section 2.5 explores the tradeoffs.
- When and how much to test? V processes introduced testing at various levels of granularity, from units of implementation to modules to entire systems; see Section 2.7. The concepts of unit, functional, integration, system, and acceptance testing come from V processes. For more on testing, see Chapter 10.
- What not to do? See the discussion of waterfall processes in Section 2.6.

Processes can help manage the risks faced by a project, but they do not eliminate it. There are examples of iterative and agile projects that have failed, just as there are examples of waterfall and plan-driven projects that ave succeeded. The Spiral Risk-Reduction Framework highlights the need to assess and manage all of the risks faced by a project.

Exercises for Chapter 2

Exercise 2.1. For each of the following statements, answer whether it is generally true or generally false. Briefly explain your answer.

a) With an iterative process, each iteration builds on the last.

b) Agile processes came after the Agile Manifesto.

c) With agile processes, project timelines can be hard to predict.

d) Agile processes are great for when you are not sure of the details of what to build.

e) Sprint planning is essentially the same as iteration planning.

f) The Scrum Master sets priorities for the product backlog.

g) Pair programming doubles the staff costs of a project.

h) Very few projects have been successful using waterfall processes.

i) With a waterfall process, testing comes very late in the process.

j) Plan-driven processes call for careful up-front planning so there are few errors in the product.

Exercise 2.2. About 10% of Scrum projects report that they use a Scrum/XP hybrid.

a) Briefly describe how you would create a hybrid process from Scrum and XP.

b) Draw a process diagram where nodes represent activities and edges represent transitions between activities.

Exercise 2.3. A usage survey identified the following as the top five agile practices:[37]

1. Daily Standup
2. Retrospectives
3. Iteration Planning
4. Iteration Review
5. Short Iterations

For each practice, describe

- its purpose or role.
- how it is practiced.

Exercise 2.4. In practice, processes do not always run smoothly. For each of the situations below, briefly answer the following questions:

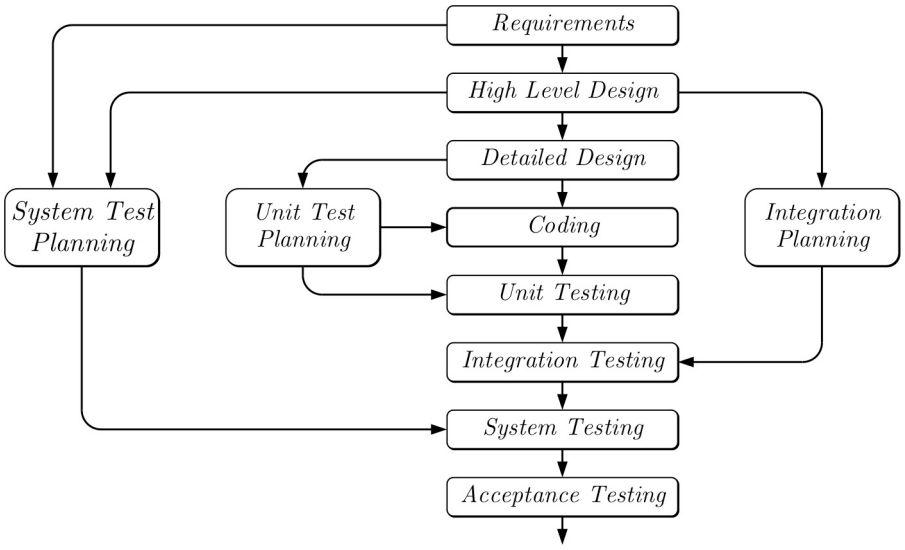


Figure 2.14: A variant of the Infosys development process circa 1996.

- Is there a potential violation of a Scrum rule or an agile value? If so, which one(s)?
- How would you resolve the situation?

The situations are as follows:

a) The product owner challenges developer estimates for the time and effort needed for work items. The product owner is convinced that the developers are inflating their estimates.

b) It appears that, for lack of time, some essential backlog items will have to be dropped entirely from the project.

c) Pressured by a loud stakeholder, a product owner comes to a daily scrum and pushes for the acceleration of work on certain items.

Exercise 2.5. The process in Fig. 2.14 is a variant of Infosys's development process circa 1996.[38] Compare the process in Fig. 2.14 with each of the following. In each case, discuss the similarities (if any) and differences (if any).

a) Waterfall processes.

b) V processes.

c) Iterative processes.

Exercise 2.6. Briefly discuss how you would decide how much to invest in up-front design, for each of the following kinds of software-development projects:

a) Create a website.

b) Develop software for a medical device.

c) Add a feature that maintains a list of the ten most requested songs. The input to the feature is a stream of customer requests for songs.

d) Build a billing system for a global music-streaming service with millions of customers.

Exercise 2.7. The two dimensions in Fig. **??** are Technology and Market. The scale along these dimensions is *New*, for new to the world, and *Stable*, for well known to the world. There can be surprises with either new technology or new markets, and there are no surprises left with either stable technology or stable markets.

a) For each of the quadrants in Fig. **??**, briefly discuss the potential for the following risks:

- Requirements risk.
- Product Design risk.
- Quality risk.

b) How would you address the above risks if you are using

- an agile process?
- a plan-driven process?

Chapter 3

Requirements

> The "most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. For the truth is clients do not know what they want."
>
> — *Fred Brooks, in an influential essay on the inherent versus the incidental impediments to progress in software production.*[1]

3.1 Introduction

All software projects face the question of what to build. What to build emerges from conversations with users on a range of topics: the motivation for the project, including their business goals for the software product; what they need and want to accomplish the goals; possible solution approaches; rough estimates by developers of the time and effort for delivering a product; and how to validate that the developers have built the right product. These topics deal with user priorities and requirements and have little to do with how the developers build the desired product.

With agile methods, user conversations are integrated into product development. Each iteration revisits and updates user requirements. The updated requirements guide the future dire direction of the growing working software product. It is not too much of a stretch to say that agile software development *is* requirements development. In short, the emphasis is on producing working software.

With plan-driven methods, requirements development is an up-front planning phase. It is a significant enough part of planning that requirements development has been called requirements engineering. . The purpose of plan-driven

requirements development is to produce a precise complete specification of what to build. In short, the emphasis is on producing a specification document..

As noted above, the user-focused aspects of requirements development deal with user goals and wants, which have little to do with the software development process. In fact, agile and plan-driven methods share many of the same techniques for identifying and refining user requirements. Roughly speaking, Chapter 4 deals with identification and Chapter 6 deals with refinement of requirements.

This chapter provides a context for discussing requirements. Section 3.3 provides an agile perspective and Section 3.4 provides a plan-driven perspective.

It

3.2 An Overview of Requirements Development

The many uses of the term requirement are reflected in the following definition:

> A *requirement* is a description of a business or user goal, a product behavior, a product attribute, or a product or process constraint.[2]

As requirements get more detailed and more specific, they blur into specifications:

> A *specification* is a complete, precise, verifiable description of a product or of an aspect of a product.

This section introduces requirements and the activities that produce them. Specifically, we consider the activities and artifacts in the process in Fig. 3.1. The boxes in the figure represent activities; the edges are labeled with artifacts. A variant of the process skips the specification activity: analysis transitions directly to validation.

Requirements development is an iterative process that involves users and developers in the following four activities (see Fig. 3.1):

1. *Elicitation.* Identify the business goals that motivated the project and the user needs that must be served by the product.

2. *Analysis.* Refine, reconcile, and prioritize the elicited information, so a product can be outlined.

3. *Specification.* (Agile projects skip this activity.) Produce a detailed unambiguous specification for the proposed product. The documented specification serves as a contract between users and developers.

4. *Validation.* Confirm with users that the right product is being built.

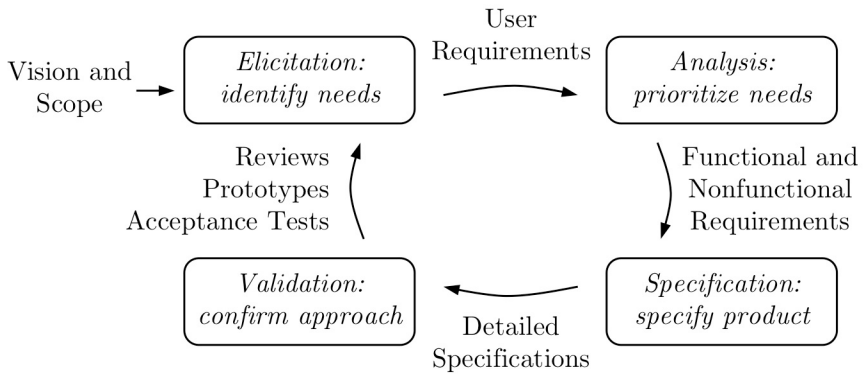


Figure 3.1: High-level view of requirements development activities and artifacts. The boxes represent activities. In practice, the activities may blur into each other; the specification activity may be skipped entirely.

3.2.1 Requirements Activities

The boundaries between the activities in Fig. 3.1 are fluid. A single conversation could uncover a need and prompt a discussion about its priority. The same conversation could then turn to possible acceptance tests for the need.

Thus, any or all of the above requirements activities in Fig. **??** can be part of a given conversation between users and developers.

Each requirements activity has its own bag of tricks and techniques, so the activities have been teased apart into individual boxes in Fig. 3.1. For a small project, some of the activities may be combined or skipped. For a large project, each of the above activities can be a process in its own right.

Elicitation

The purpose of elicitation is to (a) identify stakeholders and (b) their goals, needs, and wants. Here are some questions for the developers to explore with users:

- Who are the primary stakeholders?
- Who are the other stakeholders?
- What do the stakeholders want?
- How do they hope to benefit (i.e., further their business goals)?
- What are the stakeholders' desired overall experience with the product?

A key challenge during elicitation is that stakeholders may be unable to adequately communicate their needs and wants. Some user needs can be readily

identified through interviews and surveys; some can be inferred from what users actually do; and some may be outside their awareness. Chapter 4 explores the nature of user needs and how to access them.

Analysis

The purpose of analysis is to define the desired behavior, functionality, and attributes of a product, based on the information identified during elicitation. The elicited user needs and wants may be incomplete, ambiguous, and/or conflicting, so they have to be refined, reconciled and prioritized. Here are some questions to be addressed during analysis by users and developers, working togehter:

- How important is a given stakeholder want?
- What is a rough developer estimate of the cost of a given want?
- Are there any conflicts between elicited wants?
- Based on their desired overall experiences, are there any additional needs?
- Will the wants provide the desired benefits?
-
- Based on a cost-benefit tradeoff, what is a prioritized list of stakeholder wants?

Analysis and elicitation are overlapping activities. As needs are refined during analysis, further needs may be elicited. Scenarios can help uncover missing information: gaps in the scenarios can point to additional functionality that the product must support.

Conflicts arise when different stakeholders have different priorities; e.g., the user desire for easy access to a system may conflict with the IT staff wanting strictly controlled access due to security concerns. Developers can facilitate, but ultimately it is up to customers and users to reconcile and prioritize their wants.

The output of elicitation and analysis is a prioritized list of the product functionality and attributes to be built by the developers. See also the discussion below of functional and nonfunctional requirements. Nonfunctional requirements are also called quality attributes.

Chapter 6 deals with techniques for requirements analysis.

Specification

Plan-driven processes typically produce detailed documents called software requirements specifications that can be used for contractual discussions between customers and developers. The documents are meant to be specific, precise, and complete enough that (a) users will know what they will get and they will get, and (b) developers will know exactly what to build. See Section 3.4.

Validation

Validation with users is a key part of requirements development. It includes the definition of acceptance tests. With agile methods, working software is validated through user feedback at the end of each iteration. With plan-driven methods, the development team validates plans for the proposed product and confirms that the right product is being built. Plans for a product include a high-level system architecture and a requirements specification document. Reviews and prototyping are techniques for validation.

See Chapters 7-8 for software architecture and Chapter 9 for architecture reviews.

3.2.2 Kinds of Requirements

The labels along the edges in Fig. 3.1 represent the kinds of requirements that arise during requirements development—the edge from *Validation* to *Elicitation* is an exception. The level of documentation varies from project to project. Agile methods work primarily with prioritized user requirements, in the form of user stories. Plan-driven methods typically have detailed documentation, including software -requirements specifications.

Business Requirements

The requirements process begins with business goals, shown at the top left in Fig. 3.1.. A *business goal*, sometimes called a *business requirement*, can be anything the customer organization wants to accomplish with the proposed product. For instance, an online retailer may set the goal of fulfilling an order within 24 hours. Business goals often relate to increasing revenues, improving productivity, and doing a better job of serving the business's customers.

User Requirements

A *user requirement* for a product is a description of a stakeholder need or want identified during elicitation. Techniques for recording user requirements include features, user stories, and scenarios. The following feature describes a notification system's response to an event:

> **When** the flight schedule changes, the system **shall** send a notification by text or email, based on the traveler's contact preferences.

Features are written in a natural language like English. The above feature includes syntactic cues: the keywords **when** and **shall**. See Section 3.4 for some suggested syntax for features.

User stories are written from the user's perspective. They provide some context to a request for a feature: they add the requester's role and expected benefit from the feature. Developers might find the added context helpful during implementation. The following user story follows a template from Section 2.4:

As a traveler,
I want to be notified of changes to the flight schedule,
so that I can get to the flight at the right time.

Scenarios describe a user's desired overall experience with using the product to accomplish a goal. Thus, scenarios deal with a product's external behavior, rather than a specific feature. The details of scenarios can vary widely. The scenarios in Section 4.5 seek to bring the user to life as a person, so that developers have a feel for who their product will serve. Meanwhile, use cases focus on user-system interactions. Use cases are typically used for functional requirements, which are discussed below.

i Chapter 4 deals with user requirements, Chapter ch-usecases with use cases.

Product and Process Requirements

A *product requirement* influences or constrains the product's functionality or attributes. The development team defines product requirements by analyzing user requirements. Analysis is represented by the top-right box in Fig. 3.1.

A *process requirement* constrains or influences how a product is developed; that is, it influences the development process.

For example, projects that build safety-critical software typically face process constraints. The US Federal Aviation Administration (FAA) imposes testing constraints on airborne software systems; see MC/DC testing in Section 10.3. For decades, US defense contracts promoted the use of waterfall processes through standards such as Military Standard MIL-STD-1521B, dated June 4, 1985.

Functional and Nonfunctional Requirements

Product requirements can be grouped into two categories: functional and nonfunctional. A *functional requirement* relates to the job to be done. If the product can be described in terms of inputs and outputs, its functional requirements define the mapping from inputs to outputs. Similarly, if a product can be described in terms of stimuli and responses, its functional requirements define the stimulus-response behavior of the product.

A *nonfunctional requirements* relates to how well the job is done. Product attributes such as performance and reliability are covered by nonfunctional requirements.

Is there a better name than "nonfunctional requirement"? The terms *quality requirement* or *quality attribute* are sometimes used instead. We can then talk about the functional requirements and quality attributes of a product.

Product Specifications

Earlier, a specification was definedas a complete, precise, verifiable description of a product or of an aspect of a product. Plan-driven teams maintain detailed

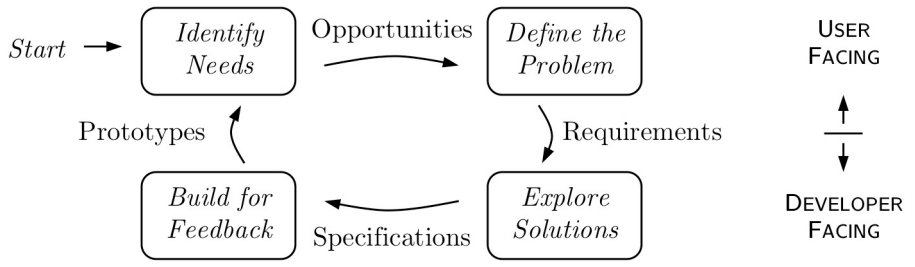


Figure 3.2: An agile process for identifying and addressing customer needs.

product specifications, called *Software Requirements Specifications (SRS)*. An SRS ,might be used by developers to build a product, by testers to write system tests, and by a project manager to estimate the cost and schedule for a project.

Agile teams avoid formal product specifications; they rely on close customer interaction to keep a software development project on track.

3.3 An Agile Case Study

With agile methods, requirements are revisited with every iteration. The descriptions of Scrum and XP in Chapter 2 begin with the selection of work items for implementation during an iteration. Both Scrum and XP value customer collaboration, but the descriptions in Chapter 2 do not spell out how user needs are identified or how product backlog items are created.

By contrast, the agile process in Fig. 3.2 emphasizes requirements development and provides little detail about implementation. The process is adapted from Scenario-Focused Engineering, which has been used within Microsoft.[3]

The activities in the upper half of of Fig. 3.2 are requirements related. They deal with what a product will do for users, independent of its implementation. In other words, they deal with the user problem to be addressed by the project. The activities in the lower half of the figure are implementation related. They deal with how the developers design and build a product that addresses/solves the user problem.

In short, the upper half of the figure is about the problem domain and the lower half is about the solution domain. It is important to keep problem-domain discussions implementation-free, using language that users can relate to.

The Problem Domain

Figure 3.3 elaborate on the activities in Fig. 3.2. Note the similarity between the bullet items in the upper half of Fig. 3.3 and the discussion of elicitation and analysis in Section 3.2.

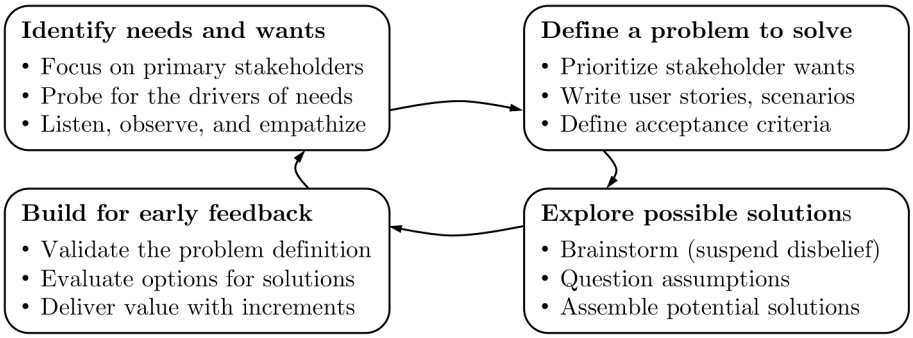


Figure 3.3: The above activities correspond to the activities in Fig. 3.2.

Starting at the top left in Fig. **??**, the first activity is about stakeholder needs and goals, not about product features and capabilities. Developers work with stakeholders to elicit their needs, especially their unmet needs. The output of this activity is a set of user goals and stories about what users want to accomplish. Different stakeholders can have different goals, so the initial set of stories could pull the project in different directions. The stories could even compete or conflict with each other. In the figure, these stories are represented by the arrow marked Opportunities.

Moving right, the next activity is to analyze the opportunities and *frame* or define a problem to be solved by a product. Collectively, developers and users sift through and prioritize the opportunities. This activity is where competing stakeholder goals are reconciled. During problem definition, users and developers converge on the specific problem to be addressed.

Example 3.1. For the distinction between identifying needs and framing the problem, consider the following:

Customer Need:	Listen to music
Problem Definition 1:	Offer songs for purchase and download
Problem Definition 2:	Offer a free streaming service with ads
Problem Definition 3:	Offer paid subscriptions without ads

□

In general, there are multiple ways of addressing a stakeholder need. It may take several discussions to narrow down the possibilities and converge on a solution approach. For example, the developers might produce alternative paper designs or a quick prototype before launching into building a product.

The Solution Domain

The lower half of Fig. 3.3 represents implementation-related activities. The development team brainstorms about possible designs and then builds something to get user feedback. As the iterations continue, the solution' is refined into a deliverable product.

3.4 Plan-Driven Specification and Validation

While plan-driven requirements development follows the outline in Fig. 3.1, the purpose of the latter two phases is to produce planning documents, not working software. This section includes a standard template for writing Software Requirements Specification (SRS) document. It also includes some jelpful rules for writing individual features or requirements. The section concludes with brief notes on prototyping and formal methods, which have been used for both requirements analysis and validation.

Form of a Feature

Plan-driven requirements are written in English or some other natural language. The intent is to make the written requirements accessible to both users and developers. Free-form English text can be imprecise and ambiguous, so written requirements follow some style guidelines to keep them focused. A feature about a system property might be written as follows:

> The system shall have a certain property.

A feature about a system response to an event might be written as follows:

> When event x occurs the system shall respond by doing y.

The rules in Fig. 3.4 are used in the next example to write features in pseudo-English. Features will be written using English phrases that are held together by keywords. The keywords mark elements of a feature, such as the condition under which the feature applies.

Example 3.2. The rules in Fig. 3.4 are for writing requirements in pseudo-English. A study at Rolls-Royce Control Systems found that the rules handled a wide range of requirements. The angle brackets, $\langle$ and $\rangle$, enclose placeholders for text.[4]

This example illustrates the syntactic rules by using them for a notification system for an airline. The first rule describes what the system "shall" do or a property the system must possess:

> The system **shall** support notifications by text message, email, and phone.

System responses may be triggered by events:

1. The ⟨*system*⟩ **shall** ⟨*response*⟩
2. **If** ⟨*preconditions*⟩ **then** the ⟨*system*⟩ **shall** ⟨*response*⟩
3. **When** ⟨*trigger*⟩ the ⟨*system*⟩ **shall** ⟨*response*⟩
4. **If** ⟨*preconditions*⟩ **when** ⟨*trigger*⟩ the ⟨*system*⟩ **shall** ⟨*response*⟩
5. **While** ⟨*state*⟩ the ⟨*system*⟩ **shall** ⟨*response*⟩
6. **Where** ⟨*feature*⟩ is enabled the ⟨*system*⟩ **shall** ⟨*response*⟩

Figure 3.4: Structured English constructions for plan-driven requirements.

> **When** a flight's schedule changes, the system **shall** send a notification by text or email, based on the traveler's contact preferences.

The next requirement illustrates the use of state information. In this case, the traveler can be in one of two states: "in transit" between source and destination with the airline; or "not traveling" with the airline.

> **While** the traveler is in transit, the system **shall** send notifications about gate-change information by text or email, based on the traveler's contact preferences.

The syntactic rules in Fig. **??** are meant to be illustrative; they are not offered as a complete set. Note also that a realistic notification system might have separate sets of requirements for (a) a subsystem that decides what and when to send as a notification and (b) another subsystem to handle modes of contact and traveler preferences about the mode of contact. □

SRS: Software Requirements Specifications

An SRS document plays a central role during plan-driven development. It provides a basis for agreement between customers, designers, implementers, and testers. IEEE Standard 830-1998 (reaffirmed in 2009) provides recommendations for writing an SRS. "There is no one optimal organization for all systems." A sample template appears in Fig. 3.5. According to the standard, a good SRS has the following characteristics:[5]

- *Correct.* The SRS must truly reflect user needs and must agree with other relevant documentation about the project.
- *Unambiguous.* Natural language descriptions can be ambiguous, so care is needed to ensure that the descriptions in the SRS permit only one interpretation. See also the suggested syntax for writing features in Fig. 3.4.

1. Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, Acronyms, and Abbreviations
- 1.4 References
- 1.5 Overview

2. Overall description
- 2.1 Product Perspective
- 2.2 Product Functions
- 2.3 User Characteristics
- 2.4 Constraints
- 2.5 Assumptions and Dependencies

3. Specific requirements
- 3.1 External Interface Requirements
  - 3.1.1 User Interfaces
  - 3.1.2 Hardware Interfaces
  - 3.1.3 Software Interfaces
  - 3.1.4 Communication Interfaces
- 3.2 Functional Requirements
  - 3.2.1 User Class 1
    - 3.2.1.1 Functional Requirement 1.1
    - ...
    - 3.2.1.n Functional Requirement 1.n
  - ...
  - 3.2.m User Class m
- Performance Requirements
- Design Constraints
- Software System Attributes
- Other Requirements

Figure 3.5: An outline of a Software Requirements Specification *SRS), from IEEE Standard 830-1998.

- *Complete*. It must cover all requirements and define the product behavior for both valid and invalid data and events.
- *Consistent*. The individual requirements within an SRS must agree, and occurrences of the same behavior must be described using the same terminology.
- *Ranked*. The description must include the priorities of the different requirements.
- *Verifiable*. All requirements must be testable.
- *Modifiable*. The requirements in the SRS can be easily modified without violating the above properties, such as consistency and completeness.
- *Traceable*. Every requirement can be traced or connected with a user requirement (*backward traceability*). Furthermore, the requirement has a reference to any documents that depend upon it (*forward traceability*).
- *Usable for Maintenance*. The operations and maintenance staff are typically different from the development team. The SRS must meet the needs of the operations and maintenance staff.

Documents created during requirements development need to be managed and maintained to keep up with requirements changes. Keeping documents current must be balanced, however, with actually producing working software.

Prototyping for Requirements Analysis and Validation

Prototyping has long been used to refine requirements and explore possible solutions. Prototyping can also reveal gaps, inconsistencies, and ambiguities in a set of requirementns. In effect, in the process of building a prototype, the developers uncover bugs in the requirements.

Example 3.3. From Example 2.7, the development of the SAGE air-defense system began with an "experimental prototype that performed all of the functions of air defense." The team used the prototype to understand all of the functional requirements and performance requirements, before building an ambitious distributed air-defense system. □

Formal Models

During either analysis or validation, a formal model can be used to debug a set of requirements. A formal model focuses on some aspect of a proposed system. Using a map analogy, a formal model is like a map, whereas the system being modeled is like the territory shown by the map. A map shows some aspect of the territory; e.g., one map provides a street view; another map provides a satellite view.

Use cases model user-system interactions in pursuit of a user goal; see Chapter 5 for use cases. Class and package diagrams model system design and architecture; see Section 7.4. State machines model behavior; state machines are outside the scope of this book.

Logic model checking is a powerful technique that is well suited to debugging concurrent systems, which are notoriously difficult to debug. Such models are executable. Tools can not only pinpoint issues like deadlocks and race conditions, they can provide the sequence of events that lead to an issue. Model checking was used for the flight-control software of the mission that landed the rover Curiosity on Mars. The flight software was highly parallel, with 120 tasks. Issues revealed by model checking prompted a redesign.[6]

Example 3.4. This example illustrates the use of a formal model to debug requirements.

Rising to the challenge of proving the usefulness of formal models, a modeling expert set out to lead a team of 6 developers in building a real product. Call this the *research* team. Meanwhile, a team of 50 developers would use traditional plan-driven methods to build a corresponding product. Call this the *business unit (BU)* team. Both teams would start with the same external requirements for a communications protocol. The developers on both teams were experienced at building such products. The teams were not to communicate with each other about their projects. Based on the BU team's experience with similar products, the experiment with the two teams was to last roughly a year. During this period, the teams were not to communicate with each other about their projects.

The research team began by building a model using a notation called SDL. They soon encountered issues with the external requirements. Some were inconsistencies that were readily resolved; others ran deeper. Over the first few weeks of the project, they fixed multiple issues, until they had a clean model and a set of refined updated requirements. Meanwhile, the BU team followed their traditional process of creating a specification based on the original external requirements—they fixed some of the surface issues in the requirements, but the deeper issues crept into their specification for the product. They then began building software based on those specifications.

While the teams were working independently, management was getting reports from both teams. As the weeks went by, management became alarmed that the research team had information that could help the BU team deliver a better product ahead of schedule. The information was being withheld from the BU team by the rules of the experiment.

About a third of the way into the project, management called off the experiment for business reasons. For the good of the business, they wanted the research team to share their findings with the BU team. While the experiment was halted, it does illustrate the benefits of formal models. □

During requirements validation, a model can warn about potential problems with the proposed system. The idea is as follows: if the model has a problem, then the system is likely to have the problem too. The model is not the full system, so a clean model does not guarantee a clean system.)

For example, suppose we want to check whether some concurrent processes can deadlock. Consider a model that focuses on the messages and coordination between processes; it abstracts away the other aspects of the system. If the processes in the model can deadlock, then it is highly likely that the system will deadlock as well. (We cannot be certain that the system will deadlock because the model is an abstraction, not the full system. The full system have a conditional that avoids the deadlock.)

Formal models can typically be checked by tools. For example, a logic model checking tool can be used to find deadlocks. When such a tool reports a deadlock, it also provides a trace of how the processes in the model get to a deadlock state. The trace can be used to debug the system that is being modeled.

3.5 Conclusion

Exercises for Chapter 3

Exercise 3.1. The two diagrams in Fig. 3.6 are for agile software-development processes. The diagram on the left highlights requirements-related activities. Assume that the output of "Identify Neds" is a set of user stories. The diagram on the right highlights implementation-related activities.

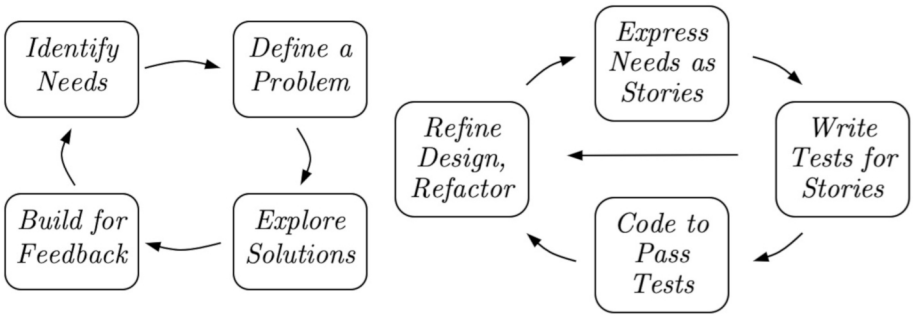


Figure 3.6: Two agiledevelopment process.

Draw a diagram for a unified process that combines the two agile process. Your diagram must highlight both requirements-related and implementation-related activities. Avoid duplication of activities in the unified process.

Exercise 3.2. Consider the agile cycle in Fig. 3.2 involves four main activities:

1. Identify Needs
2. Define the Problem
3. Explore Solutions
4. Build for Feedback

This process can be applied to any product, not just software products. Consider its application to the development of the Sony Walkman cassette-tape player, the first personal audio player. MP3 players and Apple iPods followed in its footsteps. When the Walkman was announced, critics lampooned the playback-only device, without the ability to record audio. The Walkman went on to sell 50 million units w over ten years.

The Walkman story begins with Masaru Ibuka, a co-founder of Sony. Ibuka enjoyed listening to opera on long trans-Pacific flights. He used an existing Sony product that had excellent sound, but was too heavy and bulky for regular use on airplanes—it was also expensive. When asked, Sony engineers quickly built a a light-weight prototype for Ibuka's personal use, by adapting an expensive product called the Pressman, which was marketed only to the press. The engineers dropped the recording capability and the built-in speaker from the Pressman, and added stereo sound. Ibuka was delighted with the prototype.

Sony's chairman then challenged the engineers to create an affordable consumer version with the same sound quality as the prototype. The final product was a sub-$200 small lightweight portable cassette-tape player.[7]

a) In the above account, who was the primary customer? what were the customer needs? What was the desired end-to-end user experience?

b) What was the problem definition? What requirements did the engineers build to?

c) What did the engineers build for feedback?

Chapter 4

User Requirements

> "there is one main obstacle to communication: people's tendency to *evaluate*. Fortunately, I've also discovered that if people can learn to *listen* with understanding, they can ... greatly improve their communication with others."
>
> — *Carl Rogers, in a classic paper on the barriers and gateways to communication.*[1]

4.1 Introduction

In deciding what to build, the first step is to identify the needs of the various stakeholders. The raw unprioritized collection of needs (and wants) is called the set of *user requirements* for a project. With respect to the overview of requirements development in Chapter 3, user requirements are produced during elicitation and refined during analysis; see Fig. 4.1.

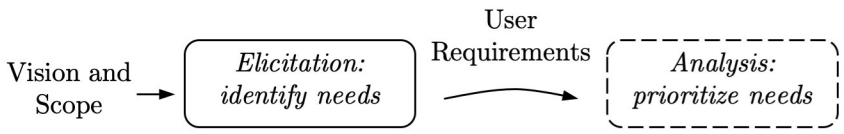


Figure 4.1: User requirements are produced during elicitation.

This chapter deals with elicitation; that is, with how to identify user needs and how to describe them as written requirements. Sections 4.4 and 4.5 provide guidelines for writing user requirements in two distinctly different ways: as user stories and as usage scenarios. A user story describes an individual task, whereas a usage scenario describes an overall user experience.

Stories and scenarios are written from a user perspective. Features, Section 3.4, are written from a system perspective: features describe specific functionality that the system must provide. r See also Chapter 5 for use cases, which describe user-system interactions.

The focus of this chapter is on the first of the following three issues with requirements:

- *User Uncertainty.* There is a grain of truth to the saying that users don't know what they want. The more nuanced point in Section 4.2 is that we need to attend not only to what users say, but also to what they do and feel.
- *Multiple Stakeholders.* Different stakeholders can have different needs, resulting in potential inconsistencies in user requirements. Chapter 6 has techniques for analyzing and prioritizing user requirements.
- *Conditions Change.* User goals and needs can change due to unforeseen factors, such as changes in business conditions or the introduction of new products from competitors. Iterative and agile processes deal with requirements changes by incorporating frequent customer feedback; see Chapter 2.

Case Study: Requirements Changes

In the case study in the Example 4.1, the development team cited requirements changes as its biggest challenge. The project suffered from the three main issues with requirements: there was uncertainty about user requirements; there were multiple stakeholders; and conditions changed in the years between the start and the cancellation of the project. The project was troubled from the start. It began as a technology initiative, rather than as something users wanted. It started before requirements were fully developed. It took too long to get to working software, long enough that users found other tools to get work done.

Example 4.1. In May 2013, the British Broadcasting Corporation (BBC) canceled its Digital Media Initiative, writing off an investment of £98.4 million over six years.[2] What went wrong? The chief technology officer, who was subsequently dismissed, was reported to have said,

> "Throughout the project, the team informed me that the biggest single challenge facing the project was the changes to requirements requested by the business."

The original intent of the Digital Media Initiative was to create a fully integrated archiving and production system for audio and video content. The

archive is extensive: the BBC began radio broadcasts in the 1920s and television broadcasts in the 1930s. The project would allow BBC staff and partners to access the archive and create, manage, and share content from their desktops.

Inadequate User Requirements. The project was initiated by the Director of BBC Technology to standardize television production across the BBC. Standardization implied a significant cultural shift in user behavior: production teams in some of the main BBC divisions would have to give up their existing tools and business practices in favor of the new integrated system.

Unfortunately, the project began before detailed user requirements were fully established. The initial set of user requirements focused on the technology, not on how receptive users would be to the technology.

External Factors. The project got off to a bad start. Funding was approved in January 2008 and a vendor was selected in February, but the contract to build a system was terminated by mutual agreement in July 2009. A year and a half after it started, the project was back to square one, with nothing to show.

The BBC then decided in September 2009 to bring the project in-house. They gave the IT organization the responsibility for building and delivering the system.

The project was never completed. An inquiry by the UK National Audit Office noted that repeated delays and technical difficulties contributed to users losing confidence in the project. Users relied on alternative tools. According to news reports, the chief technology officer cited the following example of changing requirements:

> Users wanted a function to produce a 'rough cut' of video output, which was subsequently developed, only to be told that those users now wanted to use an off-the-shelf product from Adobe instead. Once the IT team had integrated that Adobe product into the DMI software, users then said they wanted to use a different Adobe product altogether." □

The moral of the BBC story: users come first, then the technology. Too often, engineers learn this lesson the hard way.

4.2 Interacting With Users

Why is it that what users say can be very different from what they do? From example 1.4, Netflix customers ask for maximum choice and comprehensive search, but what actually works is a few compelling choices, simply presented. The problem is not new. Faced with a string of product failures in the late 1990s, Scott Cook, the founder of Intuit, resolved that "for future new product development, Intuit should rely on customer actions, not words."[3]

The main message of this section is that the development team will benefit from attending not only to what users say, but what they do and how they

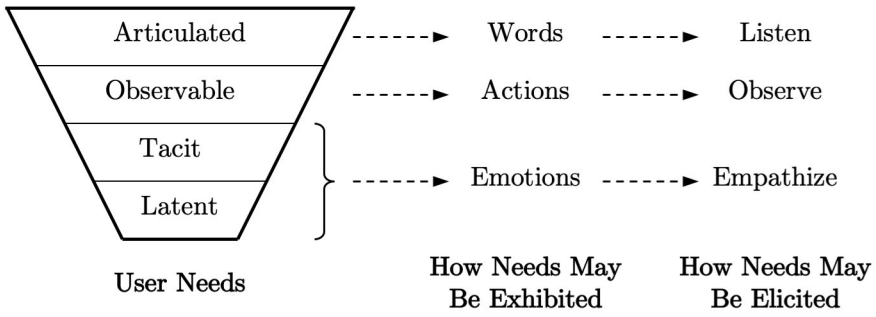


Figure 4.2: A classification of user needs. The dashed arrows represent noisy communication channels.

feel. For example, what are their pain points? In other words, what are the user frustrations that the product should address? By attending to all modes of communication—words, actions, and feelings—developers can build a close working relationship with users.

4.2.1 A Classification of Needs

Words and actions access or provide glimpses of different kinds of needs in the following classification (see Fig. 4.2):[4]

- *Articulated* needs relate to what people say and think. They are what people want us to hear. By listening, we can access and elicit articulated needs.
- *Observable* needs relate to what people do and use. They are made visible by actions and behaviors. Through direct observations and usage data, we can access observable needs.
- *Tacit* needs are conscious needs that people cannot readily express. They relate to what people feel and believe. By empathizing and trading places—walking in someone else's shoes—we can access tacit needs.
- *Latent* needs are not conscious or are not recognized by people. Through empathy and intuition we might make hypotheses about latent needs.

Example 4.2. The user's request was for an app that would send a notification when the temperature in the Lunar Greenhouse wandered outside a narrow range. The Lunar Greenhouse is part of a study of what it would take to grow vegetables on the moon or on a long space voyage.[5] Water will be a scarce resource. The temperature in the greenhouse was therefore strictly controlled:

warm enough to grow plants, but not too warm, to limit water loss due to evaporation. Without notifications, the user, a graduate student, had to personally check conditions in the greenhouse, on a regular basis.

The development team created a notification app to meet the articulated need. They got a feed from sensors in the greenhouse and sent notifications to the graduate student's phone. The graduate student no longer needed to go to the greenhouse regularly for a simple check.

There is more to the story. Every time the temperature went out of range, the graduate student would jump on a bicycle to go to check the greenhouse. Sometimes, all was well and the temperature returned to normal, without intervention. The team then set up a video surveillance camera and enhanced the app to provide a live video feed to the user's phone, on demand. Now, the graduate student needed to go to the Lunar Greenhouse only when manual intervention was warranted.

In summary, the articulated user need was for a notification system. The latent need for video monitoring emerged out of the close collaboration between the user and the developers. The developers needed to go beyond pure listening and observing. The had to relate to what the user's motivations and frustrations to discover the latent need for monitoring. □

Barriers to Communication

Developers do not have direct access to user needs. What they can access is user behavior; that is, the words, actions, and sentiments or emotions that users share or exhibit. The arrows in Fig. 4.2 represent connections between (a) the underlying user needs; (b) how users exhibit those needs; and (c) how developers can potentially access user behavior.

The arrows are dashed to indicate potential gaps. From left to right, there can be gaps between the underlying need and what users exhibit through their behavior.

For example, in putting needs into words, users may not find just the "right" words. Even if they can find the words, they may neglect to mention things that they assume "everybody knows." Either way, their words are not a true reflection on the underlying need.

Moving right, there may be additional gaps between what users send and what developers receive. A developer may miss or misinterpret what is being communicated. Natural language is ambiguous. Furthermore, the same words can have different shades of meaning for different people. Actions and emotions can similarly be missed.

Summarizing the above discussion, the channels of communication between users and developers are noisy. That is, information can be lost as needs are exhibited in user behavior (words, actions, and emotions) and then again as user behaviors are accessed and interpreted by developers. The next section contains tips and techniques for interacting with users.

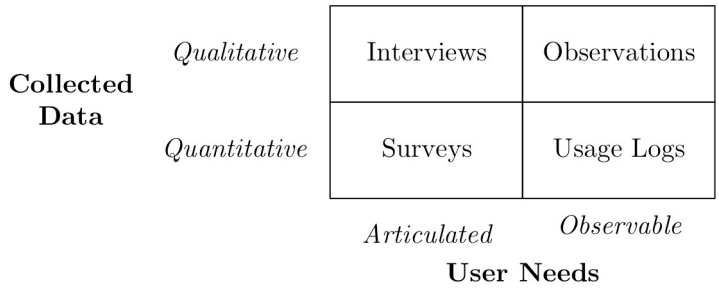


Figure 4.3: Techniques for identifying articulated and observable needs.

4.2.2 Using All Modes of Communication

What can developers do to overcome potential communication gaps, when they interact with users? This section provides some techniques for working with users to elicit their needs. Meanwhwhile, here are some general tips:

- Use multiple modes of communication. The case study in this section illustrates the use of listening, observing, and empathizing.
- Record user words, actions, and sentiments verbatim, without rephrasing them or interpreting them.
- Clarify and check that the written requirements capture the user's intent.
- Focus on the stakeholders in a given conversation. Inconsistencies with other user requirements can be resolved during analysis, after needs are identified.

It may take multiple conversations to elicit user requirements.

Listening and Observing

The qualitative and quantitative techniques in Fig. 4.3 can be used for eliciting articulated and observable needs. The left column in the figure is for articulated needs. Interviews are qualitative; that is, interview findings cannot be readily measured. Surveys, meanwhile, are quantitative; responses can be measured.

Observation of customer behavior and analysis of usage logs are techniques for eliciting observable needs; see the right column in Fig. 4.3. Observations are qualitative; usage logs are quantitative.

Establishing Rapport with Users

Listening with understanding and empathy is a big part of attending to what users say, do, and feel. In a classic paper on communication, Carl Rogers notes,

> Listening with understanding and empathy "means seeing the expressed idea and attitude from the other person's point of view, sensing how it feels to the person, achieving his or her frame of reference about the subject being discussed."[6]

Listening, really listening, is easy to state, hard to do. It is a skill that can be learned through practice. It involves attention not only to the technical content of what the other person is saying, but also to their attitude towards the subject of the communication. Hear the other person out before asking a question or changing the direction of the conversation.

Words and phrases are important to people, so resist the urge to paraphrase or reword. Marketers use the term *Voice of the Customer* for a statement "in the customer's own words, of the benefit to be fulfilled by the product or service."[7]

4.2.3 Case Study: Intuit's Design for Delight

Intuit's Design for Delight process promotes the use of all modes of communication (listening, observing, empathizing) to elicit the full range of user needs (articulated, observable, tacit, and latent). Intuit is not alone. A corresponding process, called Scenario-Focused Engineering, has been used within Microsoft. The agile requirements process in Section 3.3 is based on Scenario-Focused Engineering. See also usage scenarios in Section 4.5, which record user experiences.

Intuit's Design for Delight has three-steps:[8]

1. Develop deep empathy with customers through understanding what really matters to them and what frustrates them.
2. Offer a broad range of options before settling on a choice.
3. Use working prototypes and rapid iterations to get customer feedback.

Example 4.3. The intended users for Intuit's QuickBooks app for the iPad were one-person business owners who spent most of their time out of the office, serving their customers.[9] The business owners wanted to manage all aspects of their business while mobile. For example, they wanted to provide professional estimates and invoices on the spot, at a customer site, rather than waiting until they got back to the office. Without the ability to manage their business while mobile, work was piling up and they were spending evenings and weekends catching up.

The Intuit team used interviews, observations, and empathy to relate to user goals, frustrations, and needs. The team came up with an initial list of requirements, which they evolved based on user feedback. User requirements were captured as user stories that were implemented using agile methods. The app was instrumented to provide usage data. During iteration planning, prioritization of stories was based on both observations and analysis of usage data.

Careful observations and empathy guided the design of the user experience. As an example, the team discovered that users held their tablets differently while they were mobile and while they were at home. On the go, they favored

portrait mode, while at home they favored landscape mode. Why? The task mix on the go was different from the mix at home. At home, they dealt with more complex transactions and landscape mode allowed them to display more data on a line. □

Once a project is underway, further needs can surface as (a) the developers establish rapport with stakeholders, and (b) as working software becomes available for user feedback. In the above example, it was through the working prototype that the developers learned how users held iPad: in portrait mode in the field; and in landscape modein the office/home. This insight influenced the evolution of the user experiences supported by the app.

4.3 Clarifying User Goals

Anything a user wants to accomplish can be viewed as a user goal. The want in the user story

As a ... **I want to** get rides from trusted drivers **so that** ...

can be restated as the goal

My goal is to get rides from trusted drivers.

System features also have corresponding user goals. The feature

The system shall send notifications by text, email, or phone,
based on the user's contact preferences.

has a corresponding user goal

My goal is to receive notifications by text, email, or phone,
based on my contact preferences.

has a corresponding user goal

These goals have to be clarified before they can be implemented. Who qualifies as a trusted" driver? Under what conditions does the user want a notification? An initial conversation about a user requirement might begin with a general goal that is easy to state, and needs further discussion before it is ready for implementation.

This section introduces some simple questions that can be used to clarify user wants. It also introduces criteria for deciding when to stop clarification.

4.3.1 Properties of Goals

A goal without success criteria is called a *soft goal*. Soft goals arise naturally in conversations. In explaining the business value of a software project, users might state an aspiration such as

Improve customer satisfaction

How will customer satisfaction be measured? This soft goal can be refined into the measurable goal

Increase Net Promoter Score by 20%

Net Promoter Score is an accepted measure of customer satisfaction.

The acronym *SMART* stands for Specific, Measurable, Achievable, Relevant, and Time-bound. SMART criteria can be applied to goals, actions, questions, and so on. They are applied to user stories in Section 4.4.[10]

Example 4.4. Are the following goals SMART?

1. Get rides from rusted drivers.
2. Identify landing sites on the surface of the asteroid Bennu.
3. Get a notification when the temperature in the greenhouse goes out of range.

None of them is SMART. Goal 1 is not specific about the definition of a "trusted" driver. Goal 3 does not quantify the temperature range, so it is not measurable as stated. Goal 2 does not specify a time bound. Is a notification triggered when the temperature changes by plus or minus 5 degrees? Or is it 2 degrees? Tt turned out not to be achievable within the time bound of the student project that tackled it—the image data of the surface of the asteroid was flawed. A few years later, there was a successful landing on Bennu. □

A Temporal Classification of Goals

Frequently occurring classes of goals include the following:[11]

- *Achieve/Cease* goals eventually get to a desired state; e. g., send a notificaion.
- *Maintain/Avoid* goals keep a property invariant; e.g., keep the shopping site up 24 hours a day, 7 days a week.
- *Optimize* goals involve a comparison between the current state and the desired state; e.g., increase Net Promoter Score by 20%.

4.3.2 Asking Clarifying Questions

Some simple questions, respectfully asked, can be very helpful for identifying and clarifying user requirements. Words in a natural language can have different meanings, depending on their context in a sentence. The questions below are therefore in pseudo-English. Once readers are familiar with the questions and their intended purpose, they can rephrase the questions to fit a conversation. Let ⟨*goal*⟩ represent an English phrase for an achieve, maintain, or optimize goal.

The clarifying questions and their purpose is as follows:[12]

- *Why* questions explore the context and motivation for a goal. A *Why* question has the form

  Why do you want to ⟨*goal*⟩?

  The short form of the question is "**Why** ⟨*goal*⟩?" or simply "**Why**?" if the goal is clear from the context. For example, if the goal is "Get notifications about the temperature," then the following question explores the motivation for this goal:

  Why do you want to get notifications about the temperature?

- *Why Not* questions explore constraints or conditions on a goal or task. A *Why Not* question has either of the following forms:

  Why not ⟨*goal*⟩?
  What stops us from ⟨*goal*⟩?

- *How* questions explore subgoals or tasks that contribute to the success of a goal. In other words, the questions explore possible solution approaches. A *How* question has one of two forms:

  How can we ⟨*goal*⟩?
  How else can we ⟨*goal*⟩?

  If the goal is clear from the context, the simple forms are '**How**?" and "**How else**?". *How Else* questions bring out alternatives.

- *How Much* and *How Many* questions explore metrics and criteria for determining whether an optimize goal has been accomplished. '

During elicitation, clarifying questions can uncover information that users may not have thought to share. Users tend to neglect information that "everybody knows" in their environment. Except that developers are not in the users' environment. By respectfully asking questions, developers can elicit information that users may take for granted.

Example 4.5. Consider a conversation about providing contractors with mobile access to their business data. (This example is based loosely on Example 4.3.) The conversation might begin as follows:

> *Developer.* "Help me understand. **Why do you want to** have mobile access to business data? *Contractor.* "So I can produce invoices on the spot, at the customer site."

The conversation can then turn to what else contractors need to produce invoices. Perhaps there is additional functionality related to invoices that the app could provide. □

Example 4.6. Consider a brainstorming session with users about their interest in listening to music on the go. The following conversation illustrates the use of *How* questions:

> **How can we** provide music while mobile? Download media to a portable personal device. **How else can we** provide music while mobile? Stream music to a device over the network.

□

Goal clarification and refinement can lead to a proliferation of goals and subgoals. This proliferation can be managed by organizing goals into hierarchies; see Section **??**.

When using clarifying questions, focus on the intent of the question; e.g., to explore context or to explore solution approaches. The pseudo-English questions above can be rephrased as desired for a natural conversation.

4.4 User Requirements as User Stories

User stories are typically written using the following template from Section 2.4:

As a ⟨*role*⟩ **I want to** ⟨*task*⟩ **so that** ⟨*benefit*⟩

Here, ⟨*role*⟩, ⟨*task*⟩, and ⟨*benefit*⟩ are placeholders for English phrases. Any task a user wants to accomplish is equivalent to a user goal, as discussed in Section 4.3. We therefore treat ⟨*task*⟩ as a user goal that may require clarification.

This section provides guidelines for writing effective user stories. Specifically, the guidelines are for the English phrases to replace ⟨*role*⟩, ⟨*task*⟩, and ⟨*benefit*⟩.

4.4.1 Guidelines for Writing User Stories

What is a good user story? Above all. a story is about a user need; it is not about the technology for addressing the need. Stories must therefore be written so that users can understand them and relate to them. At the same time, stories must be written so that developers can implement them and test for them. It may take multiple conversations to outline and refine the initial set of user stories for a project.

SMART Stories

Good user stories are SMART: Specific, Measurable, Achievable, Relevant, and Time-bound.

- A story is *specific* if it is a precise and unambiguous enough that acceptance tests can be defined for it.

- It is *measurable* if success criteria are defined for it.
- It is *achievable* if the developers know how to implement it within the allotted time.
- It is *relevant* if the ⟨*task*⟩ in the story contributes to the ⟨*benefit*⟩, and the ⟨*benefit*⟩ provides business value for a stakeholder.
- It is *time bound* if it can be implemented in one iteration.

If one of these criteria is not met, then the story has to be refined until it is SMART. (SMART criteria can be applied to any goal; see Section 4.3.)

Example 4.7. The first draft of a user story may not meet all the SMART criteria. Consider the payroll story from Example 2.4:

As a	payroll manager
I want to	produce a simple paycheck
so that	the company can pay an employee

In this story, what does "simple paycheck" mean? Paychecks can be very complicated, between various forms of compensation (e.g., salary overtime, bonuses), and various deductions (e.g., taxes, savings, medical insurance).

The following refinement of the payroll story is specific about compensation and taxes:

As a	payroll manager
I want to	produce a paycheck that accounts for monthly wages and federal taxes
so that	the company can pay an employee and withhold federal taxes □

How Much Detail?

When is a story specific enough that we can stop refining it? We can stop refining a story when it is specific enough that developers can write tests for it.

Example 4.8. The refined story in Example 4.7 tells developers exactly which taxes are to be withheld: federal taxes. That is enough information for a developer to write tests about the amount withheld.

The details of the federal tax rules do not belong in a user story. The computation of federal taxes is an implementation detail that is best left to the developers. For example, it is up to the developer to decide whether to use an algorithm to compute the tax amount or to use the published tax tables to look up the amount.

Another reason for leaving the details of the tax calculations out of a story is that the tax rules change from year to year. The need to withhold taxes is constant, independent of changes to the tax calculations. □

INVEST in Stories

The INVEST acronym provides a checklist for writing good user stories. The acronym stands for Independent, Negotiable, Valuable, Estimable, Small, and Testable.[13]

- *Independent.* A good collection of stories avoids duplication of functionality across stories.
- *Negotiable.* Stories are developed in collaborations or negotiations between users and developers. They summarize functionality; they are not contracts that spell out all the details.
- *Valuable.* A good story provides value for some stakeholder.
- *Estimable.* A good story is specific enough that developers can estimate the time and effort needed to implement it.
- *Small.* A good story is small enough that it can be implemented in one iteration. Split compound or complex stories into simpler stories. See also the discussion of estimation in Chapter 6.
- *Testable.* A good story can be measured and tested.

Clarifying the Benefit

The **As-a-I-want-so-that** template contains both some functionality (the want) and the benefit that drives the want. Clear benefits help developers make implementation decisions.

If the functionality in a user story does not contribute to the stated benefit, then either the functionality is not relevant, or there may be a missing story. Stories that are not relevant can be dropped. Clear benefits can also help avoid *gold plating*, which refers to continuing to work on a project beyond the point of meeting all the stakeholder needs.

When clarifying benefits, ask "Why?" Why does a stakeholder want something? "Why" questions tend to elicit cause and relevance; see Section 4.3. Respectfully continuing to ask *Why* questions can surface an underlying need that has higher priority for the user than the starting point. The *Why* questions may either be posed explicitly or explored through other modes of communication, such as surveys and observations.

Acceptance Tests

User stories are accompanied by acceptance tests for verifying that a story has been implemented correctly. Acceptance tests are part of the conversation about stories between users and developers. The following template is helpful for writing acceptance tests:[14]

Given ⟨*a precondition*⟩
when ⟨*an event occurs*⟩
then ⟨*ensure some outcome*⟩

The payroll user story might be accompanied by a test of the form:

Given an employee is on the payroll and is single
when the paycheck is created
then use the federal tax tables for singles to compute the tax

Acceptance tests must be written using language that is meaningful to both users and developers. Acceptance tests must focus on the desired functionality, not on whether the implementation matches its specification.

4.4.2 Limitations of User Stories

Big Picture?

As the number of stories increases, it is easy to lose sight of the "big picture" of a system—the big picture can be helpful during system design. User stories are at the level of individual features. Usage scenarios, in Section 4.5 and use cases, in Chapter 5, are at the level of an overall user experience. Thus, they provide some context; they show how individual features contribute to the user experience.

User stories and use cases can be used together, with use cases providing context and user stories providing specific functionality.

Completeness?

Completeness is another issue with user stories: there is no guarantee that a collection of user stories describes all aspects of a system.

Stakeholders may have left out something that they take for granted, something that "everybody knows" in their environment. Furthermore, nonfunctional requirements, such as performance and privacy, may not have been discussed in the conversations between users and developers.

Deeper Needs?

Conversations, interviews, and surveys are well suited to identifying needs that users are able or willing to articulate in words. The **I want to** phrasing in the user story template is also suited to articulated needs.

From Section 4.2, users may have additional needs, beyond articulated needs. A skilled development team may be able to uncover additional needs through observations, intuition, and empathy. Usage scenarios do include a primary customer's emotional state. Without emotions and intuition, latent user needs are likely to remain latent.

4.5 Usage Scenarios: End-to-End Experience

A *usage scenario* is a narrative about a primary user's end-to-end experience. The narrative seeks to bring to life the user, their situation, their delights,

frustrations, and emotional state in the situation. It can lead to insights into the full range of a user's needs and wants. A usage scenario can help to immerse developers in the user's experience so they will relate to the user as they define and build a product.[15]

A usage scenario identifies

- the primary user;
- what the user wants to experience or accomplish;
- the proposed benefit to the user;
- how the proposed benefit contributes to their overall experience; and
- the user's emotional response in the overall experience.

4.5.1 Writing Usage Scenarios

A usage scenario is a narrative or a story, not a checklist of items. As a helpful suggestion, consider the following elements when writing a usage scenario:

- *Title.* A descriptive title.
- *Introduction.* An introduction to the primary user; their motivation; what they might be thinking, doing, and feeling; and their emotional state—that is, whether they are mad, glad, sad, or scared.
- *Situation.* A brief description of the real-world situation and the need or opportunity to be addressed by the product. The description puts the need in the context of an overall experience.
- *Outcome.* The outcome for the user, including success metrics and what it would take for the solution to delight the user.

Together, the introduction and situation in a usage scenario correspond to the current state. The outcome corresponds to the desired state. What a usage scenario must not include is any premature commitment to a potential implementation, to how to get from the current to the desired state.

A Checklist for Usage Scenarios

The acronym SPICIER provides a checklist for writing a good usage scenario:

S: tells the beginning and end of a *story*
P: includes *personal* details
I: is *implementation-free*
C: it's the *customer's* story, not a product story
I: reveals deep *insight* about user needs
E: includes *emotions* and *environment*
R: is based on *research*

4.5.2 Case Study: A Medical Usage Scenario

the scenario in the following example is based on a doctor's request for an app to provide reliable information for seriously ill children and their parents. From the start, the doctor described the motivation for the request (the current state) and her vision for an app (the desired state). In other words, during the first 2-3 meetings, she outlined a usage scenario. She did not have a list of features. She and her colleague were focused on overall experience for parents and kids. The desired outcome drove the iterative development of the prototype for the children's hospital.

Example 4.9. The narrative in this example has two paragraphs: one for the introduction and situation and another paragraph for the desired outcome.

> *Title*: A Medical App for Kids and their Parents
>
> *Introduction.* Alice and Bob are worried about their seriously-ill six-year old, Chris, who is seriously ill. The parents have scoured the Internet for Chris's condition, and that has left them more confused and worried than ever. They don't know who to trust. Chris has been a trooper, cheerfully engaging with the doctors and nurses. Now, Chris is scheduled for further tests and scans in the hospital. Alice and Bob are concerned about Chris being frightened at having lie still surrounded by big unfamiliar whirring machines.
>
> *Outcome.* Alice and Bob have one less thing to worry about. An app provided by the hospital gives them reliable information about Chris's specific condition. The app works on both their smartphones and a tablet, so they can sit down with Chris and go over content created specifically for kids. Chris has seen the video of the scanning machine several times. Alice and Bob also go over the graphic from the last visit to the doctor, where the doctor had used the touch screen on the tablet to highlight where monitors would be placed during the upcoming scan. Chris's biggest concern is now which playlist to choose for music during the upcoming scan.

Applying the SPICIER checklist to the above scenario, we get

S: *Story.* The narrative begins with Alice and Bob worried not only about their child's health, but by the confusing information they are getting from the Internet. They are also worried about Chris's upcoming tests and can in the hospital. The scenario ends with them having a reliable source of information.

P: *Personal.* The personal details include parents' worries about the health of their six-year old child.

I: *Implementation-Free.* The focus of the scenario is on the experience. It does not even go into the specifics of the user interface.

C: *Customer's Story.* The scenario is about Alice and Bob and their six-year old.

I: *Insight.* The scenario includes key insights into who will use the app and what the app must provide to support their desired experiences. The primary users are parents, who want reliable information. Then come children; the app needs to support a separate kid-friendly experience. Where does the content come from? If doctors add and edit content, they will need their own interface. The outcome could be taken apart, sentence by sentence, for follow-up conversations about requirements.

E: *Emotions and Environment.* The scenario is specific about the environment (a flight). The scenario starts with the parents being worried. A key purpose of the app is to prepare children so they are not frightened when they are surrounded by unfamiliar medical equipment. The scenario ends with Chris being more concerned about the music during a test than about the prospect of the test.

R: *Research.* The scenario is based on interactions with doctors at a children's hospital.

□

4.6 Conclusion

Exercises for Chapter 4

Exercise 4.1. Come up with your own examples of

a) articulated,

b) observable,

c) tacit, and

d) latent needs.

The examples can be drawn from different software projects. For each example, address the following questions: What was the need in the example? How was the need exhibited How was it accessed, in terms of listening, observing, and/or emphathy?

Exercise 4.2. Write user stories for the pool-service application in Example 1.7. Include stories from both a technician's and a manager's perspective.

Exercise 4.3. In your own words, summarize the issues that led to the failure of the BBC Digital Media Initiative, described in Example 4.1.

Write 4 user stories based on the following scenario. Let these be the 4 highest priority user stories, based on perceived business value for the client.

Your team is doing a project for a nationwide insurance company that prides itself on its personalized customer service. Each customer has a designated insurance agent, who is familiar with the customer's needs and preferences. The company has engaged you to create an application that will route customer phone calls and text messages to their designated agent.

And, there may be times when the designated agent is not available or is busy with someone else. If a customer needs to speak to someone—say, to report an accident—then, as a backup, the automated application must offer to connect the customer with another agent at the local branch (preferred) or at the regional support center, which is staffed 24 hours a day, 7 days a week. The regional center will be able to help the customer because the application will simultaneously send both the phone call and the relevant customer information to the regional agent's computer.

At any choice point, customers will be able to choose to leave a voice message or request a call-back.

Exercise 4.4. Write 4 user stories based on the following scenario. Let these be the 5 highest priority user stories, based on perceived business value for the client.

Your team is doing a project for a nationwide insurance company that prides itself on its personalized customer service. Each customer has a designated insurance agent, who is familiar with the customer's needs and preferences. The company has engaged you to create an application that will route customer phone calls and text messages to the designated agent.

And, there may be times when the designated agent is not available or is busy with someone else. If a customer needs to speak to someone—say, to report an accident—then, as a backup, the automated application must offer to connect the customer with another agent at the local branch (preferred) or at the regional support center, which is staffed 24 hours a day, 7 days a week. The regional center will be able to help the customer because the application will simultaneously send both the phone call and the relevant customer information to the regional agent's computer.

At any choice point, customers will be able to choose to leave a voice message or request a call-back.

Chapter 5

Use Cases

> "The reason for the success of the use-case approach is not just that it is a very practical technique to capture *requirements* from a usage perspective or to design practical *user experiences*, but it impacts the whole development lifecycle."
>
> — *Ivar Jacobson, Ian Spence, and Brian Kerr, reflecting on three decades of experience with use cases, which were introduced by Jacobson in the 1980s.*[1]

5.1 Elements of a Use Case

A *use case* describes how a user interacts with a system to accomplish something of value to the user. The three main elements of a use case can be identified by asking:

a) Who is the use case for? Call that user or role the *primary actor* of the use case.

b) What does the primary actor want to get done with the system? Call what the *user goal* of the use case.

c) What is the simplest sequence of actions, from start to finish, by users and the system that successfully accomplishes the user goal? Call that sequence the *basic flow* of the use case.

As might be expected, the main elements of a use case are the primary actor, the goal, and a basic flow.

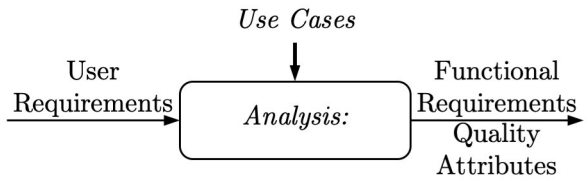


Figure 5.1: Use cases span iser and functional requirements.

Use cases are intended to be a single, plain English, description of a system that is suitable for all stakeholders. A well written collection of use cases characterizes a system's externally visible behavior. User goals provide an overview of the system. As we shall see, basic flows are helpful for an intuitive understanding of the system's responses to user actions,

Use cases are developed during requirements analysis; see Fig. 5.1. The user goals for use cases are extracted from user requirements. As basic flows and additional behaviors are added, use cases evolve into a description of the external behavior of the system. In other words, use cases evolve from an outline (the user goals) into functional requirements.

Use cases do not address quality attributes, which are also known as non-functional requirements. Thus, scale, performance, reliability, and other quality attributes, have to be specified separately.

The rest of this section explores the main elements of use cases: actors, user goals, and basic flows.

Actors Represent Roles

In order to allow for multiple user roles and automated entities, discussions of use cases refer to "actor" rather than "user." An *actor* represents a role or an entity. For example, Alice can be both the manager of a team and an employee of a company. Alice would be represented by two actors: manager and employee. As another example, an actor might be an external map service or a storck-price service.

In most of the examples in this chapter, users have a single role. For convenience, we use the terms "actor" and "user" interchangeably, unless there is a need tdistinguish between them. Hence, primary actor and primary user are equivalent t, unless stated otherwise.

Note that a use case can have multiple actors. A use case for a phone call has two actors: the caller and the callee, with the caller being the initiator (the primary actor).

Identifying User Goals

A prioritized set of usr goals provide a "big picture" overview of what users want from a system. Ideally, goals can be extracted from user requirements . Goal identification begins during requirements elicitation and continues into analysis. *Why* questions were introduced in Section 4.3.2 to explore the motivation and contexxt for a user want.

User feedback on the user goals for a system can reveal missing goals. It can also expose conflicts between stakeholder goals. As a small example, the security team's requirement for strong frequently changed passwords can conflict with the user requirement for passwords they can remember. Goal analysis is discussed in Section **??**.

More generally, in the process of writing use cases, any inadequacies in the user requirements have to be addressed, in consultation with users. Any inconsistencies have to be resolved. For example, consider the goal of booking a flight. Writing a basic flow from start to finish, is akin to debugging the user requirements for booking a flight.

v

Flows and Basic Flows

Any linear sequence of actions by either an actor or the system is called a *flow*. A *basic flow* has two additional properties (in addition to being a flow):

- A basic flow extends from the start of a use case to the end of the use case.
- A basic flow tells a story in which the primary actor accomplishes the user goal without any complications, such as errors and exceptions.

  i

When the primary actor and the user goal are clear from the context, a simple use case can be shown by writing just the basic flow.

Example 5.1. It is common practice to write a flow as a single numbered sequence that includes both actor actions and system responses. The following basic flow describes how a user (the primary actor) interacts with the system to resizing a photo (the user goal):

1. The user opens a photo.
2. The system offers a photo-editing menu.
3. The user chooses to resize by dragging.
4. the user drags a corner of the photo.
5. The system displays the resized photo.
6. The user exports the resized photo.
7. The user quits the system. □

Basic Flow: Make a Phone Call

1: The caller provides the callee's phone number.
2. The system rings the callee's phone.
3. The callee answers.
4. The caller and callee talk.
5. The callee disconnects.
6. The caller disconnects.
7.. The system logs the phone call.

Figure 5.2: A basic flow for a phone call.

Here are some developing basic flows; see also Section 5.3 for tips on writing use cases.

- Write text in language that is meaningful for all stakeholders, including users, developers, and testers.
- Start each action in a flow with either "The actor ..." or "The system ...;" see Example 5.1.
- For each basic flow, choose the simplest way of reaching the goal, from start to successful finish.
- See Section 5.2 for alternative flows, which represent additional behavior, beyond the basic flow.

a

5.2 Alternative Flows

The complexity of most systems is due to the many options, special cases, exceptions, and error conditions that the systems must be prepared to handle.

Example 5.2. labelex-phone-usecases Use cases were motivated by the problem of modeling phone systems, which can be enormously complex.[2] There is a simple basic flow, however, for making a phone call. The flow in Fig. 5.2 achieves a caller's goal of connecting with a callee.

The basic flow avoids the complexity of handling cases like the following. What if the callee does not answer? What if the destination phone is busy? What if the destination phone number is no longer in service? What if the callee has turned on a do-not-disturb feature? The simple idea of a caller connecting with a callee can get lost among the myriad special cases. □

Informally, alternative flows model variants of the success scenario represented by a basic flow. They model behaviors that are required of a system, but are not central to an intuitive understanding of the system.

Example 5.3. Many systems begin by authenticating a user. In a use case, the basic flow would handle successful authentication. An alternative flow would handle authentication failure. The system is required to gracefully handle authentication failures, but their handling does not tell us much about the purpose of the system. □

During discussions with stakeholders, alternative flows can be elicited by exploring alternatives to the actions in the basic flow. Such discussions can lead to clarifications and refinements of the basic flow itself.

This section considers two kinds of alternative flows: specific and bounded. Looking ahead, specific alternative flows replace specific actions in a basic flow; e,g., replacing successful authentication and related actions by unsuccessful authentication and its related actions. Looking ahead again, bounded alternative flows can be triggered at unexpected points in a basic flow; e.g., they can model the handling of exceptions such as network failure.

For convenience, the term alternative flow by itself refers to a specific alternative flow. The term bounded alternative flow will be spelled out in full; it will not be abbreviated.

5.2.1 Specific Alternative Flows

A *specific alternative flow* is formed by replacing a contiguous subsequence of actions in a basic flow. This definition allows a basic flow to have alternative flows; it does not allow alternative flows to have their own alternative flows.

Example 5.4. Consider the basic flow in Fig. 5.2, which connects a caller with a callee. The following specific alternative flow handles the case where the caller leaves a message because the callee does not answer in time. This alternative flow was formed by replacing actions 3-5 in the basic flow:

Alternative Flow *A*: Leave a Message
1. The caller provides the callee's phone number.
2. The system rings the callee's phone.
3.1a. The system invites the caller to leave a message.
3.2a. The caller records a message.
6. The caller disconnects.
7. The system logs the phone call.

□

While writing an alternative flow, we do not need to repeat the unchanged segments of the basic flow. It is enough to show how the new sequence of actions diverges from and reconnects with the basic flow.

Example 5.5. As another variant of the basic flow in Fig. 5.2, consider the case where the caller provides an invalid phone number. This time, only the replacement actions are shown, along with instructions for where they attach to the basic flow.

Alternative Flow *B*: Invalid Phone Number
After Action 1 in the basic flow, if the phone number is invalid,
2b. The system gives an error message.
Resume the basic flow at Action 6.

□

5.2.2 Extension Points

Instead of writing alternative flows by referring to numbered actions in a basic flow, we can give names to points in a basic flow. Named points decouple an alternative flow from sequence numbers in the basic flow, so the sequence numbers can be changed without touching the alternative flows. Sequence numbers can change as a use case evolves, so .

Extension points are named points between actions in a flow. In addition, there can be extension points just before the first action and immediately after the last action of a flow. The term "extension point" is motivated by their use for attach additional behavior that extends the system. We follow the convention of writing the names of extension points in boldface. Within basic flows, extension points will be enclosed between braces, { and }.

Example 5.6. Cash withdrawal from an automated teller machine (ATM) is a classic example of a use case.[3] The basic flow in Fig. 5.3 has four extension points. This example uses two of them, **Bank Services** and **Return Card** to attach an alternative flow for authentication failures.

Alternative Flow *A*: Fail Authentication
At {**Bank Services**}, if authentication has failed,
Display "Authentication failed."
Resume the basic flow at {**Return Card**}.

□

Example 5.7. The alternative flow for insufficient funds attaches between {**Dispense Cash**} and {**Return Card**}:

Alternative Flow *B*: Insufficient Funds
At {**Dispense Cash**}, if account has insufficient funds,
Display "Insufficient funds in account."
Resume the basic flow at {**Return Card**}.

□

BASIC FLOW: WITHDRAW CASH

1. The cardholder inserts a card.
   { **Read Card** }
2. The system prompts for a passcode.
3. The cardholder enters a passcode.
4. The system authenticates the cardholder.
   { **Bank Services** }
5. The system displays options for bank services.
6. The cardholder selects Withdraw Standard Amount.
7. The system checks the account for availability of funds.
   { **Dispense Cash** }
8. The system dispenses cash and updates the account balance.
   { **Return Card** }
9. The system returns the card.

Figure 5.3: A basic flow with extension points in boldface.

A Graphical Aside

The graphical view in Fig. 5.4 is included solely to illustrate the relationship between a basic flow and its alternative flows. It is NOT RECOMMENDED for writing use cases because the separation of basic from alternative flows is central to the simplicity of use cases. To combine them in one representation, would be bad practice.

For graph in Fig. 5.4 uses arrows to show how specific alternative flows attach to the basic flow for cash withdrawals from an ATM. Flows correspond to paths through the graph. The basic flow goes sequentially through the numbered actions, from the start to the end of the use case. The two alternative flows in the figure are for handling authentication failure and insufficient funds.

5.2.3 Bounded Alternative Flows

Bounded alternative flows attach anywhere between two named extension points in a basic flow.

Example 5.8. What if the network connection between the ATM and the server is lost? Connection loss is an external event that can happen at any time; it is not tied to any specific point in the basic flow.

For simplicity, we assume that the actions in the basic flow are atomic. In particular, we assume that the action "Dispense cash and update account balance" is handled properly; that is cash is dispensed if and only if the account is updated. A real system would use transaction processing techniques to ensure that the account balance reflects the cash dispensed.

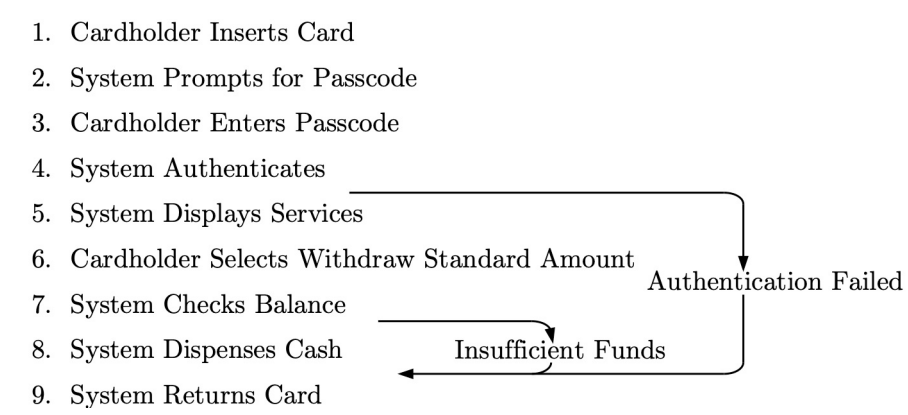


Figure 5.4: Basic and specific alternative flows for cash withdrawals from an ATM. Such a graph representation does not readily accommodate bounded alternative flows.

With the basic flow and extension points in Fig. 5.3, the following bounded alternative flow returns the card if the network connection is lost before cash is dispensed:

Bounded Alternative Flow: Lose Network Connection
At any point between {**Read Card**} and {**Dispense Cash**}, if the network connection is lost,
Display "Sorry, out of service."
Resume the basic flow at {**Return Card**}.

□

aa

5.3 Writing Use Cases

The promise of use cases is that they can serve as a single description for all stakeholders. For this promise to be fulfilled, use cases have to be written so they are readable enough for users, yet specific enough for developers.

The structure of use cases allows them to be read at multiple levels. The actors and goals in a collection of use cases serve as an outline of a system. Basic and alternative flows provide increasing levels of detail.

The structure of use cases also supports incremental development of use cases. This section provides a template, where elements can be added incrementally, as needed. The section closes with tips for developing use cases.

Name:	Active phrase for what actor(s) want to achieve
Goal:	Brief description of the purpose of the use case
Actors:	The primary and other roles, human or automated
Basic Flow:	Sequence of actions—text, readable by stakeholders
Alternative Flows:	List of variations on the basic flow
Extension Points:	Insertion points in the basic flow for additional behavior
Preconditions:	System state for the use case to operate correctly
Postconditions:	System state after the use case completes
Relationships:	Possible connections with other use cases

Figure 5.5: A representative template for writing use cases, with elements listed in the order they can be added.

5.3.1 A Template for Use Cases

aa

There is no standard format for use cases. Templates do exist, however. A representative template appears in Fig. 5.5. The elements in the template are listed in order of decreasing priority.

Name and Goal. The first element in the template is a name for the use case. The name is preferably a short active phrase such as "Withdraw Cash" or "Place an Order." Next, if needed, is a brief description of the goal of the use case. If the name is descriptive enough, there may be no need to include a goal that says essentially the same thing.

The Basic Flow. The basic flow is the heart of a use case. It is required. Begin a basic flow with an actor action. The first action typically triggers the use case; that is, it initiates the use case.

Alternative Flows. Long use cases are hard to read, so the template asks only for a list of alternative flows. Identify an alternative flows by its name or its goal. Since requirements can change, alternative flows need not be fleshed out until the need arises.

Alternative flows must be about optional, exceptional, or truly alternative behavior. Otherwise, the relevant actions may belong in the basic flow. If the behavior is not conditional, it is not alternative behavior and may not belong in this use case; it may belong in some other use case.

Extension Points. Use extension points in the basic flow only to indicate points for inserting additional behavior. Alternative flows, both specific and

bounded, attach to a basic flow at named extension points; for examples, see Section 5.2.

Preconditions and Postconditions. Preconditions are assertions that must be true for the use case to be initiated. For example, a Cancel Order use case may have a precondition that an order exists or is in progress. If there is no order, there is nothing to cancel. Similarly, postconditions are assertions that must be true when the use case ends. In simple examples, preconditions and postconditions are often omitted.

Relationships. See Section 5.5 for relationships between use cases.

5.3.2 From User Intentions to System Interactions

For perspective on the right level of detail in a use case, consider the distinctions between the three words intention, interaction, and interface; as in user intentions, system interactions, and user interfaces. These three words represent increasing levels of detail. For example,

User Intention:	authenticate
System Interaction:	use a passcode to authenticate
User Interface:	enter a passcode on a keypad

The intention level is suitable during early stakeholder discussions, where the focus is on what users want. The interaction level is suitable once user goals have settled and the focus can shift to the desired system response to a user action. Use cases do not usually go down to the interface level, where the focus is on the design and implementation of the user interface.[4]

Intentions are technology-free and implementation-free. Exceptions and special cases need not come up during a discussion about user intentions. The intent to authenticate is independent of the authentication technology. It does not matter whether the technology is fingerprint, password, or a verification code sent to the user's device.

Interactions are implementation-free, but may have technology dependencies.

Example 5.9. The following flow is at the interaction level; it is geared to a specific technology for authentication:

1a. The user initiates Login.
2a. The system sends a verification code to the device on file.
3a. The user supplies the verification code.
4a. The system authenticates.

Compare actions 2a-3a with the following flow at the intention level:

2b. The system prompts for identification.
3b. The user provides credentials.

Note that "prompt for identification" does not commit to any specific technology for authentication, whether it's by password, by sending a verification code to a device, by fingerprint, or by some other method. □

Interfaces can depend on design and implementation choices such as the layout of graphics, buttons, and text. Interface and implementation details do not belong in use cases, although some design and implementation choices may creep in.

5.3.3 How to Build Use Cases

Use cases can be built incrementally. A full-blown use case can involve significant effort, some of which may be wasted if requirements change. The following is a suggested order for iteratively building use cases, as a project proceeds:[5]

1. Begin with a list of actors and goals for the whole system. Such a list can be reviewed with users to (a) validate and prioritize the list of stakeholders and goals; and (b) explore the boundaries of the proposed system.

2. Draft basic flows and acceptance tests for the prioritized goals. The draft can focus on user intentions, with system interactions included as needed. Acceptance tests are helpful for resolving ambiguities and for reconciling the basic flow and the goal of a use case.

3. Identify and list alternative flows; see Section 5.2. A list of alternative flows may be enough to guide design decisions and to create backlog items during development.

4. Finally, develop alternative flows as needed during development.

5.4 Use-Case Diagrams

A *use case diagram* summarizes the actors, the goals, and the interactions between actors and goals. Alternatively, a use case diagram summarizes the use cases for a system, since goals correspond to use cases and vice versa. In diagrams, an actor is represented by a stick figure; see Fig. 5.6. A use case is represented by an ellipse labeled with the goal of the use case. Interactions are represented by arrows. The arrow starts from the initiator of an interaction. Note that an arrow represents a dialogue that potentially involves an exchange of messages back and forth. If there is no arrowhead, then either party can initiate the interaction.

A Diagram Provides a Big Picture Summary

Diagrams are intended for an overall understanding of a system with multiple use cases. Collectively, the goals of the use cases describe the purpose of the

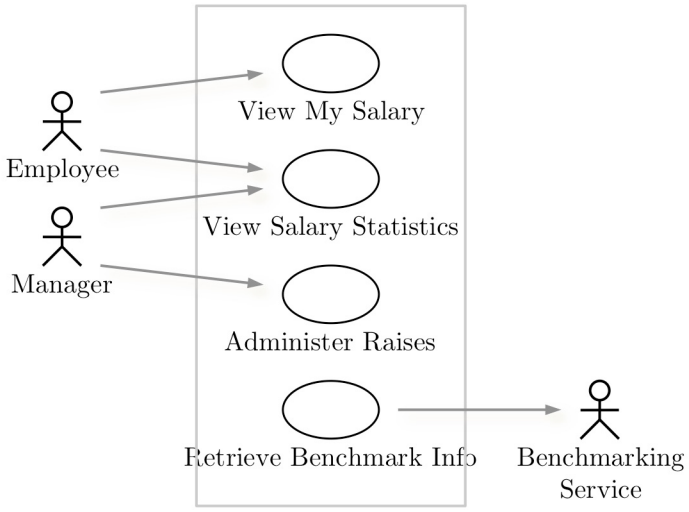


Figure 5.6: Use case diagram for a salary system.

system. As discussed in Section 5.3, actors and goals are a good starting point for developing the use cases for a system.

Example 5.10. The diagram in Fig. 5.6 shows the actors and use cases for a salary system. The two primary actors are employee and manager.

Employees can view their own salaries by initiating the View My Salary use case. They can also initiate View Salary Statistics for perspective on how their salaries compare with similar roles both within their company and within their industry. Information about salaries outside the company is provided by a benchmarking service, which is shown as a secondary actor (on the right).

Managers can view salary statistics and can administer raises for the people they manage. The system can initiate the Retrieve Benchmark Info use case to get industry salary statistics from the Benchmarking Service actor. □

Use Case Diagrams in Practice

Use case diagrams are part of UML (Unified Modeling Language), a standard graphical language for modeling and designing software systems; see Section **??**. Based on empirical studies of UML usage, the concepts of use cases are used more than the diagrams themselves. Use cases may be used in combination with textual descriptions, as in the following comments by participants in a survey:

> "It's hard to design without something that you could describe as a use case."

> "Many described use cases informally, for example, as: 'Structure plus pithy bits of text to describe a functional requirement. Used to communicate with stakeholders.'"[6]

5.5 Relationships Between Use Cases

Most systems can be described by self-contained collections of use cases. A self-contained use case may benefit from a private subflow that serves a well defined purpose. Inclusion and extension relationships between use cases must be handled with care. As an influential book on use cases notes,

> "If there is one thing that sets teams down the wrong path, it is the misuse of the use-case relationships."[7]

Subflows

Even for simple systems, the readability of flows can be enhanced by defining subflows: a *subflow* is a self-contained subsequence with a well defined purpose. The logic and alternatives remain tied to the basic flow if subflows are linear, where all the actions are performed or none of them are.

A subflow is *private* to a use case if it is invoked only within that use case.

Inclusion of Use Cases

An *inclusion* is a use case that is explicitly called as an action from a basic flow. The callee (the inclusion) is unaware of its caller. We might use the idea to A possible application is to partition a use case by factoring out some well defined behavior into an inclusion (a use case of its own). Alternatively, two or more use cases can share the common behavior provided by an inclusion.

Example 5.11. Authentication is a good candidate for an inclusion, since it is a well-defined frequently-occurring subgoal. Consider two use cases, *Withdraw Cash* and *Transfer Funds*, which share an inclusion, *Authenticate Cardholder*. Action 3 in the following flow treats the inclusion as an action:

1. The actor inserts a card
2. The system reads the card
3. Include use case *Authenticate Cardholder*
   { **Bank Services** }
4. The system displays options for bank services

Upon successful authentication, the flow resumes at Action 4. See Example 5.6 for how to handle authentication failure by attaching an alternative flow at extension point **Bank Services**. □

Extensions of Use Cases

An *extension* is a use case V that is invoked at a named extension point p in another use case U. The use case U is complete by itself and is unaware of the extension V. Extension is a form of inheritance. The purpose of an extension is to support an additional goal for an actor.

The distinction between inclusion and extension is as follows:

- *Inclusion.* A use case knows about the inclusion that it calls. The inclusion use case is unaware of its caller.
- *Extension.* An extension knows the underlying use case that it extends. The underlying use case provides extension points, but is unaware of the extension use case.

Example 5.12. Suppose that *Place Order* is a complete use case that allows shoppers to select a product and place an order. Its basic flow has an extension point **Display Products**:

Basic Flow:Placee Order
1. The shopper enters a product category.
2. The system gets products.
{ **Display Products** }
3. The system displays products and prices.
4. The shopper places an order.

Now, suppose that the shopper has the additional goal of getting product recommendations. The extension use case, *Get Recommendations* invokes itself at the extension point in *Place Order*:

Extension: Get Recommendations
At {**Display Products**} in use case *Place Order*
⟨ system makes recommendations ⟩

With the extension, the shopper enters a product category, gets recommendations, then sees the products and prices, and places an order. The *Place Order* use case remains unchanged. □

As another example, consider a surveillance system with a use case to monitor an area for intruders. Another use case notifies authorities. Surveillance and notification can be combined by making notification an extension of the surveillance use case.

At an extension point **Intruder Detected** in the surveillance flow, the notification use case can notify the authorities that an intruder has been detected.

5.6 Conclusion

Summary of Use Cases

A well written collection of use cases models the externally visible behavior of the system, so it can serve as a set of functional requirements for the system. A *use case* has three required elements: a primary actor; the actor's goal with the system; and a basic flow, which describes a complete user-system interaction that ends to accomplishes the goal. The term *actor* refers to a role, human or automated.

A list or diagram of actors and their goals provides an overview of a system. It can be used in stakeholder discussions to validate and prioritize the actors and goals.

A *flow* is a linear sequence of actions by an actor or the system. The *basic flow* of a use case is a flow that (a) extends from the start of the use case to the end; and (b) successfully accomplishes the goal of the use case in the simplest possible way, without any complications. The separation of special cases, exceptions, and conditional behavior is key to the readability of use cases. Use cases are built around basic flows.

Special cases, errors, exceptions, and any other alternative behaviors are handled by using alternative flows; see Section 5.2. An *extension point* is a named point between actions in a basic flow. A *specific alternative flow* represents conditional behavior that diverges from the basic flow at an extension point and rejoins the basic flow at a later extension point. It replaces the segment of the basic flow between the two extension points. A *bounded alternative flow* is triggered at any point between two named extension points in the basic flow.

In short, actors and goals outline the purpose of a system. Basic flows are helpful for an intuitive understanding of system behavior. Alternative flows fill in alternative behaviors that are required, but are not essential to an intuitive understanding of the system. This ordering—actors and goals, basic flows, alternative flows—is a natural ordering for incrementally building the collection of use cases for a system. A list of the names or goals of the alternative flows is enough to begin with. Details can be added as needed.

Use Cases and User Stories

v Use cases and user stories are complementary; they provide different perspectives on requirements and can be used together.[8]

Use cases Provide Context. Whereas use cases describe end-to-end scenarios, user stories correspond to features or snippets of functionality. In introducing user stories along with Extreme Programming, Kent Beck noted, "you can think of [user stories] as the amount of a use case that will fit on an index card."[9] Use cases provide context because they illustrate how individual features fit together in a flow. Flows also help surface any gaps in the functionality needed to implement a system.

User Stories are Lighter Weight. User stories require less effort to create than use cases because a use is closer to a collection of related user stories than it is to an individual story.

Note, however, that a use case need not be fully developed up front; it can be evolved as a project progresses. roject can opt for a combination of context and light weight by incrementally developing both use cases and user stories.

Use Cases and Iterative Development

Ivar Jacobson and his colleagues provide three guidelines for iterative software development based on use cases:[10]

- *Build the system in slices.* Instead of implementing an entire use case all at once, consider slicing it, where a "slice" is a subset of the flows in the use case, along with their test cases. In other words, a slice corresponds to a set of paths through a flow graph, such as the one in Fig. 5.4.
- *Deliver the system in increments.* Use an iterative process, based on delivering slices of use cases. Begin with the slice or slices that provide the most value.
- *Adapt to meet the team's needs.* Fit the development process to the project. A small cohesive team in a close collaboration with stakeholders, might document just the bare essentials of use cases, relying on informal communication to address any questions along the way. A large team would likely require documented use cases with key details filled in.

Exercises for Chapter 5

Exercise 5.1. What is the difference between

a) use cases based on intention and interaction?

b) user stories and use cases?

c) specific and bounded alternative flows?

d) inclusion and extension of use cases?

Answer in your own words.

Common Instructions for Exercises 5.2-5.7

In each of the following exercises write use cases that include a descriptive name, a primary actor, the primary actor's goal, and the following: :

a) A basic flow

b) A full specific alternative flow

c) A full bounded alternative flow

d) How the alternative flows attach to the basic flow using extension points

e) Inclusion of use cases

For each alternative flow, show the full flow, not just the name of the flow. For inclusions, provide both a descriptive name and a brief comment about the role of the included use case.

Exercise 5.2. Write a use case for the software for the insurance company scenario from Exercise 4.4.

Exercise 5.3. Write a use case for the software to control a self-service gasoline pump, including the handling of payments, choice of grade of gas, and a receipt. In addition, when the screen is not being used otherwise, the system must permit targeted advertising on the screen. Targeted means that the ads are based on the customer's purchase history with the vendor.

Exercise 5.4. Prior to meeting with the customer, all you have is the following brief description of a proposed system:

> The system will allow users to compare prices on health-insurance plans in their area; to begin enrollment in a chosen plan; and to simultaneously find out if they qualify for government healthcare subsidies. Visitors will sign up and create their own specific user account first, listing some personal information, before receiving detailed information about the plans that are available in their area.[11]

Write a use case based on this description.

Exercise 5.5. HomeAway allows a user to rent vacation properties across the world. Write a use case for a renter to select and reserve a vacation property for specific dates in a given city.

Exercise 5.6. Write a use case for an airline flight-reservations system. For cities in the United States, the airline either has nonstop flights or flights with one stop through its hubs in Chicago and Dallas. Your reservations system is responsible for offering flight options (there may be several options on a given day, at different prices), seat selection, method of payment (choice of credit card or frequent flier miles). Another team is responsible for the pricing system, which determines the price of a round-trip ticket.

Exercise 5.7. Write a use case for the software to send a text message between two mobile phones, as described below. Each phone has its own Home server in the etwork, determined by the phone's number. The Home server keeps track of the phone's location, billing, and communication history. Assume that the source and destination phones have different Home servers. The destination Home server holds messages until they can be delivered. Also assume that the network does not fail; that is, the phones stay connected to the network.

Chapter 6

What to Build?

> "In many situations, people make estimates by starting from an initial value that is adjusted to yield the final answer. ... We call this phenomenon anchoring."
>
> — *Amos Tversky and Daniel Kahneman*[1]

6.1 Introduction

Between them, Chapters 4-6 address the related questions:

- What do users want?
- What to build?

Chapter 4 addresses the first question: how to interact with users to identify their wants and then how to record tho wants as user requirements. Use cases, Chapter 5 span the two questions: they can be written to focus on user intentions (wants) or on user-system interactions (what to build).

This chapter focuses on what to build, at the project level and at the iteration level. Sections 6.2-6.4 deal with the prioritization of requirements. The prioritization consists of classifying requirements into a small number of categories. The simplest approach is to group user stories into three buckets marked 1, 2, or 3 story points, in order of increasing complexity. Section 6.2 considers classification based on a single factor, such as development effort. Section 6.3 considers classification based on combinations of factors, such as value, cost,

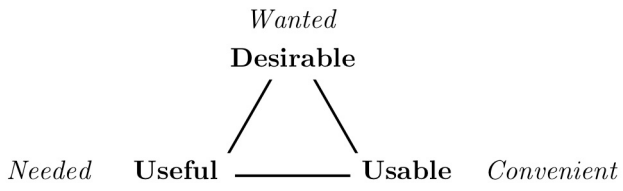


Figure 6.1: Key product attributes. Great products address needs, wants, and ease of use.

and risk. Section 6.4explores classification based on not only what satisfies customers, as well as what dissatisfies them.

Goal analysis, Section 6.5, can be used to refine what users want into potential work items that developers can build. The analysis can also reveal conflicts between user goals. Discussions with users are needed to resolve such conflicts.

The chapter concludes with a brief discussion of empirical models that have been used for time and effort estimates during plan-driven development.

6.1.1 Primary Customers

You can't please all of the people all of the time, as the saying goes. While the needs of all stakeholders must be addressed, it helps to focus on a class of primary stakeholders (customers). We can then begin by defining a product that is useful, usable, and desirable for "the" primary customer and then adapting it meet the needs of other stakeholders. For example, products that have been designed for accessibility have benefited all users.

InIn addition to being useful, great products are usable and desirable; see Fig. 6.1.[2]

- *Useful.* They serve a need and will be used.
- *Usable.* They are easy to use and can either be used immediately or can be readily learned.
- *Desirable.* Customers want them!

These product attributes are independent of each other:

- A product can be useful and desirable, but not usable; e.g., a product with a complex interface that is hard to use. Apple is known for its great products; however, after one of its product announcements, the news headline was "Innovative, but Uninviting." Why? Because "using the features is not always easy."[3]
- A product can be usable and desirable, but not useful; e.g., fashionable apps that are downloaded, but rarely used.

- A product can be useful and usable, but not desirable; e.g., an application that gets treated as a commodity or is not purchased at all. Section 6.4 explores 'useful features that are taken for granted if they are implemented, but cause dissatisfaction if they do not live up to expectations.

Example 6.1. Different stakeholders can have different criteria for prioritizing user requirements. A speech-therapy app had two groups of stakeholders: parents who bought the app and children who used it. Both were equally important, so the app had two primary customers.

For parents, useful came first, then usable. They bought the app because they felt it would be useful for their children, but were frurstrated because they found the app hard to use. For children, desirable came first, then usable. They had no trouble using the app, but were bored because it was like a lesson.

The wishes of both groups of stakeholders were met by revisiting requirements and redesigning the product. The user interface was simplified to make it easier to use for parents. The app was made for engaging for children by making it like a game. u This example is based on Example 1.3. □

6.1.2 Cognitive Bias and Anchoring

Before considering analysis techniques that rely on experience and judgment, we pause to reflect on the potential for bias in human decision making. *Cognitive bias* is the human tendency to make systematic errors in judgment under uncertainty. *Anchoring* occurs when people make estimates by adjusting a starting value, which we call an *anchor value*. The anchor value introduces cognitive bias because the new estimates reflect adjustments to the new anchor value.[4]

Simply stated: Avoid bias. In order to get unbiased estimates from developers, avoid telling them about customers or management expectations about effort, schedule, or budgets.

The next example illustrates that including an anchor value in a requirements document can lead to biased estimates.

Example 6.2. Participants in a case study were asked to estimate the time it would take to deliver a software application.[5] Each participant was given a 10-page requirements document and a 3-page "project setting" document. The project setting document had two kinds of information: (1) a brief description of the client organization, including quotes from interviews; and (2) background about the development team that would implement the application, including the skills, experience, and culture of the developers.

The 23 participants were divided into three groups. The only difference between the instructions to the three groups was a quote on the second page of the project setting document; see Fig. 6.2. The quote was supposedly from a middle manager.

Group 1, the control group, got a quote with no mention of the time it might take to develop the application. Group 2 got a quote with a low anchor: 2 months. Group 3 got a high anchor: 20 months.

- **Group 1 was given a quote with no mention of time**
  - "I admit I have no experience estimating."
- **Group 2 got a quote that mentioned 2 months.**
  - "I admit I have no experience with software projects, but I guess this will take about 2 months to finish."
- **Group 3 got a quote that mentioned 20 months.**
  - "I admit I have no experience with software projects, but I guess this will take about 20 months to finish."

Figure 6.2: Case study of the effect of anchoring on software effort estimation.

The results confirmed the phenomenon of anchoring. The mean development-time estimates from the three groups were 8.3 months for the control group, 6.8 months for the 2-month group, and 17.4 months for the 20-month group. The mean estimates from the experienced participants in the three groups were 9 months, 7.8 months, and 17.8 months, respectively. The differences between the low-anchor and high-anchor groups are clear: 7-8 months versus 17-18 months. Further studies would be needed to explain the small differences between the control group and the low-anchor group. □

6.2 Rough Estimates of Development Effort

Size and effort estimates for work items are used to answer questions like the following: How big is this project? When could we deliver? How much can we complete in this iteration? What will the project cost? Is the project on track? Corrective action can be taken if early estimates indicate that some aspect of a project is in trouble.

Predictions are difficult, especially about the future, as the Danish saying goes.[6]

In estimating size and effort, comparisons with past projects help, since people are better at predicting relative, rather than absolute, magnitude. Given work items A and B, it is easier to estimate whether A requires more or less effort than B, or whether A is simpler or more complex than B. It is harder to estimate the code size of an implementation pf either A or B.

This section begins with relative estimates for user stories.

It then explores how a group of experts can do better than an individual. Variants of the group technique go by names such as Delphi (the original name) and Planning Game or Planning Poker (used in agile circles). Finally, Three-Point Estimation provides a simple formula for estimating absolute magnitude. The formula appears to work well in practice.

6.2.1 Agile Story Points

Quick estimates can be made by assigning *points* to user stories, where a point is a unit of work. Points are relative: a 1 point story is simpler than a 2 point story, and so on. Point-based estimation techniques rely on the judgment of developers. Based on their experience and intuition, the developers agree on how to assign points to a story. Point systems are specific to a team; a different team might assign points differently.

Point Values 1, 2, 3

The simplest point system has point values 1, 2, and 3, corresponding to easy, medium, and hard. For example, the developers might assign points as follows:

- 1 *point* for a story that the team knows how to do and could do quickly (where the team defines quickly).
- 2 *points* for a story that the team knows how to do, but the implementation of the story would take some work.
- 3 *points* for a story that the team would need to figure out how to implement.

Hard or 3-point stories are candidates for splitting into simpler stories.

Fibonacci Story Points

With experience, as the team gets better at assigning points to stories, they can go beyond an easy-medium-hard or three-value scale. A Fibonacci scale uses the point values $1, 2, 3, 5, 8, ...$ The reason for a Fibonacci, rather than a linear $1, 2, 3, 4, 5, ...$ scale is that points are rough estimates and it is easier to assign points if there are some gaps in the scale. The gaps leave room for a new story to be slotted in, based on its relative complexity.

6.2.2 Velocity

With any point scale, a team's *velocity* is the number of points of work it can complete in an iteration.

Example 6.3. In Fig. 6.3, the team's estimated velocity was 17, but it only completed 12 points worth of stories in an iteration. For the next iteration, the team can adjust its estimated velocity based on its actual velocity from recent iterations. One possible approach is to use the average velocity for the past few, say 3, iterations. □

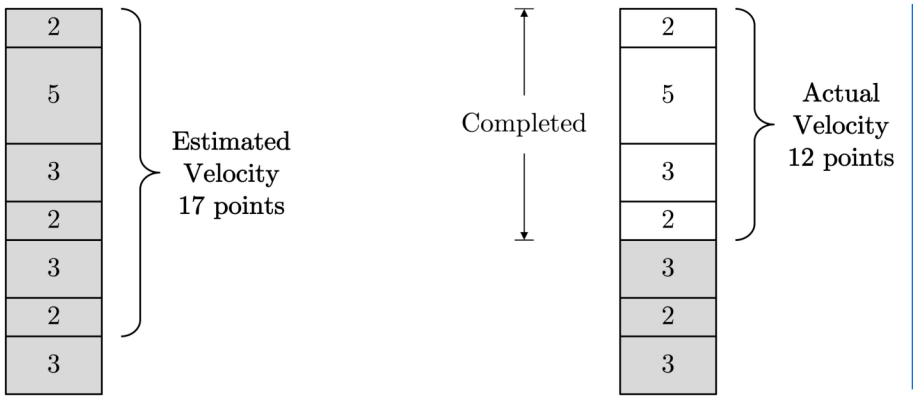


Figure 6.3: Estimated and actual velocity for an iteration. The team planned 17 points worth of work, but completed only 12 points worth.

6.2.3 Group Consensus

Under the right conditions, the consensus estimate from a group can be more accurate than individual expert judgment. The notion of group wisdom dates back to Aristotle:

> "the many, of whom each individual is an ordinary person, when they meet together may very likely be better than the few [experts] ... for some understand one part, and some another, and among them they understand the whole."[7]

Example 6.4. Consider the experience of the retailer Best Buy in estimating the number of gift cards they would sell around Christmas. The estimates from the internal experts proved to be 95% accurate. The consensus estimates from 100 randomly chosen employees proved to be 99.9% accurate. Both groups began with the actual sales number from the prior year.[8] □

Considerations for Structuring a Group.

Group estimation techniques differ in how they address two issues:

- *Avoiding cognitive bias.* Groups are subject to anchoring and groupthink, where some members are swayed and adapt to the group rather than giving their own opinion. Structured estimation techniques begin with each member independently supplying an initial estimate.
- *Converging on a consensus.* How are the various estimates by the group members combined into a consensus estimate? For example, the "consensus" may be a weighted average, the median, or a true consensus that emerges from structured group interaction.

	Round 1	2	3	4
Expert 1	1,000	525	332	349
2	200	256	300	292
3	300	250	250	276
4	150	184	200	206
5	125	158	166	167

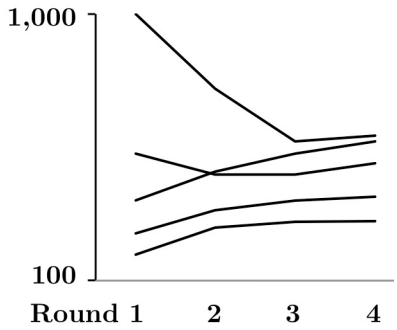


Figure 6.4: An application of the Delphi method with 5 experts. The same data appears in both tabular and graphical form.

The Original Delphi Method

The designers of the Delphi method were concerned that simply bringing a group together for a roundtable discussion

> "induces the hasty formulation of preconceived notions, an inclination to close one's mind to novel ideas, a tendency to defend a stand once taken or, alternatively and sometimes alternately, a predisposition to be swayed by persuasively stated opinions of others."[9]

With the original Delphi method, the group members were kept apart and anonymous. Instead of direct contact with each other, they were provided with feedback about where their estimate stood, relative to the others. Consensus was achieved by having several rounds of estimates and feedback.

Example 6.5. The data in Fig. 6.4 is for four rounds of forecasets by a group of five experts. The same data appears in tabular and graphical form.

T The first round forecasts range from a low of 125 to a high of 1,000. The median forecast for the first round is 200. In the second round, the range of forecasts narrows to 158-525. The ranges for the third and fourth rounds are close: 166-332 and 167-349, respectively.

Note that the group is converging on a range, not on a single forecast. With forecasts, it is not unusual for experts to have differences of opinion. □

in different forecasting exercises, the designers experimented with different forms of feedback between rounds; e.g., the median of the group estimates, a weighted average, the range (with outliers dropped), or some combination of these. In practice, a majority of applications of the Delphi method achieved group consensus. Even in cases where the rounds were stopped before the group's opinions had converged, the method was helpful in clarifying the issues and highlighting the sources of disagreement, leading to better decisions.[10]

repeat
    Participants anonymously submit individual estimates
    if the estimates have converged enough
        done
    else
        The moderator convenes a group meeting to discuss outliers

Figure 6.5: The Wideband Delphi Method.

Wideband Delphi and Planning Poker

For software projects, group discussion can lead to valuable insights that can be useful during design, coding, and testing. Any concerns that surface during the discussions can be recorded and addressed later. Any pitfalls can hopefully be avoided. Such insights are missed with the original Delphi method because it isolates group members to soften cognitive bias. Perhaps, the benefits of group discussion outweigh the risks of bias.

The *Wideband Delphi* method, outlined in Fig. 6.5, combines anonymous individual estimates with group discussion between rounds. The members have a chance to explain the reasoning behind their estimates. The method has been used successfully for both up-front and agile planning.[11]

A variant of the Wideband Delphi method, called Planning Poker has been applied to estimation for agile iteration planning. In *Planning Poker*, participants are given cards marked with Fibonacci story points $1, 2, 3, 5, 8, ...$ Each developer independently and privately picks a card, representing their estimate for the userstory under discussion. All developers then reveal their cards simultaneously. If the individual estimates are close to each other, consensus has been reached. More likely, there will be some high cards and some low cards, with the others being in the middle.

Do the developers with the high and low cards know something that the others don't? Or, are they missing something? The only way to find out is through group discussion. The process then repeats for further rounds of individual card-selection and group discussion until consensus is reached.

A moderator captures key comments from the group discussion of a story, so that the comments can be addressed during implementation and testing.[12]

6.2.4 Three-Point Estimation

A *Three-Point Estimate* is the weighted average

$$estimate \;=\; (b + 4m + w)/6$$

of three values:

b the best case estimate
m the most likely case estimate
w the worst case estimate

In practice, this weighted-average formula is roughly right, even the assumptions behind it do not fully hold. The formula is derived from on a theoretical model, based on the following assumptions: [13]

- Estimates b, m, and w for a work item can be made without considering the other work items in the project.
- Estimates b, m, and w are independent of schedule, budget, resource, or other project constraints. Furthermore, the estimates are assumed to be free of cognitive bias.
- Technically, the best case b is assumed to be the 95th percentile case and the worst case w is assumed to be the 5th percentile case.

6.3 Balancing Priorities

Raw user requirements correspond to wish lists from the various stakeholders. Requirements analysis organizes and prioritizes the wish lists and produces a single coherent set of product requirements.

6.3.1 Must-Should-Could-Won't (MoSCoW) Prioritization

The *MoSCoW* approach classifies requirements into the following four categories, listed in order of decreasing priority: must-have, should-have, could-have, and won't-have. The initial letters in the names of these categories appear capitalized in MoSCoW. Note that the order of the capitalized letters in MoSCoW matches the priority order of the categories. Here are some suggestions for classifying requirements:[14]

1. *Must-have* requirements are essential to the viability of a product and have to be designed and built in for the project to be successful.
2. *Should-have* requirements are important, but are lower in priority than must-have requirements. For example, there may be alternative ways of meeting the needs addressed by should-have requirements.
3. *Could-have* requirements are nice to have. They contribute to the overall user experience with a product; however, user needs can be met without them.
4. *Won't-have* requirements can be dropped.

The MoSCoW prioritization approach relies on a shared understanding between users and developers on how to classify into the above categories. What if

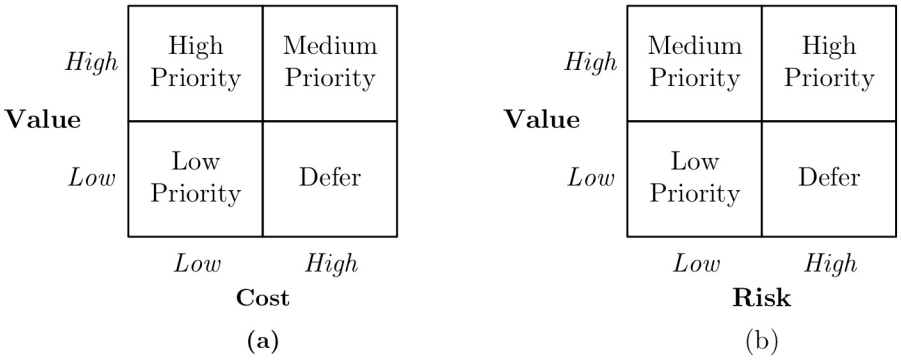


Figure 6.6: Prioritize first by value and cost and then by value and risk.

users undervalue some requirements that the development team considers essential to the success of the product? For example, customers often take for granted quality attributes such as reliability and performance. Such requirements can be classified as should-have, if they do not make it into the must-have category.[15] (See also Section 6.4, where the classification of requirements is based on both customer satisfiers and dissatisfiers.)

MoSCoW prioritization can be combined with other prioritization methods. For example, we could start by assigning a high priority to must-have requirements and then apply another method to the rest.

6.3.2 Balancing Value and Cost

Rough cost-value prioritization of user requirements can be done by classifying them as high or low in value and high or low in cost; see Fig. 6.6(a). Here, value is value for users; development effort can serve as a proxy for cost. Users can classify requirements as high or low value, based on the The cost-value analysis will be refined below to balance value, cost, and risk.

For maximizing value while minimizing cost, prioritize the requirements as follows:

1. *High priority* for high-value low-cost requirements.
2. *Medium priority* for high-value high-cost requirements.
3. *Low priority* for low-value low-cost requirements.
4. The low-value high-cost requirements can potentially be deferred or dropped entirely.

6.3.3 Balancing Value, Cost, and Risk

Let us now refine the above value and cost prioritization to minimize risk as well as cost. The following is a two-step approach:[16]

a) Prioritize first by value and cost, as summarized in Fig. 6.6(a). The four categories in the figure are high-value low-cost; high-value high-cost; low-value low-cost; and low-value high-cost.

b) Within each category from Step(a), prioritize the requirements as follows (see Fig. 6.6(b)):

   1. *high priority* for high-value high-risk requirements;
   2. *medium priority* for high-value low-risk requirements; and
   3. *low priority* for low-value low-risk requirements.
   4. The low-value high-risk requirements can potentially be deferred or dropped.

   In Step(b), it may seem counter-intuitive to rank high-value high-risk requirements ahead of high-value low-risk. All high-value requirements must presumably be implemented sooner or later. Tackling high-risk requirements early allows the development team more time to address the risks. For example, if risk resolution involves discussion and renegotiation with customers, it is better to surface issues earlier rather than later.

6.4 Customer Satisfiers and Dissatisfiers

What makes one product or feature desirable and another taken for granted until it malfunctions? An analysis of satisfiers and dissatisfiers is helpful for answering this question. Such analysis can unearth latent needs.

6.4.1 Background: Job Satisfiers and Dissatisfiers

The factors that lead to job satisfaction are different from the factors that lead to job dissatisfaction, as Frederick Herzberg and his colleagues discovered in the 1950s:[17]

- *Job satisfaction* is tied to the work, to what people do: "job content, achievement on a task, recognition for task achievement, the nature of the task, responsibility for a task and professional advancement."
- *Job dissatisfaction* is tied to "an entirely different set of factors." It is tied to the situation in which the work is done: supervision, interpersonal relationships, working conditions, salary.

Improving working conditions alone reduces dissatisfaction, but it does not increase job satisfaction because the nature of the work does not change. Similarly, improving the nature of the work alone does not reduce job dissatisfaction, because working conditions and salary do not change.

6.4.2 Kano Analysis

Noriaki Kano and his colleagues carried the distinction between job satisfiers and dissatisfiers over to customer satisfiers and dissatisfiers.[18] *Kano analysis* assesses the significance of product features by considering the effect on customer satisfaction of (a) building a feature and (b) not building the feature. Kano analysis has been applied to user stories and work items in software development.[19]

Paired Questions

Kano et al. used questionnaires with paired positive and negative questions about product features. The paired questions were of the form:

- If the product *has* this feature ...
- If the product *does not have* this feature ...

For example, consider the following positive and negative forms of a question:

- If this product *can* be recycled ...
- If this product *cannot* be recycled ...

With each form, positive, and negative, they offered five options:

a) I'd like it

b) I'd expect it

c) I'm neutral

d) I can accept it

e) I'd dislike it

6.4.3 Classification of Features

The results from paired positive and negative questions can be classified using a grid such as the one in Fig. 6.7. For simplicity, the classification in the figure is based on the following three, instead of five, options:

a) I'd be satisfied

b) I'm neutral

c) I'd be dissatisfied

The rows in the nine-box grid correspond to the cases where a feature is built. The columns are for the cases where the feature is not built. Note the ordering of the rows and columns: with rows, satisfaction decreases from top to bottom; with columns, satisfaction decreases from left to right.

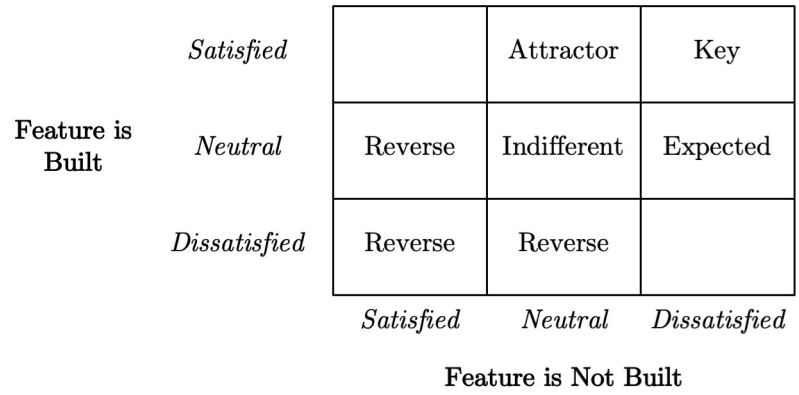


Figure 6.7: Classification of features.

Key Features

The top-right box in Fig. 6.7 is for the case where customers are satisfied if the feature is built and would be dissatisfied if it is not built. With key features, the more the better, subject to the project's schedule and budget constraints. For example, with time-boxed iterations, a constraint would be the number of features that the team can build in an iteration.

Reverse Features

The shaded box to the bottom left is for the case where customers would be dissatisfied if the feature is built and satisfied if it is not built. The two adjacent shaded boxes are also marked Reverse. With reverse features, the fewer that are built the better.

Example 6.6. Features that customers do not want are examples of reverse features. Clippy, an "intelligent" Microsoft Office assistant would pop up when least expected to cheerily ask something like, "I see that you're writing a letter. Would you like help?" The pop-up interrupted the flow of work, so it caused dissatisfaction.

Clippy was so unpopular that an anti-Clippy web site got about 22 million hits in a few months after the launch of the web site.[20] □

The following is a modest example of a reverse feature.

Example 6.7. Users complain when there are changes in user interfaces. Out of the pages and pages of changes in the 4.** series of versions of TeXShop, the change that generated the most email was a formatting change in the default indentation of source text. Richard Cox, the coordinator and original author of TeXShop writes,

> "I learned an important lesson. When a new feature is introduced which changes the appearance of the source code, the default value should make no change."

The change in formatting behavior in 4.08 was undone in 4.13 by resetting the default value.[21]

Disclosure: this book was created using TexShop. □

Attractor Features

The box in the middle of the top row in Fig. 6.7 is for the case where customers would be satisfied if the feature is built and neutral if it is not. This box represents features that can differentiate a product from its competition. Features that address latent needs are likely to show up as attractors.

Example 6.8. Mobile phones with web and email access were a novelty around the year 2000. Kano analysis in Japan revealed that young people, including students, were very enthusiastic about the inclusion of the new features, but were neutral about their exclusion. These features were therefore attractors.

Indeed, such phones with web and email access rapidly gained popularity. □

Expected Features

The box in the middle of the right column in Fig. 6.7 is for the case where customers would be neutral if the feature is built and dissatisfied if it is not built. Features customers take for granted fit in this box. For example, if a feature does not meet the desired performance threshold—that is, if performance is not built in—customers would be dissatisfied. But if performance is built in, then customers may not even notice it.

Indifferent Features

The middle box in Fig. 6.7 is for the case where customers would be neither satisfied nor dissatisfied if the feature were built. Features that customers consider unimportant would also fit in this box.

6.4.4 Degrees of Sufficiency

The classification of features in Fig. 6.7 is based on a binary choice: either the feature is built or it is not built. Response time and capacity are examples of system attributes that are not binary. They potentially improve along a sliding scale.

The conceptual diagram in Fig. 6.8 illustrates features that have degrees or levels of sufficiency. The horizontal axis represents degrees of sufficiency, increasing from *Insufficient* on the left to *Sufficient* on the right. The vertical axis

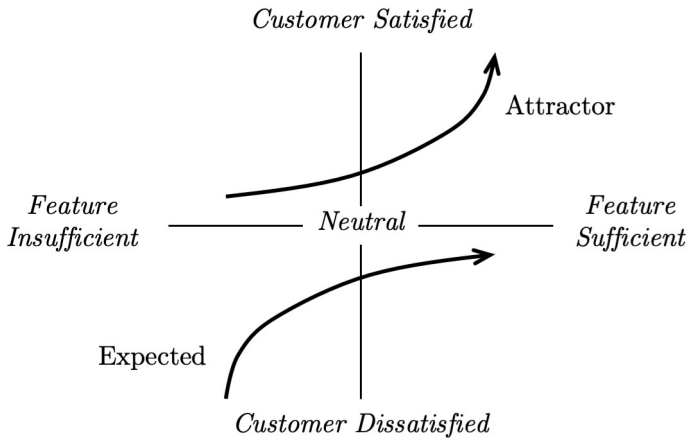


Figure 6.8: Conceptual diagram illustrating the increase in satisfaction with the increase in sufficiency for attractors and expected features.

represents degrees of satisfaction, from *Dissatisfied* at the bottom to *Satisfied* at the top.

The top curve is for an attractor. With an attractor, customers are satisfied if the feature is built, and are neutral if it is not built. In Fig. 6.8 the Attractor curve rises from roughly *Neutral* to *Satisfied* as the degree of the feature increases from insufficient to sufficient.

The bottom curve is for an expected feature. With an expected feature, customers are dissatisfied if the feature is not built, but are neutral if it is built. The curve marked Expected rises from *Dissatisfied* to roughly *Neutral* as the degree of the feature increases from insufficient to sufficient.

The curves for key, indifferent, and reverse features are not shown in Fig. 6.8. The curve for a key feature would increase diagonally up and to the right; the curve for an indifferent feature would stay at *Neutral*; and the curve for a reverse feature would decrease diagonally down and to the right.

6.4.5 Life Cycles of Attractiveness

The attractiveness of features can vary over time. Consider again web and email access for mobile phones. When these features were first introduced in Japan in the late 1990s, young people found them to be attractive, but middle-aged people were indifferent—they had not experienced the need for the features. In the early days of the World Wide Web, people outside the technology community may have heard the term, but few felt the need for it. Today, any new smartphone is expected to provide web and email access; these features have become expected features.

For features, a possible progression is as follows:

1. *Indifferent.* When an entirely new feature is introduced, people may be indifferent because they do not understand its significance.
2. *Attractor.* As awareness of the new feature grows, it becomes an attractor, a delight to have, but people who do not have it are neutral.
3. *Key.* As usage of the feature grows, people come to rely upon it and are dissatisfied if they do not have it.
4. *Expected.* Eventually, as adoption of the feature grows, it is taken for granted if it is present and a dissatisfier if it is not present.

The above is not the only progression. A fashionable feature may rise from indifferent to key and fade back to indifferent as it goes out of fashion.

6.5 Goal Analysis

The goal analysis techniques in this section are broadly applicable to any goal. They appear to have been reinvented in three different contexts: requirements, measurement, and security.[22] The central idea in goal analysis is goal refinement.

Initial or top-level goals tend to be general statements of a user want. With requirements, the easily stated top-level goals need to be refined into specific work items for developers to build. Refinement is done by asking clarifying questions. From Section 4.3.2, *Why* questions explore the context and motivation for a goal; *How* questions explore subgoals or tasks that contribute to the success of a goal.

Example 6.9. A company wants an app for its service technicians, who go to customer sites to maintain equipment. The top-level goal is

Maintain equipment at customer sites.

How do they do that? The top-level goal can be refined into three subgoals, all of which must be done:

1. Travel to a customer site. **and**
2. Provide service. **and**
3. Update service log.

Let us continue refinement for one more ste. How do the technicians travel to customer sites? This time, there are two possible subgoals, either of which will work:

1. Get an assigned route. **or**
2. Provide service. **and**
3. Update service log.

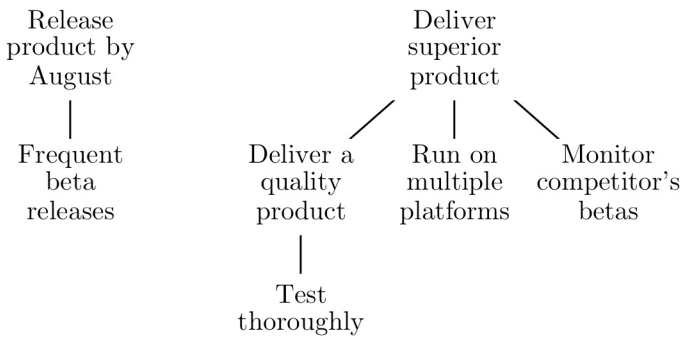


Figure 6.9: A goal hierarchy.

Refinement can continue with the current set of subgoals, until we get to refined goals that developers know how to implement. At this stage, the developers might have questions about how the app can help. Maps showing locations? Who assigns routes and what is their interface for doing so? Do technicians update the service log by taking notes on a mobile device and how does that device synch with the company's database? □

6.5.1 Goal Hierarchies

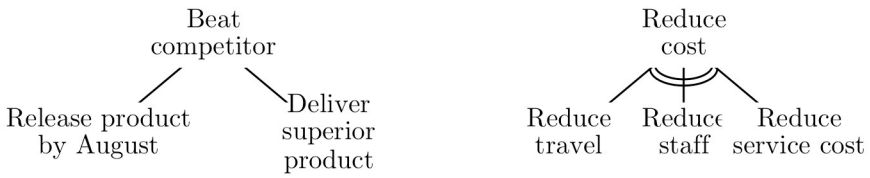
As the number of goals increases, it is helpful to organize them into a hierarchy that links goals and their subgoals. The hierarchy in Fig. 6.9 is a tree, but, in general, a subgoal may be shared by more than one higher goal.

A *goal hierarchy* has nodes representing goals and edges representing relationships between goals. In diagrams of hierarchies, goals appear above their subgoals. The nodes in a hierarchy are of two kinds:

- An *and* node represents a goal G with subgoals $G_1, G_2, ..., G_k$, where every one of $G_1, G_2, ..., G_k$ must be satisfied for G to be satisfied. For example, the goal in Fig. 6.10(a) of beating a competitor is satisfied only if both of the following are satisfied: release the product by August; and deliver a superior product.

- An *or* node represents a goal G with subgoals $G_1, G_2, ..., G_k$, where G can be satisfied by satisfying any one of $G_1, G_2, ..., G_k$. For example, the goal in Fig. 6.10(b) of reducing costs can be satisfied by doing any one or any combination of the following: reduce travel costs; reduce staff; or reduce the cost of service after delivery.

Or nodes will be identified by a double arc connecting the edges between the goal node and the nodes for the subgoals.



(a) *And* node (b) *Or* node

Figure 6.10: Examples of *and* and *or* nodes in a goal hierarchy.

Example 6.10. The goal hierarchy in Fig. 6.9 has two initial goals. In general, a project can have multiple initial goals; e.g., one for each use case.

In Fig. 6.9, there is only one goal with more than one subgoal: "Deliver superior product." This goal has an *and* node, so its three subgoals must all be satisfied for this goal to be satisfied.

This example is motivated by the browser wars between Netscape and Microsoft; see Section 2.2.2. Netscape's strategy was to deliver a browser that ran on multiple operating systems; Microsoft's browser ran only on Windows, at the time. Netscape also closely monitored the beta releases from Microsoft to ensure that their browser would be competitive. □

6.5.2 Contributing and Conflicting Goals

A goal such as "Deliver a quality product" contributes strongly to the higher goal, "Deliver a superior product." Monitoring a competitor's beta releases does contribute to delivering a superior product, but not as strongly. Meanwhile, there can be conflicting goals: "Improve quality" potentially conflicts with "Reduce staff." The prioritization techniques in Section 6.3 can help resolve goal conflicts.

Goal G_1 *conflicts* with goal G_2 if satisfaction of G_1 hinders the satisfaction of G_2. For example, "Add testers" conflicts with "Reduce staff." If we add testers, we are adding, not reducing staff. Conversely, if we reduce staff, we are potentially hindering the addition of testers—potentially, since the reductions could be elsewhere.

Goal G_1 *contributes* to goal G_2 if satisfaction of G_1 aids the satisfaction of G_2. For example, "Add testers" contributes to "Improve product quality."

The conflicts and contributes relationships will be represented by directed graphs with goals for nodes. An edge is marked `++` for a strong contribution, `+` for a weak contribution, `--` for a strong conflict, and `-` for a weak conflict. For clarity, edges representing weak contributions or weak conflicts will be dashed.

Example 6.11. The graph in Fig. 6.11 shows the contributes and conflicts relations for the goals in Example 6.10.

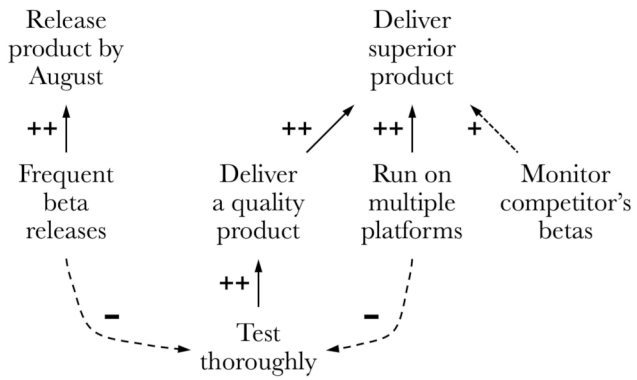


Figure 6.11: Examples of contributing and conflicting goals.

Since subgoals contribute to higher goals, there are directed edges from subgoals to higher goals. As an example, the goal at the top right, "Deliver superior product," has three edges coming into it. Two of the edges are for strong contributions, so they are shown as solid edges. The third edge is for a weak contribution from "Monitor competitor's betas;" it is dashed.

There are two edges for conflicts, both to "Test throughly." At Netscape, the number of testers was fixed, so more platforms (operating systems) to test meant more work for the same testers. Furthermore, every beta release had to be fully tested, so more frequent beta also meant additional work for the same testers. Hence the conflicts edges from "Frequent beta releases" and "Run on multiple platforms" to "Test thoroughly." □

6.5.3 When to Stop Goal Elaboration

A goal hierarchy grows downwards as subgoals are identified and nodes are added for them. This process of growing the hierarchy is called *goal refinement* or *goal elaboration*. Goal refinement can stop when the subgoals become SMART, where SMART stands for specific, measurable, achievable, relevant , and time-bound. Weith requirements, SMART goals can be implemented. More generally, actions can be defined from SMALRT goals.

Example 6.12. Consider the following refinement of "Run on multiple platforms:"

> *How many platforms does the browser need to run on?*
> Three: Windows, Mac OS, and Unix.

No further refinement is needed, since the last goal of running on the three platforms tells the developers what they need to know to implement this requirement. □

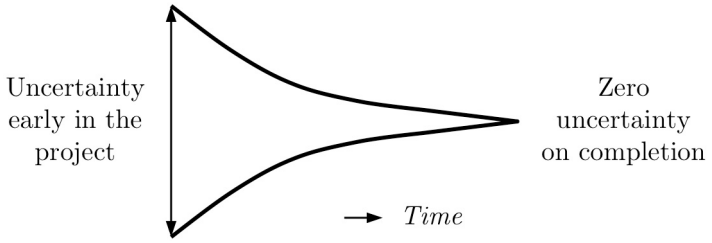


Figure 6.12: The Cone of Uncertainty is a conceptual diagram that illustrates the intuition that estimation uncertainty decreases as a project progresses.

6.6 Plan-Driven Estimation Models

A software project has a cost and schedule overrun if the initial development-effort estimates are too low. The state of the art of effort estimation can be summarized as follows:[23]

- Most estimation techniques ultimately rely to a lesser or greater extent on expert judgment.
- Historical data about similar past projects is a good predictor for current projects.
- Estimation accuracy can be improved by combining independent estimates from a group of experts.
- Simple models tailored to the local work environment can be at least, if not more. accurate than advanced statistical models.
- There is no one best estimation model or method.

Estimation Uncertainty

Estimation uncertainty decreases as a project progresses and more information becomes available. This intuition motivates the conceptual diagram in Fig. 6.12. Ket *estimation uncertainty* be the range between a pair of upper and lower bounds on estimates. The solid lines represent upper and lower bounds, so uncertainty at a given point in the project is the distance between the sold lines.

The diagram in Fig. 6.12 has been dubbed the *Cone of Uncertainty*, after the shape enclosed by the solid lines. Similar diagrams were in use in the 1950s for cost estimation for chemical manufacturing.[24]

6.6.1 How are Size and Effort are Related?

The "cost" of a work item can be represented by either the estimated program size or the estimated development effort for the item. Size and effort are related,

but they are not the same. As size increases, effort increases, but by how much?

The challenge is that, for programs of the same size, development effort can vary widely, depending on factors such as team productivity, problem domain, and system architecture. Team productivity can vary by an order of magnitude.[25] Critical applications require more effort than casual ones. Effort increases gradually with a loosely coupled architecture; it rises sharply with tight coupling.

Project managers can compensate for some of these factors, further complicating the relationship between size and effort. Experienced project managers can address productivity variations when they form teams; say, by pairing a novice developer with someone more skilled. Estimation can be improved by relying on past data from the same problem domain; for example, smartphone apps can be compared with smartphone apps, embedded systems with embedded systems, and so on. Design guidelines and reviews can lead to cleaner architectures, where effort scales gracefully with size.

The relationship between size and effort is therefore context dependent. Within a given context, however, historical data can be used to make helpful predictions.

For large projects or with longer planning horizons, it is better to work with size. Iteration planning, with its short 1-4 week planning horizons, is often based on effort estimates. The discussion in this section is in terms of effort—the same estimation techniques work for both size and effort.

Estimating Effort from Size

Consider the problem of estimating development effort E from program size S. In other words, determine a function f such that

$$E = f(S)$$

If f is a linear function, then estimated effort E is proportional to size S: if size doubles, then effort doubles.

Effort does indeed increase with size, but not necessarily linearly. Does it grow faster than size, as in the upper curve in Fig. 6.13? Or does it grow slower than size, as in the lower curve? The dashed line corresponds to effort growing linearly with size.

The three curves in Fig. 6.13 were obtained by picking suitable values for the constants a and b in the equation

$$E = aS^b \tag{6.1}$$

The curves in Fig. 6.13 correspond to the following cases:

- *Case* $b = 1$ (dashed line). The function in Equation (6.1) is then linear.
- *Case* $b < 1$ (lower curve). This case corresponds to there being economies of scale; for example, if the team becomes more productive as the project proceeds, or if code from a smaller project can be reused for a larger project.

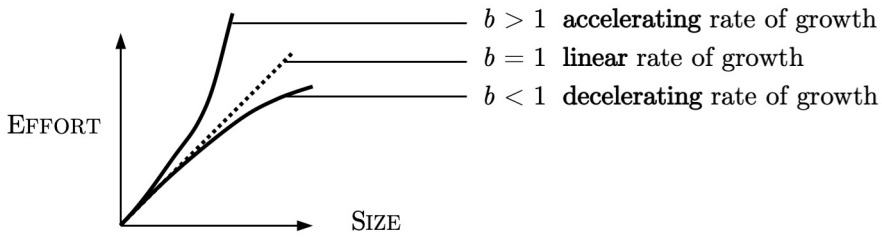


Figure 6.13: How does effort grow with size?

- *Case* $b > 1$ (upper curve). The more usual case is when $b > 1$ and larger projects become increasingly harder, either because of the increased need for team communication or because of increased interaction between modules as size increases. In other words, the rate of growth of effort accelerates with size.

6.6.2 The Cocomo Family of Estimation Models

Equation (6.1), expressing effort as a function of size is from a model called Cocomo-81. The name Cocomo comes from Constructive Cost Model. The basic Cocomo model, introduced in 1981, is called Cocomo-81 to distinguish it from later models in the Cocomo suite.[26]

For a given project, the constants a and b in Equation (6.1) are estimated from historical data about similar projects. IBM data from waterfall projects in the 1970s fits the following (E is effort in staff-months and S is in thousands of lines of code):[27]

$$E = 5.2\, S^{0.91} \tag{6.2}$$

Meanwhile, TRW data from waterfall projects fits the following (the three equations are for three classes of systems):[28]

$$\begin{aligned} E &= 2.4\, S^{1.05} && \text{Basic Systems} \\ E &= 3.0\, S^{1.12} && \text{Intermediate} \\ E &= 3.6\, S^{1.20} && \text{Embedded Systems} \end{aligned} \tag{6.3}$$

The constants in (6.3) can be adjusted to account for factors such as task complexity and team productivity. For example, for a complex task, the estimated effort might be increased by 25% (the actual percentage depends on historical data about similar projects by the same team). Such adjustments can be handled by picking suitable values for the constants a and b in the general equation (6.1).

Cocomo II: A Major Redesign

Cocomo-81 was applied successfully to waterfall projects in the 1980s, but it lost its predictive value as software development processes changed. As processes changed, the data relating effort and size changed and the earlier statistical models no longer fit.

A major redesign in the late 1990s resulted in Cocomo II. The redesign accounted for various project parameters, such as the desired reliability and the use of tools and platforms. With Cocomo II and its many variants, the relationship between effort E and size S is given by the general form

$$E = aS^b + c \tag{6.4}$$

Constants a, b, and c are based on past data about similar projects. Factors like team productivity, problem complexity, desired reliability, and tool usage are built into the choice of constants a, b, and c.

New estimation models continue to be explored. As software engineering evolves, the existing models lose their predictive power.[29] Existing models are designed to fit historical data, and the purpose of advances in software development is to improve upon (disrupt) the historical relationship between development effort, program size, and required functionality. Az

6.7 Conclusion

Requirements analysis is at the interface between the user domain and the develoer doamain. It therefore spans a range of activities, from clarifying what users want to planning what to build. Agile planning revisits the activities with every iteration. With plan-driven methods, the activities dig deeper, since plans are set before design and coding. Agile projects can begin with rough estimates since planning is incremental. The rough estimates potentially get better as the iterations progress and the team gains more experience.

The logical progression of activities during requirements analysis is as follows. Projects range in size and complexity, so for some projects the items in the progression will be check-list items; for others they may be major activities. In practice, the items may be revisited or done iteratively.

- Confirm the list of major stakeholders and their goals.
- Identify the significant quality attributes.e.
- Clarify and refine the top-level user goals.
- Select prioritization criteria.
- Classify the requirements or work items into prioritized categories.

Stakeholders and Goals

A prioritized list of stakeholders and their goals provides an overview of a system. The list of primary actors and user goals in a collection of use cases provides such an overview; see Chapter 5. Users may overlook quality attributes (nonfunctional requirements). The goals need to reflect both functionality and attributes such as usability, scale, availability, and a host of other " 'ilities." Measures for the attributes can be identified during goal analysis.

The initial "top-level" user goals can be soft; e.g., "I want a friendly app for children admitted to this hospital." Goal refinement, Section 6.5, refines such soft goals into specific measurable subgoals/work items; e.g., provide reliable medical information, or enable children o select music during a test. Refinement is done by asking clarifying questioins, such as the ones in Section 4.3.2. *How-Much* and *How-Many* questions can help quantify goals and quality attributes.

Classify and Prioritize Requirements

For many projects, prioritization can be based on cost-benefit criteria: a user-determined benefit and a developer-estimated cost, in terms of time and effort. For other projects, prioritization criteria may need to be defined; e.g., if there are goal conflicts or if there are additional requirements beyond the functionality. Prioritization criteria emerge from user-developer discussions.

Estimates are needed for prioritization. With agile projects, rough estimates suffice, since iterations are short, 1-4 weeks, and work items are selected per iteration. Section 6.2 considers classification of work items is done by assigning story points: 1 point for the simplest story, with more points for increasingly complex stories. The points can be on either on a 1-2-3 scale or on a Fibonacci scale.

Estimates can be improved by combining independent estimates from a group. Wideband Delphi and Planning Poker are group estimation techniques. With them, there are rounds of independent estimates, followed by group discussion to comopare notes. The rounds continue until the group converges on a value or a range; ; see Section 6.2.3. The techniques manage, but do not eliminate, cognitive bias; see Section 6.1.2 for cognitive bias and anchoring.

A high-medium-low classification may suffice for prioritization of requirements . Section 6.3 begins with MoSCoW prioritization, which has four categories: must have, should have, could have, and won't have, in order of decreasing priority. Together, users and developers classify requirements into these categories, using criteria that work for them. Depending on the project, quality attributes may fit into the must-have or should-have category.

Section 6.3 then considers classification based on two factors: high or low on value; and high or low on value. We therefore have four buckets, in order of decreasing priority: high-value low-cost; high-value high-cost; low-value low-cost; and low-value high-cost. The lowest priority requirements can be dropped. A third factor, risk, can be included by prioritizing first by value and cost and

then by value and risk. Subdivide each value-cost bucket as follows: high-value high-risk; high-value low-risk; low-value low-risk; and low-value high-risk. Again, the lowest priority requirements may be dropped. Note that high-value high-risk items get the highest priority, to allow more time to address the risk.

Classification by Satisfiers and Dissatisfiers

Kano analysis, Section 6.4 classifies requirements/features along two dimensions:

- customer reaction if the feature was built;
- customer reaction if feature were not built.

Customer reaction has three possible values: satisfied, neutral, and dissatisfied. Of course, the highest priority are Key features that customers want (satisfied if built) and would be unhappy without (dissatisfied if not built). Similarly, the lowest priority is for Reverse features that customers don't want (dissatisfied if built) and would be happier without (satisfied if not built). Annoying popups are an example of a Reverse feature.

The interesting categories are called Attractors and Expected features. The Attractor category is where nwe might find novel features, whose value is not yet recognized by users. Attractors are features where customers would be satisfied if built and neutral if not built.

Expected features include ones that customers might take for granted; e.g., quality attributes like performance and reliability. Reliability goes unnoticed until a produt fails. Expected features are ones where customers would be dissatisfied if not built and neutral if built.

The section also considers the case where there are degrees of sufficiency. That is, instead of the binary choice, built or not built, a feature goes on a sliding scale from insufficient to sufficient.

Estimation Models for Plan-Driven Projects

With plan-driven processes, key project decisions about schedules, budgets, team formation, and work assignments are made early, before design and coding. Early estimates can be improved by using a combination of expert judgment and formal models. Expert judgment can be tapped using Wideband Delphi; see Section **??**. Meanwhile, Section 6.6 introduces the Cocomo family of models. The tradeoffs between them are as follows:[30]

- Expert judgment can be more effective and accurate, but it is time consuming, so it cannot be used widely.
- Statistical models may be less effective, but they can be run automatically to complement expert judgment.

Exercises for Chapter 6

Exercise 6.1. Come up with your own examples of products that are

a) Useful, but neither usable nor desirable

b) Usable, but neither useful nor desirable

c) Desirable, but neither useful nor usable

d) Useful and usable, but not desirable

e) Useful and desirable, but not usable

f) Usable and desirable, but not useful

g) Useful, usable, and desirable

Your examples need not relate to software.

Exercise 6.2. Are the following statements generally true or generally false? Briefly explain your answers.

a) The expected overall planning effort is less with up-front planning than with agile planning.

b) The Iron Triangle illustrates connections between time, cost, and scope.

c) The Agile Iron Triangle fixes time and scope and lets costs vary.

d) Anchoring is the human tendency to stick to a position, once taken.

e) The shorter the planning horizon, the lower the uncertainty.

f) Anchoring reduces uncertainty during planning.

g) Planning Poker involves successive rounds of individual estimation and group discussion.

h) Wideband Delphi involves successive rounds of individual estimation and group discussion.

i) Three-point estimation takes the average of the best, the most likely, and the worst case estimates.

j) With traditional projects, development effort always grows exponentially afster as program size increases.

Exercise 6.3. Use the nine-box grid in Fig. 6.14 to classify features during Kano analysis. Give a one-line explanation for why a given class of features belongs in one of the boxes in the grid. (Note that the ordering of rows and columns in Fig. 6.14 is different from the ordering in Fig. 6.7.)

Exercise 6.4. For each of the following categories, give an example of a feature related to video conferencing. Be specific and provide realistic examples.

a) Key

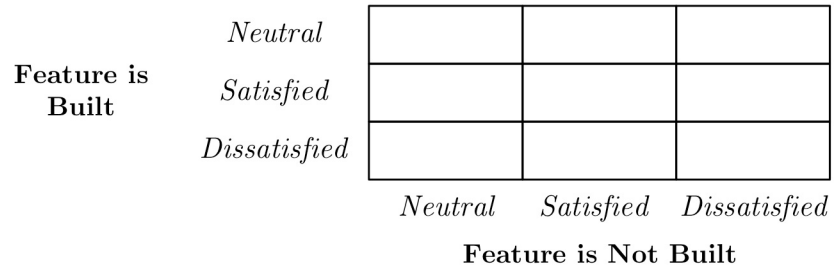


Figure 6.14: Classification of features during Kano analysis. The rows and columns have been scrambled, relative to Fig. 6.7.

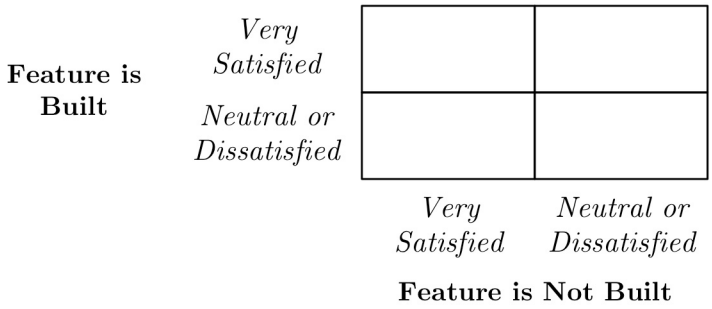


Figure 6.15: Classification of features during Kano Analysis.

b) Attractor

c) Must Have

d) Indifferent

e) Reverse

Exercise 6.5. Consider Kano analysis using the four-box grid in Fig. 6.15, instead of the nine-box grid in Fig. 6.7.

a) How would you classify features using the four-box grid? Explain your answer.

b) For each of the four boxes, give an example of a specific feature that belongs in that box. Why does it belong in the box?

Exercise 6.6. The conceptual diagram in Fig. 6.8 illustrates the increase in customer satisfaction for Attractors and Must-Have features as the sufficiency

of a feature increases. Draw the corresponding curves for Key, Reverse, and Indifferent features.

Exercise 6.7. Magne Jørgensen and Barry Boehm engaged in a friendly debate about the relative merits of expert judgment and formal models (e.g., Cocomo) for estimating development effort. Read their debate [**??**] and summarize their arguments pro and con for:

Summarize their arguments, pro and con, for

a) expert judgment.

b) formal models.

Based on their arguments, when and under what conditions would you recommend estimation methods that rely on

c) expert judgment.

d) formal models.

Chapter 7

Software Architecture

"A map is not the territory."

— *The distinction is attributed to Eric Temple Bell.*[1] *Similarly, a view is not the architecture. A view is a description of an architecture. It is not the intangible thing that is being described.*

7.1 Introduction

The intuitive the is simply stated: the architecture of a software system describes (a) the parts of the system, and (b) how the parts work together to form the whole.

Why then has it been so hard to converge on a precise definition of software architecture? The Software Engineering Institute (SEI) has compiled a list of a dozen definitions from "some of the more prominent or influential books and papers on architecture."[2] Previously, the SEI collected over two hundred definitions from visitors to its website.

The transition from simple statement to a precise definition is hard because the the simple statement is about a "view" of the architecture, not the overall architecture itself. Below, we illustrate the concept of a view by considering the architecture of physical buildings. As we shall see in Section 7.3, there can be multiple views of an architecture. Each view serves a different purpose. This chapter is largely about views and how to describe them. Frequently occurring architectures are discussed in Chapter 8.

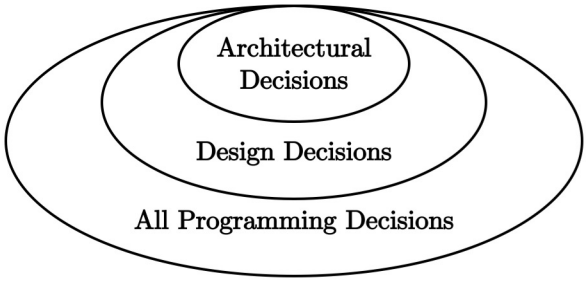


Figure 7.1: Architecture is used to reason about the overall properties of a system. Design is used to reason about both external and internal properties.

Design Includes Architecture

Architecture is a subset of design, although the boundary between them is fuzzy. Based on their experience at the SEI, Rick Kazman and Amnon Eden observe:

> "In practice, the terms 'architecture,' 'design,' and 'implementation' appear to connote varying degrees of abstraction in the continuum between complete details ('implementation'), few details ('design'), and the highest form of abstraction ('architecture')."[3]

Another characterization is that architectural decisions are "early," compared to overall design decisions. Early decisions can strongly influence a project, but, not all architectural decisions are early, nor are all early decisions architectural.

For our purposes, architectural decisions are a subset of design decisions, and design decisions are a subset of overall programming decisions, as shown in Fig.. 7.1.[4] The distinction between architecture and design is as follows:

- Architecture is used to understand and reason about the external properties of a system and its parts, such properties as the overall behavior, structure, quality attributes, and geographic deployment.
- Design is used to understand and reason about both the external and internal properties of the system and its parts. Design therefore includes architecture.

7.2 Lessons from Classical Architecture

Classical architecture—the architecture of physical buildings—has some lessons to offer for software architecture. Chief among them is the concept of patterns, the subject of Chapter 8. The primary purpose of this section is to use buildings to illustrate the concept of architectural views. The section also includes some thoughts on desirable properties for the architecture of a software system.

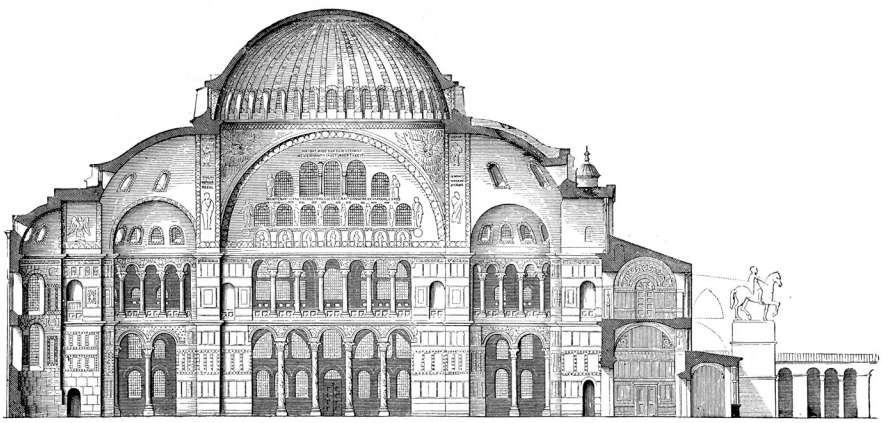


Figure 7.2: Cross section of the Hagia Sophia, which was built roughly 1500 years ago.

7.2.1 Architectural Views of Buildings

Unlike software physical buildings can be seen and touched, so they are convenient for illustrating the concept of architectural views.

Example 7.1. Inaugurated in 537 AD, the Hagia Sophia in Istanbul was perhaps the greatest cathedral in the world for a thousand years. It was converted to a mosque in 1453. Its architecture was subsequently replicated in mosques throughout the Ottoman empire.

The architectural view of the Hagia Sophia in Fig. 7.2 highlights the proportions and interior spaces. The vast main hall is capped by a dome that rises 182 feet from the floor. Clustered around the round dome are half-domes that make room for galleries and side-halls.For information about the Hagia Sophia, see the Wikipedia article on the subject. [5] □

A view focuses on some aspect of an architecture; the view is not meant to be a complete characterization of the architecture. For example, the cross section in Fig. 7.2 shows interior spaces, but it does not provide a floor plan; it does not address how the interior is lit; it does not show how the weight of the round dome is supported; and so on. Other views are needed for other purposes. As we shall see in Section 7.3, descriptions of software architecture include multiple views.

Example 7.2. The view in this example focuses on a key architectural problem: support for the weight of a round dome. The weight of the dome is distributed to the four corners of a square base by an architectural element called a *pendentive*; see Fig. 7.3. A pendentive has a round opening at the top and four legs. So,

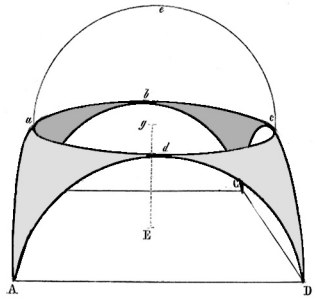


Figure 7.3: The geometry of a pendentive (shaded).

the core of the building can be constructed by building pillars at the corners of a square base; putting the legs of a pendentive on the pillars; and then putting a dome atop the pendentive.[6]

The dome needs to be shaped so its weight bears down on the pendentive. The original dome collapsed during the earthquake of 558. The rebuilt dome is 30 feet higher to better distribute its weight to the supports below. □

7.2.2 What is a Good Architecture?

Asking what makes an architecture good is akin to asking what makes a piece of software good. There are no hard and fast rules; only guidelines. At the very least, the architecture must do the job it is designed for; that is, it must fulfill its intended purpose. But, there are other properties we might want in an architecture such as maintainability, extensibility, and elegance.

These three virtues date back to the work of the Roman architect Vitruvius.

For buildings, the Roman architect, Vitruvius, laid out three virtues in the first century BCE: strength, utility, and beauty. These three virtues are still applied to buildings. The following properties for software architecture are inspired by Vitruvius's work:[7]

- *Functional.* Will the system meet user requirements? Will it be fit for purpose? Will it be worth the cost?
- *Robustness.* Will the architecture result in a system that is well built? Will the system have the desired quality attributes? Is the technology approach appropriate? Robustness might also include software quality and defensive implementation practices.
- *Elegance.* Is the architecture easy to understand and modify? Will it lead to a clean implementation? Can it be built iteratively and incrementally? Does the design have conceptual integrity; that is, is it consistent?

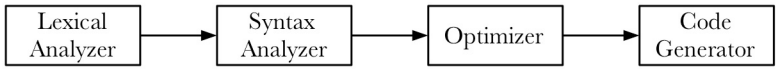


Figure 7.4: A view of the architecture of a compiler that depicts the flow of data between the parts of the compiler. Boxes represent parts, arrows represent data flows.

7.3 Architectural Views

Extrapolating from the discussion of architectural views of buildings in Section 7.1, the term architecture may refer to different views for different stakeholders of a software system. For users, architecture can relate to a logical view, to how the system will support the desired functionality and attributes. (Recall that quality attributes are properties such as scale, performance, and reliability.) For developers, architecture can mean the decomposition of the source code into modules and interfaces. For operations staff, architecture can mean the deployment of software onto physical servers. All of these are valid views. By itself, none of them is a complete characterization of the software architecture of the system.

Example 7.3. Architecture is like the elephant in the proverbial story about the blind philosophers who come upon an elephant for the first time. Each touches a different part of the elephant and comes up with a different view of the animal.

"It's like a thick snake," said the first philosopher, touching the trunk and feeling it moving about. "It's like a fan," argued the second, touching a large ear. "No, it's like a spear," observed the third, feeling the hard smooth tusk. "It's like a tree," said the fourth, touching the elephant's leg.

They were all right and none of them described the elephant as a whole.[8]
□

7.3.1 Structures and Views

Informally, a structure decomposes a system into related parts or elements. More precisely, an *architectural structure* consists of a set of elements and a binary relation on the elements. An *architectural view* is a representation of one or more structures on the same elements. A structure is a mathematical abstraction; a view is a concrete representation of structures.

Example 7.4. In the view in Fig. 7.4, the four boxes represents the elements of a compiler and the lines represent the flow of data between the elements. In effect, the lines relate pairs of elements, so they define a binary relation on the elements. The elements and the binary relation constitute a structure.

The *Data-Flow* structure includes the following relation:

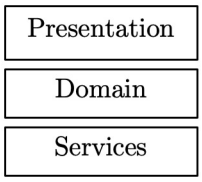


Figure 7.5: A layered view of an application.

From	To
Lexical Analyzer	Syntax Analyzer
Syntax Analyzer	Optimizer
Optimizer	Code Generator □

Example 7.5. The layered view of an application in Fig. 7.5 has three elements, represented by the boxes. By convention, in a layered view, the relationship between elements is represented by the vertical positioning of the layers. A layer may use the layer immediately below it, but a lower layer may not use any layer above it. The *May-Use* structure includes the following relation:

From	To
Presentation	Domain
Domain	Services □

Definition of Software Architecture

The potential for multiple structures and views leads to the following definition:

> "The *software architecture* of a system is the set of structures and views needed to reason about the system. The structures comprise software elements, relations among them, and properties of both."[9]

The concept of views has been incorporated into international standards for architectural descriptions.[10] Typically, a view outlines a solution to some architectural problem. Other aspects of the system that are not relevant to the solution are suppressed. No one view (or structure) captures everything about an architecture.

7.3.2 The 4+1 Grouping of Views

The above definition of software architecture does not specify the choice of views and structures. The choice is left to the architect for a given software project. In other words, the definition provides flexibility, but it does not provide any guidelines for which views to consider.

The *4+1* grouping of views proposes four kinds of views for a software architecture (see Fig. 7.6):

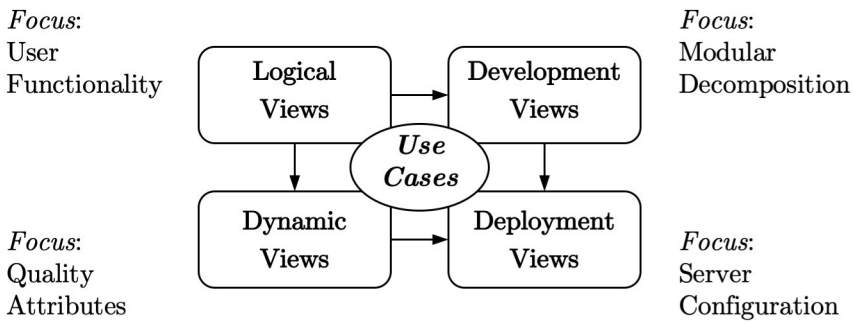


Figure 7.6: The 4+1 grouping of architectural views.

- *logical views* that describe the handling of functional requirements;
- *development views* of the modular decomposition of the source text;
- *dynamic views* for the run-time components and quality attributes; and
- *deployment views* for the mapping of the software onto servers.[11]

The 4+1 grouping is a helpful starting point for picking views for a project. The "+1" refers to selected scenarios or use cases, that span the four kinds of views: in Fig. 7.6.

Logical Views: Get Early Stakeholder Feedback

Logical views focus on end-user functionality. The architectural elements and relations in a logical view can be used to outline possible solutions. In the early stages of a project, logical views are helpful for getting feedback on questions like the following with users: Will the proposed solution meet user needs and goals? Will it satisfy all stakeholders? Are there any major gaps?

Logical views can also be used to probe for requirements that might change. Design decisions related to anticipated changes can be isolated in the architecture; see the discussion of modular architecture in Section 7.5. ‘

Development Views: Guide System Development

Development views focus on the static source code. For example, a development view might identify the major modules and their interactions with each other. Independent modules can be developed independently. When new members join a team, development views can be used to train them on the goals and architectural decisions for the project.

Project managers can use development views to make estimates, identify needed skills, and assign work. Estimation can be done by recursively estimating the parts of the system and then combining the estimates for the parts into

an estimate for the system as a whole. During iterative development, development views can be used for release and iteration planning.

Example 7.6. Using a practice known as continuous deployment, companies make tens, even hundreds, of small code changes a day to their production software. Each change is made by a small team, working independently.

The production software must be architected to accommodate the code changes gracefully. Development and deployment views are needed to implement the changes. □

Dynamic Views: Model System Properties

Dynamic views model the objects and components that arise at run time. Dynamic views can handle distributed or concurrent systems.

Models can be used to study quality attributes such as performance, reliability, and security. They can also be used for synchronization of concurrent processes and system fault tolerance.

Deployment Views (Physical Views)

Deployment views are also known as physical views because they deal with the actual servers that the software runs on. A deployment view could describe how software is distributed geographically, or how it might be replicated for scale and availability.

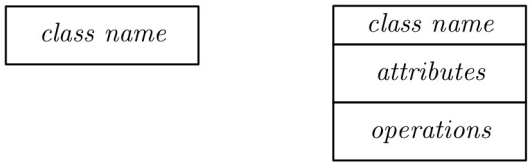
Example 7.7. With continuous deployment practices, deployment views would describe how freshly updated software is first deployed on a trial basis (with a few customers) before it goes into full production (with all customers). □

7.4 UML Class Diagrams

A *UML class diagram* is a development view of classes and their relationships. Individual classes are at the detailed design level; they embody decisions about the implementation of a system. Modules, which correspond to groupings of classes, provide a higher level view of a system. Modules are considered in Section 7.5. (Modules correspond roughly to packages in UML.)

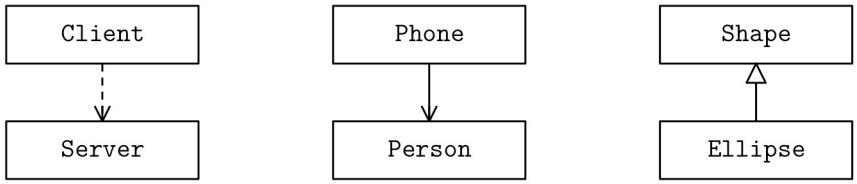
Unified Modeling Language (UML)

Unified Modeling Language (UML) provides a formal graphical alternative to informal box-and-line diagrams for software architecture. UML was formed by combining three separate object-oriented design methods. Its most popular facilities are those for modeling classes and their relationships. UML is a large and complex language that is widely known, but not so widely used; few understand it fully. This section introduces class, package, and sequence diagrams. See Chapter 5 for use cases, which are also part of UML.[12]



(a) (b)

Figure 7.7: (a) A class can be represented by its name, or (b) by its name and its attributes and operations.



(a) Dependency (b) Association (c) Generalization

Figure 7.8: Simple examples of relationships between classes. (a) A client depends on a server for services, but not vice versa. (b) Associated with each phone is a person (the owner). (c) A generalization is a subtype-supertype relationship. A shape is more general than an ellipse: a shape can also be a rectangle.

Representing Classes

Like any view, class diagrams focuses on the information that is relevant about a class and suppresses the rest of the information. The given purpose of the diagram guides what to show and what to suppress.

The simplest view of a class is a box with the name of the class; see Fig. 7.7(a). UML diagrams can be developed incrementally, with details added as needed. The view in Fig. 7.7(b) adds attributes and operations to the class name. Attributes and operations can be added selectively.

Class diagrams support three kinds of relationships: dependencies, associations, and generalizations. Like dependencies between modules (Section 7.5.1), dependencies between classes arise for a number of reasons Two key reasons are (1) a caller-callee relationship, or (2) a uses relationship, where one class uses another class for its implementation. In Fig. 7.8(a), a client depends on a server for services. Dependencies are directional: the server does not depend on the client.

Associations and generalizations are discussed below.

Properties as Attributes

A *property* denotes a range of values. Properties can have names and types. Informally, a property of a class can be thought of as a field in the class. Properties can be written as text or drawn as links in a diagram. In written form, they must have names and are called *attributes*; in graphical form they are called *associations*.

In class diagrams, an attribute is written as a line of text within a class; see Fig. 7.7(b). For example, a class `Shape` might have an attribute `center` for the geometric center of a shape. The following are some of the ways of writing the attribute:

`center`	Only the name is required
`center : Point`	`center` has type `Point`
`center : Point` = origin	Default initial value is `origin`
`+center`	`center` is public

In attribute form, a property has the following syntax:

⟨*visibility*⟩ ⟨*name*⟩ `:` ⟨*type*⟩ ⟨*multiplicity*⟩ `=` ⟨*default-value*⟩

Here, ⟨ and ⟩ enclose placeholders. The colon, `:`, goes with the type; it is dropped if the type is not provided. Similarly, the equal sign, `=`, is dropped if no default value is provided. The default value is used for initialization at run time, if an initial value is not specified.

Visibility

The visibility markers are as follows:

`+`	public: visible to other classes
`-`	public: not visible outside the class
`~`	visible to classes within this package
`#`	"protected:" rules vary subtly across languages
⟨*empty*⟩	visibility unspecified

While the meaning of public and private visibility is essentially the same across languages, the interpretation of package and protected visibility can vary subtly. Java and C++ have different notions of protected. In Java, it means visible within the package and to subclasses, whereas in C++, it means visible only to subclasses.

Multiplicity

Just as a course can have multiple students, some properties may represent multiple objects. *Multiplicity* specifies a range for the number of objects represented by a property. A range is written as

$m..n$, where $m \geq 0$ and $m \geq n$ or]tt *

The symbol * represents some unlimited non-negative number. An integer n is an abbreviation for the range $m..n$. Thus, `1` is an abbreviation for `1..1`.

When an attribute specification includes a multiplicity, the range is enclosed between `[` and `[`, as in

```
center :  Point [1] = origin
```

Here, `[1]` is redundant because 1 is the default, if multiplicity is not specified.

Operations

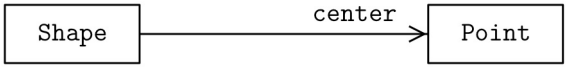
Informally, operations correspond to methods in a class. The syntax for operations is as follows:

⟨*visibility*⟩ ⟨*name*⟩ `(` ⟨*parameters*⟩ `)` `:` ⟨*return-type*⟩

The name and the parentheses are required; the other elements of the syntax are optional. If the return type is not specified, then the colon, `:` is dropped.

Properties as Associations

Shapes and their geometric centers were modeled above by a class `Shape` with a property `center` of type `Point`. This property can alternatively be drawn as an association between classes `Shape` and `Point`:



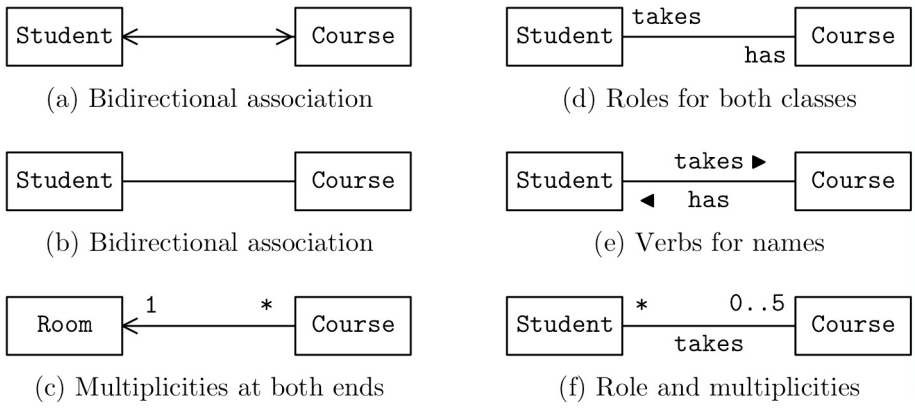
More generally, an association between classes A and B depicts a property of A that has values of type B. Associations are drawn as solid open arrows. The name, visibility, and multiplicity of the property are optional in an association.

With two notations for properties the question arises: when to use attributes and when to use associations? As a rule of thumb, use an association when both its source and target are significant classes in their own right. Use an attribute if the underlying property has relevance primarily to the given class.

Bidirectional Associations. Associations can be bidirectional. A bidirectional association represents a pair complementary properties. Bidirectional associations are depicted either by arrow heads at both ends or by a line without arrow heads. In Fig. 7.9(a-b), the association between `Student` and `Course` is bidirectional. The relationship between students and courses is complementary: students take courses and courses have students.

Roles. Associations can be named either after the underlying property or by the role played by a class. For example, students take courses and courses have students. A role is shown next to the class.

If a name is a verb, it can be used in a sentence of the form



(a) Bidirectional association

(b) Bidirectional association

(c) Multiplicities at both ends

(d) Roles for both classes

(e) Verbs for names

(f) Role and multiplicities

Figure 7.9: Examples of associations.

langlesource classrangle langleverbrangle langletarget classrangle

For students and courses, we might have

A student takes courses.
A course has students.

Verbs are directional. The direction of a verb is indicated by a solid triangle.

Multiplicity. Multiplicity allows us to model the fact that a student can take multiple courses and, conversely, a course can have multiple students. The example in Fig. 7.9(c) has a unidirectional association between `Course` and `Room`. We can specify that a course meets in a single room by placing `1` next to room. The `*` next to `Course` denotes that this property applies to all rooms. That is, for all courses, each course has a single meeting room.

Roles, direction, and multiplicity can be combined, as in "A student take up to 5 courses.

7.5 Modules

A *module* is a grouping of program elements, such as classes, values, and perhaps submodules. A subset of the program elements is identified as the *interface* of a module. Modules correspond to packages in UML or in a language such as Java.

This section introduces modules, and some guidelines for designing them. A large system can have tens or hundreds of modules, A module hierarchy provides a quick overview of a system, especially if it is accompanied by module dependencies (e.g., who calls whom). This section also includes a template for

1. Understanding the rationale behind a piece of code	66%
2. Having to switch tasks often because of ... teammates or manager	62%
3. Being aware of changes to code elsewhere that impact my code	61%
4. Finding all the places code has been duplicated	59%
5. Understanding code that someone else wrote	56%
6. Understanding the impact of changes I make on code elsewhere	55%
7. Understanding the history of a piece of code	51%
8. Understanding who "owns" a piece of code	50%

Figure 7.10: Percentage of Microsoft developers agreeing that the statement represented a "serious problem for me."

a module guide: the purpose of a module guide is to provide brief descriptions of modules, in plain English.

7.5.1 Modular Software Architecture

A software architecture is called *modular* if it is composed from modules that interact only through their interfaces. The benefit of a modular architecture is that changes that are internal to a module are isolated within the module and do not ripple through to affect the other modules. In other words, as long as there is no change to the interface of a module M (and the services provided by M through its interface), changes within M do not require changes to the other modules.

Example 7.8. The architecture of a compiler in Fig.7.4 is modular. (More precisely, the diagram in the figure is a logical view of the modular architecture of a compiler.) The boxes in the figure represent modules. The only interaction between the boxes is that the output of one module becomes the input to the next, from left to right. The modules therefore interact only through their interfaces. □

Well designed modules can help address developer concerns like the following:

> "Being aware of changes to code elsewhere that impact my code"
> "Understanding the impact of changes I make on code elsewhere"

These were among the top eight concerns that emerged from a survey of Microsoft software architects, developers, and testers; see Fig. 7.10. Modular architecture cannot help with interruptions by teammates and managers, but it can help address the other concerns.[13]

Module Dependencies

Dependencies between modules can arise for a number of reasons, as we shall see when coupling is discussed later in this section. Here are two common forms of dependencies:

call A caller depends on its callee. If a source module (the caller) uses a service provided by a target module (the callee), then the source depends on the target.

use A uses B if B must work for A to work. More precisely, A *uses* B if B must be present and satisfy its specification for A to satisfy its specification.

Calling is a form of using, since the callee must work for the caller to work.

Example 7.9. With a layered architecture, an upper layer uses the layer immediately below it, but not vice versa; for a diagram of a layered architecture, see Fig. 7.5. (Technically, an upper layer "may use" the layer below it, but when discussing dependencies, the safe assumption is that the upper layer does indeed use the lower layer.) Meanwhile, lower layers are independent of upper layers. □

In general, module A *depends* on module B if a change to B can lead to a change in the semantics of A. The change to B may break A or change A's functionality. Thus, if B changes, then A must be examined to assess whether A is affected and needs to be modified.

With classes, a subclass depends on its superclass, but not the other way around. The superclass is independent of its subclass. An object depends on its class; the '*instance-of* relation between an object and its class is a form of dependency.

Example 7.10. Are there dependencies between the modules in the diagram of a compiler in Fig. 7.4? No! The arrows in the diagram represent data flows, not dependencies. Consider the first two modules, from left to right. The output of the lexical analyzer becomes the input to the syntax analyzer. Given an input, the lexical analyzer produces some output, independently of the syntax analyzer. Similarly, given an input, the syntax analyzer produces some output, independently of the lexical analyzer. Thus, the input-output functionality of either of these modules is independent of the other module. This reasoning extends to all of the modules in the architecture: they are all independent of each other. □

Dependencies are Not Transitive

We might expect that if module X depends on Y and Y depends on Z, then it follows that X depends on Z. But, it may not be true that X depends on Z.

To see why, consider an application A, where A uses a data-access broker B to fetch data from a database module D. Here, the application A depends on

the broker B and the broker B depends on the database D. But, the purpose of the broker is to insulate the application from the database. If the database changes, the broker needs to change, but it can still maintain a stable interface for the application. So, the application does not need to change. The job of the broker is to insulate the application from changes in the database.

7.5.2 Designing Modular Architecture

Modular architectures are based on the principle of *information hiding*: isolate design and implementation decisions in modules. The concepts of coupling and cohesion are in the same spirit. *Coupling* is the degree to which there are dependencies between the modules in a system. *Cohesion* within a module is the degree to which the elements of the module belong together. Low coupling and high cohesion are desirable for modular architectures. Further design guidelines appear below.

Information Hiding Principle

The intent behind modular architecture is to make software easier to understand and change. Hidden design decisions are referred to as a module's *secrets*.[14]

Design decisions are the decisions that guide an implementation of some task. Design decisions include the choice of data data structures and algorithms. If an algorithm might change, isolate it in a module, so it can be improved in the future, without affecting the rest of the system. If a user interface might change—and they frequently do–isolate the specifics of the user interface in a module so the layout and color schemes can be tweaked without touching the rest of the system.

Information hiding is more about conceiving and thinking of programs than it is about specific programming languages or constructs. It can be practiced by controlling the visibility of the program elements in a module: private elements are not visible outside the module; the public elements are part of the module's interface.

Coupling

Two modules are *loosely coupled* if they interact only through their interfaces; they are *tightly coupled* if the implementation of one module depends on the implementation of the other. The following list progresses from looser (better) coupling to tighter (worse) coupling:

- *Message Coupling.* Modules pass messages through their interfaces.
- *Subclass Coupling.* A subclass inherits methods and data from a superclass.
- *Global Coupling.* Two modules share the same global data.
- *Content Coupling.* One module relies on the implementation of another.

Depending on the language, there may be other possible forms of coupling. For example, in a language with pointers, module A can pass module B a pointer that allows B to change private data in A.

Cohesion

A module has *high cohesion* if the module has one secret and all its elements relate to that secret; it has *low cohesion* if its elements are unrelated. The Unix philosophy of having each tool do one thing well leads to tools with high cohesion.[15] The following forms of cohesion reflect different approaches to grouping program elements into modules. The list progresses from higher (better) cohesion to lower worse cohesion:

- *Functional Cohesion.* Group elements based on a single well defined decision or functionality.
- *Sequential Cohesion.* Group based on processing steps. The phases of a compiler—lexical analysis, syntax analysis, optimization, code generation—represent processing steps. A module with sequential cohesion can potentially be split into submodules, where the output of one submodule becomes the input to the next.
- *Informational Cohesion.* Group based on the data that is being manipulated. Such a module can potentially be split by grouping based on the purpose of the manipulations.
- *Temporal Cohesion.* Group based on the order in which events occur; e.g., grouping initializations or grouping housekeeping events that occur at the same time. Redesign modules based on object-oriented design principles.
- *Coincidental Cohesion.* The elements of a module have little to do with each other. Redesign.

Design Guidelines

The following design guidelines for modular architecture are based on the principle of information hiding:[16]

- Begin with a list of significant design decisions or decisions that are likely to change. Then put each such design decision into a separate module so that it can be changed independently of the other decisions. Different design decisions belong in different modules.
- Keep each module simple enough to be understood fully by itself.
- Minimize the number of widely used modules. Design so the more widely used a module, the more stable its interface.
- Hide implementations, so that the implementation of one module can be changed without affecting the behavior and implementation of the other modules.

- Plan for change, so that likely changes do not affect module interfaces; less likely changes do not affect the interfaces of widely used modules; and only unlikely changes affect the modular structure of the system.
- Allow any combination of old and new implementations of modules to be tested, provided the interfaces stay the same.

7.5.3 Managing Modules

Beyond a dozen or so modules, it becomes increasingly difficult to find the modules that are relevant to some specific aspect of a system. Below, we consider two aids to navigating through the modules of a large system. The first aid is a module hierarchy, which reflects module-submodule relationships. If an automated took is available, the module hierarchy can be extracted from the source code.

The second aid is a plain English guide, with brief descriptions of the modules. A module guide need not be elaborate and it need not be completed up front. Developers can start with an outline and fill in design decisions as needed. The preparation of a simple module guide can be a useful exercise for systems large and small. The exercise prompts developers to think about and sketch the design of each module. The exercise can therefore help to validate the design.

Module Hierarchies

A *module hierarchy* imposes a tree structure on modules, where a parent module has its submodules as its child nodes. In other words, the parent-child relationship in the module hierarchy corresponds to the module-submodule relationship in the source code. By design, the secret of a child module is a sub-secret of the secret of its parent module. (Recall that secret refers to the design and implementation decisions that are isolated in a module; the isolated decisions are hidden from the rest of the system.) A consequence of this design is that one branch of the hierarchy can therefore be studied with minimal knowledge about modules that belong to unrelated branches.

Example 7.11. The partial module hierarchy in Fig. 7.11 is for a communication app for a tablet or smartphone. The app handles sessions that support any combination of audio, video, and text communication. A user can be in multiple sessions at the same time. Each session is represented by a window. People can be added or removed from a session by dragging and dropping their contact cards. In other words, each session is a conference, even a session with two participants.

The names of the submodules in Fig. 7.11 are Model, View, and Controller. The names anticipate the model-view-controller architectural pattern in Section 8.5.

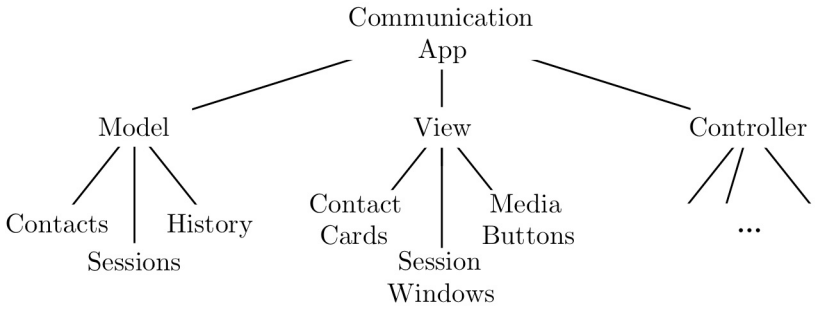


Figure 7.11: A partial module hierarchy for a communication application.

The Model has three submodules: Contacts, Sessions, and History. The Contacts submodule is responsible for managing a contact's personal information, addresses, and preferred communication modes (audio, video, text). The Sessions submodule is responsible for the current sessions, including session participants and their communication modes; for example, a participant may be able to receive both audio and video, but may only be able to respond via audio. The History submodule is responsible for information about past sessions.

The information about contacts, sessions, and communication history in the model is distinct from a view of such information. For example, the view on a tablet with a large screen may be quite different from the view on the smaller screen of a smartphone.

The modular architecture of the Controller module is omitted. □

UML Package Diagrams

Modules correspond to packages in UML; module hierarchies correspond to package diagrams. A *UUML package* is a grouping of UML elements. Packages can be nested; that is, a package can have sub-packages.

The package diagram in Fig. 7.12 corresponds to the module hierarchy in Fig. 7.11. In UML package diagrams, packages are represented by tabbed boxes. The simplest representation of a package is a box with the name of the package in box; e.g., see the box for the Controller on the right. When the contents of a package can be shown within the box, then the name of the package belongs in the tab; e.g., see the representation of the View in the middle of Fig. 7.12.

Module Guides

A *module guide* is a description of a module hierarchy in plain English. A plain English guide is intended to supplement, not replace, a specification of module interfaces. The purpose of a guide is threefold:

- provide an overview of the system;

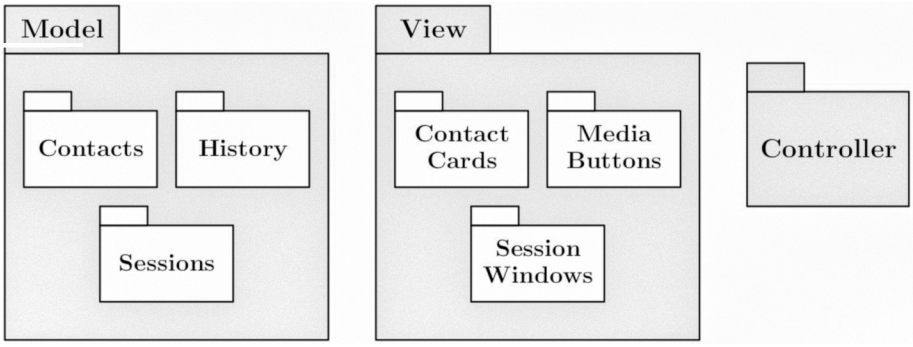


Figure 7.12: Packages corresponding to the module hierarchy in Fig. 7.11 .

- bring out the context and assumptions behind the design approach; and
- describe the responsibilities and behavior of the modules.

The template in Fig. 7.13 touches on the main points about a module that need to be covered by a guide. The template begins with the name and a plain English textual description of the module. The description includes the responsibility of the module and an overview of the design decisions that are hidden by the module. Any notes to the reader can also be included as part of the description.

The subsection on the module's service helps the reader find relevant modules. From the service, readers can decide whether the module is relevant to the aspect of the system that is of interest to them. If the module seems relevant, they can consult a module interface specification for more information.

The guide includes the module's secret, but not the implementation of the secret. Recall that the purpose of the guide is to allow the overall behavior of the system to be inferred, without looking at the implementation. In some cases, it is helpful to include secondary secrets that are uncovered during the implementation of the primary secret.

7.5.4 Addressing Developer Concerns

This section considers three forms of documentation:

- *Module Hierarchy.* Organize the modules into a hierarchy that identifies modules and their decomposition into submodules.
- *Module Dependencies.* Identify dependencies between modules. A dependency graph can be used to gauge the potential ripple effects of a change to a given module.
- *Module Guide.* Provide a concise guide to the modules, written in plain English. The guide focuses on the rationale for the module's design.

- **Module Name**
  - **Textual Description**
    - The responsibility of the module
    - Overview and context for the service and the secret of the module
    - Notes to the reader
  - **Service Provided**
    - Service provided to the other modules through the module interface
  - **Secret**
    - Primary design decision that is hidden by the module
    - Any secondary design decisions that are needed for the implementation
  - **Error and Exception Handling**
    - List of possible errors and exceptions

Figure 7.13: Template for a Module Guide.

Such documentation can help address the developer concerns listed in Fig. 7.10, including:

> "Understanding the rationale behind a piece of code"
> "Understanding code that someone else wrote."

Someone new to the system can use the guide to navigate through the module hierarchy and find the modules that are relevant to a proposed change. Someone familiar with the system can use the guide and dependencies to convince themselves that any change is propagated to all the modules that are affected by the change.

A module guide is separate from an interface specification. Once the relevant modules are found with the help of the guide, the specifications and implementations of the modules can be consulted for more information.

7.6 Conclusion

Software architecture touches all aspects of a project. For customers, an architecture is helpful for confirming the project's direction. For development teams, it is helpful for assembling skills and resources and for assigning work to developers. For system architects, it is helpful for modeling system properties such as performance, security, and availability. For project managers, it is helpful for estimating budgets and schedules.

These many uses of architecture lead to multiple views of an architecture, where a *view* focuses on some aspect of the system or on the solution to some

specific problem. The 4+1 grouping of views further illustrates the many roles for software architecture:

- *Logical Views* focus on end-user functionality and the problem domain.
- *Development Views* focus on program structure and the source text.
- *Process Views* focus on run-time objects and processes.
- *Physical Views* focus on system configurations and deployment.

These views are accompanied by *scenarios* or use cases, which span the four views.

Technically, a view is a structure, where a *structure* consists of a set of components and a binary relation on the components. Just as there are multiple views, there are multiple structure; e.g., the modular or component structure of the system and the calling or the "who calls who" structure for components.

No one view or structure defines an architecture. Hence the following definition (see Section 7.3.:

> "The *software architecture* of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."

The main principle for creating an architecture is *information hiding*: isolate design decisions into program units called modules, so individual decisions can be changed without affecting the rest of the system. A *module* is a collection of related program elements, like classes, that implement design decisions.

Here are some guidelines for designing modules (see Section 7.5):

- Isolate each significant design decisions into a separate module.
- Keep each module simple enough to be understood fully.
- Minimize the number of widely used modules.
- Hide implementations, so that they can be changed independently.
- Plan for change, so that likely changes are easy and only unlikely changes result in a redesign.
- Allow any combination of old and new implementations of modules to be tested, provided the interfaces stay the same.
- For a product family, design for commonalities before variabilities.
- Hide implementation decisions about variabilities in separate modules.

A *product family* or *product line* is a set of related programs that is specifically designed to be implemented together. The common properties of the family are called *commonalities* and the special properties of individual family members are called *variabilities*. Versions of a program can be treated as a family, as can successive working versions that are implemented during an iterative development process. Thus, the family approach is helpful for planning iterations of the same system.

Beyond a dozen or so modules, it is helpful to create a *module hierarchy*, a tree-structured grouping of related modules that share a secret; the secret of a submodule is a subsecret of its parents' secret. The *secret* of a module refers to the design decisions that are hidden in the module. In effect, the secret is the implementation of the behavior of the module.

Someone tasked with maintaining or changing a system would benefit greatly from a module guide. A *module guide* is a plain English description of the module hierarchy that provides an overview of the modules and includes the rationale for the design decisions that are relevant to a module.

A module guide is highly recommended for even a small system. For a large system with tens or hundreds of modules, it is invaluable.

Exercises for Chapter 7

Exercise 7.1. Modules are a key concept in software architecture and design.

- What is Information Hiding? Define it, explain it, and give an example.
- What is a Module Guide? List its main elements and their purpose or roles.

Exercise 7.2. Are the following statements generally true or generally false? Briefly explain your answers.

a) The Information Hiding Principle refers to hiding the design decisions in a module.

b) A Module Guide describes the implementation of each module.

c) A module with a secret cannot be changed.

d) A Module Interface Specification specifies the services provided and the services needed by modules.

e) With the XP focus on the simplest thing that could possibly work and on refactoring to clean up the design as new code is added, there is no need for architecture.

f) A system that obeys the Information Hiding principle is secure.

g) If module A uses module B, then A and B must have an ancestor-descendant relation in the module hierarchy.

h) If module A uses module B, then B must be present and satisfy its specification for A to satisfy its specification.

i) A Development View of an architecture specifies the internal structure of modules.

j) Conway's "law" implies that the architecture of a system reflects the social structure of the producing organization.

Exercise 7.3. A video-streaming service wants to track the most-frequently requested videos. They have asked your company to bid on a system that will accept a stream of video orders and incrementally maintain the top ten most popular videos so far. Requests contain other information besides the item name; e.g., the video's price, its category (e.g., Historical, Comedy, Action).

a) Using Information Hiding, give a high-level design.q Include each module's secret.

b) Change your design to track both the top ten videos and the top ten categories; e.g., to settle whether Comedy videos are more popular than Action videos.

Exercise 7.4. Coupling is the degree to which modules are inter-related. Forms of coupling include:

a) *Message*: pass messages through their interfaces.

b) *Subclass*: inherit methods and data from a superclass.

c) *Global*: two or more modules share the same global data.

d) *Content*: one module relies on the implementation of another.

For each of the above cases, suppose modules A and B have that kind of coupling. How would you refactor A and B into modules $M_1, M_2, \cdots$ that comply with Information Hiding and provide the same services as A and B. That is, for each public function $A.f()$ or $B.f()$ in the interfaces of A and B, there is an equivalent function $M_i.f()$, for some refactored module M_i.

Exercise 7.5. For the system in Exercise 7.3, you decide to treat the system as a product family because you recognize that the same approach can be applied to track top selling items for a retailer or the top most emailed items for a news company.

a) What do product family members have in common?

b) What are the variabilities; that is, how do product family members differ?

Exercise 7.6. KWIC is an acronym for Key Word in Context. A KWIC index is formed by sorting and aligning all the "significant" words in a title. For simplicity, assume that capitalized words are the only significant words. As an example, the title `Wikipedia the Free Encyclopedia` has three significant words, `Wikipedia`, `Free`, and `Encyclopedia`. For the two titles

```
KWIC is an  Acronym for Keyword in Context
Wikipedia the Free Encyclopedia
```

the KWIC index is as follows:

```
                  KWIC is an   Acronym for Keyword in Conte
is an Acronym for Keyword in   Context
          Wikipedia the Free   Encyclopedia
               Wikipedia the   Free Encyclopedia
      KWIC is an Acronym for   Keyword in Context
                               KWIC is an Acronym for Keywo
                               Wikipedia the Free Encyclope
```

Design an architecture for KWIC indexes that hides the representation of titles in a module.

a) Give brief descriptions of the modules in your architecture

b) For each module, list the messages to the module and the corresponding responses from the module.

c) Give a module hierarchy

d) Describe the secret of each module

Exercise 7.7. Given a month and a year, the Unix `cal` command produces a calendar for that month; e.g., `cal 10 1752` produces

```
October 1752
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Given a year, as in `cal 2000`, the output is a calendar for that year.

Your task is to design a modular architecture for a proposed implementation of the `cal` command. (See instructions below.)

a) List the modules in your architecture. Design each module so it has a coherent responsibility and so it can be modified independently of the others. Use descriptive names for the modules.

b) Provide a brief overview of how the modules interact to produce the calendar for a month.

c) Provide a module guide for your architecture—use the template for the guide.

Here are some instructions:

- Allow for your version of `cal` to be modified, if needed, to produce dates in the European style, with Monday as the first day of the week in the calendar for a month. Note that the example above is in the American style, with Sunday as the first day of the week.

  The format for a yearly calendar has yet to be determined.

- As a simplification, your design must work for months starting October 1752 and full years starting 1753. Thus, the start date for your calendar is Sunday, October 1, 1752. (The US calendar changed in September 1752: the month had only 19 days, instead of the usual 30.)
- Note that this exercise is about a modular software architecture, not its implementation.

Chapter 8

Architectural Patterns

> " there are beds and tables in the world—plenty of them, are there not?
>
> "Yes.
>
> "But there are only two ideas or forms of them—one the idea of a bed, the other of a table."
>
> — *Plato, circa 380 BCE.*[1]

---

a

8.1 Introduction

Rather than start each design from scratch, software architects tend to adapt earlier successful designs for similar problems. Even if they do not reuse any code from an earlier project, the solution approach gives them a starting point for creating a solution to the current problem. Problems like the design of a graphical user interface, or the design of a compiler, have been addressed over and over again. The design of another graphical user interface is bound to have properties in common with previous designs.

Patterns Deal With Recurring Problems

An *architectural pattern* outlines properties that are common to all solutions of a problem that occurs over and over again. In short, a pattern provides the "core of a solution" to a recurring problem.

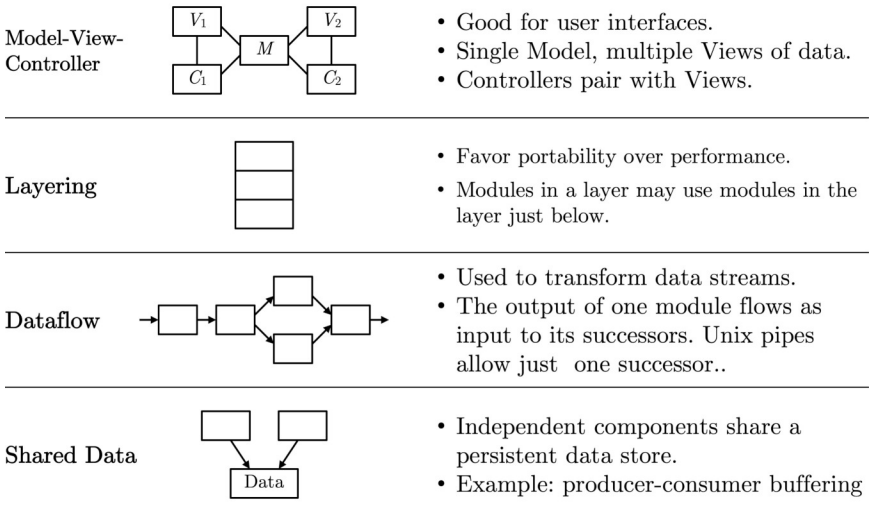


Figure 8.1: Some frequently occurring patterns.

Patterns were introduced by the building architect and urban planner Christopher Alexander. The concept of patterns carries over directly to software. The term "pattern" has become part of the vocabulary of software engineering.[2]

The description of a pattern in this chapter has three sections:

- *Context.* An optional description of the context for the pattern.
- *Problem.* A statement of some aspect of a recurring problem.
- *Core of a Solution.* The core of a solution consists of design guidelines for creating a solution to the problem..

Any catalog of patterns is very likely to be incomplete.[3] There were early attempts to classify patterns in terms of the nature of their architectural elements and their connectors. However, given the diversity of software problems, a complete list of patterns has not emerged. This chapter introduces some frequently occurring architectural patterns. See Fig. 8.1 for a partial list of the patterns in this chapter. Additional patterns related to clients and servers appear in Section **??**. Section 8.8 deals with product families and product lines, which are built around a set of common properties.

Quality Attributes Influence Architecture

"Quality attributes" is another term for nonfunctional requirements, which include system attributes such as modifiability, scalability, reliability, and so on. These attributes are key considerations when choosing patterns. They have to

be built into a system from the start; they can be hard to retrofit. The overview in Fig. 8.1 include brief notes about the quality attributes that the patters are good for.

8.2 Software Layering

In earlier chapters, software layering was discussed informally. The idea is that software is organized into layers that are stacked, one above the other. An upper layer may use the layer just below it.

Layering is useful for writing reusable software. For the distinction between reusability and portability, consider an app running on an operating system. We want the operating system to be reusable by other apps, and we want the app to be portable so it runs on other operating systems. The operating system does not change—it is reused—when the app above it is changed.

This section explores the layered pattern. The Internet protocols, TCP and IP, make a good case study, since layering has been enormously successful for the Internet—we all use the protocols every day. Looking ahead, a layer is a cohesive grouping of modules. It helps if each layer has a responsibility, which the modules in the layer carry out.

8.2.1 The Layered Pattern

Context. The general problem is to simplify the development of software by building useful software components.

Problem. Design and build reusable software.

Core of a Solution. Partition the system into groups of related modules called *layer*s, where each layer has a specific responsibility. Organize the layers so they can be stacked vertically, one layer above another, as in Fig. 8.2. The modules in a layer A may use the modules in layer A and in the layer immediately below, but they may not use any modules in any layer above A. □

Example 8.1. Consider an application that runs on a virtual machine that runs on an underlying operating system. The application may use the services of the virtual machine. Meanwhile, the virtual machine may use the of the underlying operating system, but, not the other way around. The operating system can do its job with or without the virtual machine, and the virtual machine can do its job with or without the application. □

Example 8.2. The diagram in Fig. 8.2 is motivated by apps running on a smartphone. The top layer is for apps available to the user. Just below the top layer is a framework that provides building blocks for assembling apps.

The bottom layer is for the operating system, which hides the hardware. Immediately above the operating system is a layer that supports services that

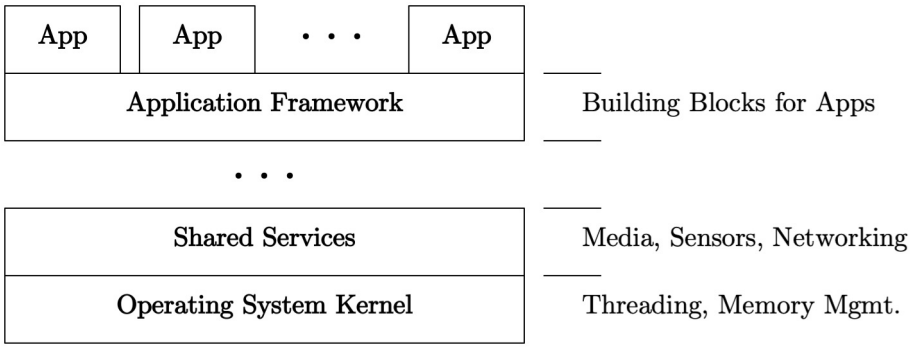


Figure 8.2: A layered architecture for implementing apps.

are shared by the apps, such as services for audio, video, sensors, and network connections.

In between the top two and the bottom two layers. there may be one or more other layers. For example, the Android architecture has a "runtime" layer that insulates the apps: each app sees its own virtual machine. □

Strict Layering

The layered pattern results in *strict layering*, where each layer knows only about itself and the interface of the layer directly below. Layer boundaries are hard. Layer implementations are hidden. New layers can be added without touching the layers below.

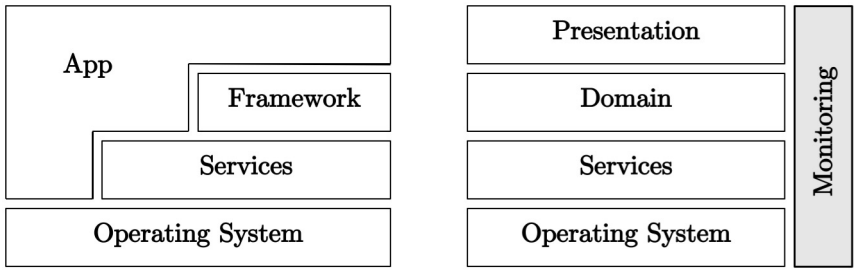
As we shall see, Internet protocols rely on strict layering. The story is different with operating systems, compilers, and other systems software. Systems software is implemented and then used over and over again. Designers have been known to (very carefully) some purity of layering for performance.

8.2.2 Variants of the Layered Pattern

The layered pattern has spawned a number of variants. Two of them are considered below: layer bridging and layers with sidecar.

Layer Bridging

With *Layer Bridging*, a layer may have more than one layer directly below it. In Fig. 8.3(a), the App layer is drawn so it touches each one of the other layers below it. Thus, modules in the App layer may use modules in any of the other layers: Framework; Services; and Operating System.



(a) Layer Bridging (b) Layers with Sidecar

Figure 8.3: Variants of the Layered Pattern.

Layers With Sidecar

The *layers-with-sidecar* pattern differs from the Layered pattern by having one layer—called the *sidecar*—that vertically spans all the other horizontal layers; e.g., see Fig. 8.3(b). A vertical cross-cutting sidecar layer may be used for debugging, audit, security, or operations management. In Fig. 8.3(b), the sidecar layer is a monitoring layer.

Care is needed with a sidecar to ensure that layering is preserved; that is, lower layers cannot access upper layers indirectly through the sidecar. Layering can be preserved if communications with the sidecar are one-way, either to or from the sidecar, but not in both directions. In a debugging or audit setting, modules in the sidecar may reach into the other layers and use modules in the horizontal layers. On the other hand, in a logging or monitoring setting, modules in the horizontal layers may use the sidecar to deposit event logs.

8.2.3 Case Study: The Internet Protocol Suite

Internet protocols provide prime examples for illustrating the following:

- *Strict Layering.* Endpoints attached to the Internet can experiment with and add new protocols independently of the Internet infrastructure. Web browsing capabilities were added with HTTP in 1991, long after the main protocols, TCP and IP, were defined in the 1970s.
- *Specific Layer Responsibilities.* As we shall see, each layer has a distinct responsibility; see Fig. 8.4.
- *Design Tradeoffs.* For the Internet, the benefits of strict layering far outweigh the cost. The cost is in the form of data transmission overhead, as we shall see below.

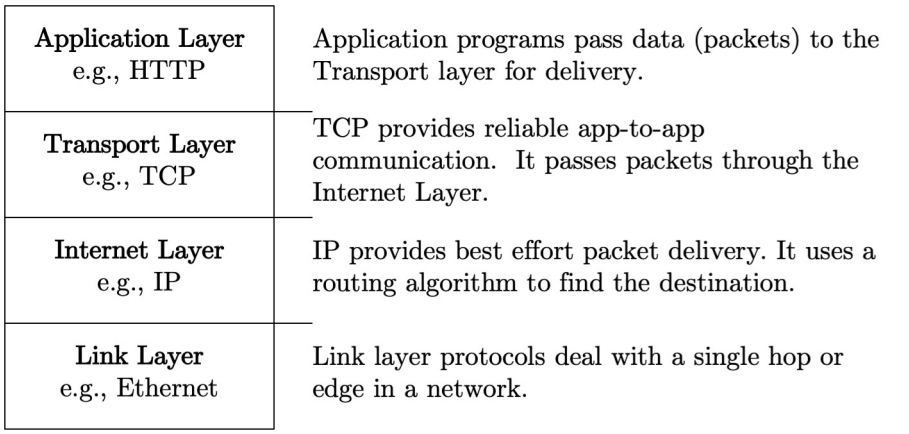


Figure 8.4: Layers of the Internet Protocol Suite.

Layers of the Internet Protocol Suite

Endpoints attached to the Internet exchange data using protocols from the Internet Protocol Suite, also known as the *IP stack*. The IP stack has four layers: Application, Transport, Internet, and Link. Conceptually, a given layer at the sender endpoint communicates with the corresponding layer at the receiver endpoint. The endpoints exchange data in chunks called *packets*.[4]

In the following overview of the layers, note that each layer has a distinct responsibility.

Application Layer. Starting at the top of Fig 8.4, HTTP is a typical Application Layer protocol. Conceptually, a browser sends an HTTP packet to a web server. Actually, the Application Layer at the browser uses the Transport Layer to deliver the packet to the Application Layer at the destination, the web server.

Transport Layer. TCP stands for Transport Control Protocol. TCP handles end-to-end communication: it ensures that packets are delivered reliably and in order. TCP sends packets through the Internet layer. If the Internet layer loses a packet, TCP retransmits the lost packet. Retransmission may introduce delay, so TCP favors reliable delivery over timely delivery. TCP relies on the Internet Layer to take care of getting packets from source to destination.

Internet Layer. IP stands for Internet Protocol. The Internet layer is responsible for routing packets through a network, where the network consists of a set of routers connected by links. A route is a path of links from source to destination. A packet from a source in New York to a destination in San Francisco might go through a router in Chicago. Then, if there is congestion in the network, another packet might take a more circuitous route or may simply be

dropped. IP provides *best effort* service, which means that IP provides timely delivery, but packets may be lost en route.

Link Layer. The links between routers in the network use various wired and wireless technologies. The Link layer isolates IP from the specifics of the network technology that is used to carry packets over a given link.

Distinct Roles of IP and TCP

From the above descriptions, TCP and IP have distinct responsibilities:

- TCP supports reliable end-to-end delivery, but packets might be delayed. The delays are due to retransmission of packets that are lost or received out of order.
- IP supports timely delivery, but packets might be dropped en route. IP is said to provide best-effort service.

Reliable delivery is needed for applications such as email, where delays are acceptable, but data must be delivered intact, without loss.

Timely delivery is needed for applications such as voice and video conferencing, where delay is unacceptable, but some loss can be smoothed over.

TCP and IP are an interesting case study because originally, there was only one protocol: TCP. IP was separated out as a separate layer with the rise of conferencing and the needs of an Internet debugger—for debugging, some timely best-effort data is better than data that is never delivered.[5]

Design Tradeoff: Header Overhead

Each protocol in the IP stack adds a little overhead to the data it carries. The overhead is in the form of *headers*, which are used for identifying information, such as source and destination addresses. The headers are a small price indeed for the flexibility provided by strict layering.

Example 8.3. For an overview of this example, see Fig. 8.5. The packet headers at each layer of the IP protocol stack represent data overhead. Headers are added as an application packet of size 64-1518 bytes goes down the protocol stack at the source. Headers are stripped at the destination as an application packet goes up the protocol stack.

The header sizes in Fig. 8.5 are based on the assumption that IPv4 (IP version 4) is the Internet Layer protocol and that Ethernet is the Link Layer protocol. For simplicity, we abbreviate IPv4 to IP.

In more detail, suppose a browser at a source endpoint in Denver wants to send some data to a web server at a destination endpoint in Virginia. The browser and web server are matching applications; they know how to interpret messages from each other. Both applications belong in the Application layers of their IP protocol stacks.

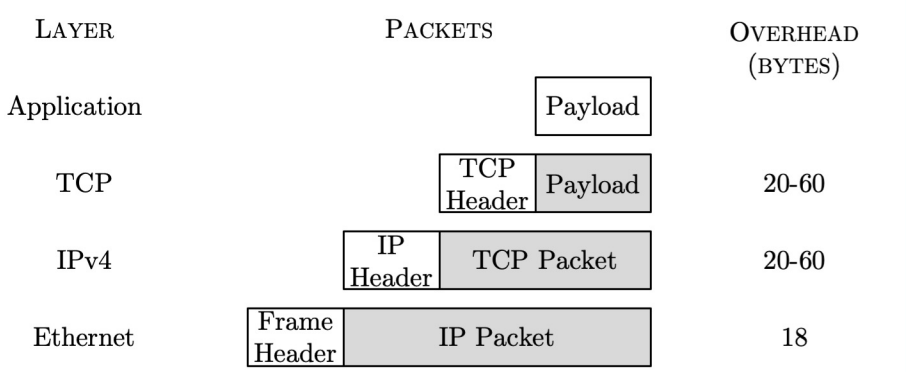


Figure 8.5: Packet headers represent overhead during data communication. The size of an application packet (payload)) is 64-1518 bytes. The layers are independent, so each layer has its own headers.

The source application in Denver sends data in packets that are between 64 and 1518 bytes long. Let us refer to an application data packet as a *payload*; see Fig. 8.5. The payload is handed down to TCP in the Transport Layer. Together, the payload and a TCP header form a *TCP packet*. A TCP header containing addresses and some bookkeeping information. TCP treats the payload as data to be delivered intact to the destination. The TCP header is 20-60 bytes.

TCP hands the TCP packet down to IP in the Internet Layer at the source. Together, an IP header and the TCP packet form an *IP packet*. IP treats the TCP packet as data to be carried; IP does not look inside the TCP packet. The IP packet then goes down to the Link Layer, where Ethernet has its own frame header.

At the destination in Virginia, headers are stripped as the payload goes up the protocol stack through the Link, Internet, and Transport Layers. Finally, TCP delivers the original payload to the web server application.

For short payloads, the size of the headers can far exceed the size of the payload. □

Design Tradeoffs: Unix Portability

Different projects make different tradeoffs between strictness of layering and performance. In the next example, some portability was sacrificed for continuing efficiency of the completed system..

When Unix was first ported from one machine to another, the designers prioritized efficiency for the user of Unix over some one-time difficulty for the designers. Measured in lines of C code, the operating system was 05% portable.

Example 8.4.

The Unix operating system and its derivatives, including Linux, run on a wide range of machines, from smartphones to servers in data centers. For each machine class, the operating system is ported once to that class of machines. Users, perhaps millions of them, then run copies of the operating system on their individual machines.

When Unix was first ported from one machine to another, the designers wanted both efficiency for users and portability for designers. Efficiency was a higher priority. Some portability as sacrificed: the resulting operating system was 95% portable, in terms of lines of code. One of the goals of the designers was "identifying and isolating machine dependencies." The 5% of the code that was machine dependent was there for efficiency: it took advantage of the special features of the underlying machine.[6] □

8.3 The Shared Data Pattern

The shared-data pattern is useful in its own right. It is also a useful building block for other more complex patterns. In a model-view-controller architecture, multiple views present data from a shared model; see Section 8.5. In a variant of the client-server architecture, a multiple servers access client data from a shared data store; see Section 8.7.

Name. The Shared Data Pattern.

Context. The need for a shared data repository arises when there are two or more independent components, where each of the components wants to access the data, but none of the components wants to "own" and manage all the data. Here, "own" means that the data does not logically fit with the responsibilities of any of the components. The data may range from a data structure shared by two components to a very large distributed data store that holds information about a company's business transactions.

Example 8.5. As a small example, consider two components: a producer and a consumer. The producer sends objects at its own pace, and the consumer receives those objects at its own pace. A shared data buffer allows the produce and consumer objects at different rates. If the producer runs ahead, then the buffer holds objects until the consumer is ready for them. At other times, the consumer might work down some of the objects held in the buffer. If the buffer is empty, the consumer waits until the producer adds another object to the buffer.

The use of a separate shared buffer simplifies the design of both the producer and consumer. The design also allows the implementation of the buffer to be changed without affecting the producer and the consumer. □

Problem. Allow multiple independent components called *accessors* to access and manipulate persistent data.

Core of a Solution. Add a new component called a *shared data store* to hold and manage the persistent data. The shared data store provides an interface through which the accessors access and manipulate the data. The accessors are independent of each other; they do not interact directly with each other. They communicate with each other through the shared data store. The accessors use the data store, but the data store is not aware of them—it responds to requests through the interface. □

Building on the Shared Data Pattern

The shared data pattern isolates and hides the implementation of the data store. The implementation can therefore be changed without affecting the accessors of the data store (provided the interface remains the same). The shared data pattern therefore accommodates a range of reliability, performance, and scalability requirements. The following comments apply to large data stores:

- *Reliability.* A shared data store is potentially a single point of failure. The implementation can be made fault tolerant by adding redundancy in the form of a backup copy of the data. In other words, there is a primary and a backup copy of the data. If the primary fails, the backup takes over. For added protection, the backup can be at a different geographic location. Geographic dispersion guards against a disaster at one geographic site.
- *Performance.* A shared data store can also be a potential performance bottleneck. Performance can be improved by caching. Caching can also help to reduce delays when data is accessed over a network, from a remote site. If a local cached copy of the data is available, then network delays can be avoided.
- *Scalability.* A shared data store can be scaled by adding copies of the data or by distributing the data across multiple servers. With copies comes the concern of data consistency. The copies need to be kept in synch with each other.

8.4 Observers and Subscribers

The observer and the publish-subscribe patterns are closely related. In both patterns, a *publisher* component raises (publishes) an event, unaware of who is interested in being notified about the event. An interested component is called an *observer* if it knows the identity of the publisher. It is called a *subscriber* if it does not know about the publisher. The difference between the two patterns is that the observer pattern has observers, and the publish-subscribe pattern has subscribers.

Shared Context for Observers and Subscribers

The term component refers to processes and objects that are created and garbage collected dynamically, at run time. Independent components may need to communicate during a computation, but may not know one another's identity. The reasons are twofold: (a) components can come and go dynamically at any time; and (b) identities may be hidden by design.

The Observer Pattern

Problem. How can observers be notified about events raised by publishers? An observer knows the identity of a publisher, but a publisher is unaware of the identity—or even the existence—of its observers.

Core of a Solution. The general idea is as follows:

1. An observer registers directly with the publisher of a specific class of events. Recall that observers know about publishers.

   An observer registers to be notified about a specific class of events.

2. When it has something to communicate (publish), the publisher raises an event.

3. The raised event is delivered to all observers to that class of events.

Registration and notification are often handled by maintaining lists of observers. When an observer registers for a class of events, it is put on a list for that event. The list is isolated from the rest of the code of the publisher, hence the claim that the publisher is unaware of the specific observers. The publisher knows that there is a list, but need not know the contents of the list. When the publisher raises an event, the list object notifies the observers. □

The Publish-Subscribe Pattern

Problem. How can subscribers be notified about events raised by publishers? Neither publishers nor subscribers are aware of each others' identity. A component can be both a publisher and a subscriber (of different classes of events).

Core of a Solution. With this pattern, a third party provides a level of indirection. Both publishers and subscribers know about the third party, but they do not know about each other. The third party takes the form of *middleware*, which sits in a layer between the operating system and an application. (The publishers and subscribers are part of the application.)

Communication between publishers and subscribers is through the middleware:

1. A subscriber registers with the middleware to be notified about a specific class of events.

2. A publisher raises an event through the middleware.

3. The middleware broadcasts the event to all subscribers.

4. , Subscribers are notified of all events. They pick out the events that are of interest to them.

The overhead of scanning for events of interest increases as the ratio of uninteresting events increases. The routing of events through the middleware also adds latency to event delivery. In other words, the use of the middleware adds a delay between the time a publisher raises an event and the time the event is delivered to a subscriber.

8.5 Interactive User Interfaces

Interactive user interfaces are routinely based on a model-view-controller (MVC) architecture, which takes its name from the three main components in the architecture.[7] This section explores the design decisions behind MVC variants.[8]

8.5.1 A Model-View-Controller (MVC) Pattern

By definition, a pattern describes properties and design decisions that are common to all possible ways of solving the problem. The central idea behind MVC architectures is the following:

> *Separated Presentation.* Separate the application domain objects from their presentation through the user interface. This decision enables multiple presentations of the same information.

In an MVC architecture, the model is responsible for the domain objects and logic. Together, a view and a controller are responsible for a presentation. Views and controllers are paired. Each pair gets the data it needs from the same model, but, each view-controller pair has its own way of presenting the data it gets from the model. (MVC architectures differ in the division of work between views and controllers.)

When it is clear from the context, we use the terms “presentation” and “view” interchangeably. Both terms refer to the display of information to a user. The other meaning of “view” is quite distinct: it refers to a programming component that is paired with a controller.

Figure 8.6: Two views of a photo object.

Separated Presentation

The popular saying, "The map is not the territory," illustrates the distinction between views and models: a model corresponds to the territory and a view corresponds to a map. There can be many maps of the same territory; for example, a satellite map, a street map, or a topological map. Even for the same map on a screen, say a street map, the information it conveys depends on the scale and size of the map. Additional details may appear as we zoom in from state to city to neighborhood.

The following example illustrates the separation between models and views.

Example 8.6.

In a photo-editing application, the photo objects belong in the model. Each photo object has attributes such as the photo's height, width, resolution, and the pixels that make up its image. The model also includes the logic for operations on the photo; e.g., for resizing it or cropping it.

Two presentations of a photo object appear in Fig. 8.6. Both presentation gets its information from the model, but note that they display different attributes. The image view on the left displays a picture of the Mona Lisa; that is, it displays the pixels in the photo object. The dialog view on the right displays numeric values for the photo's height, width, and resolution in pixels per inch. The height and width are in both inches and in pixels. □

The Need for Synchronizing with the Model

The need for synchronization arises when there are multiple presentations of the same information. A change through one presentation triggers the need for the other presentations to synchronize with the updated information.

Example 8.7. Consider again the photo-editing application in Example 8.6 and the two presentations of a photo in Fig. 8.6. For this example, suppose that the proportions of the photo are fixed. In other words, the ratio of height

to width is fixed: if the height changes, the width changes accordingly, and vice versa.

Now, suppose that the user doubles the numeric value of the height from 450 to 900 pixels in the dialog view. The model is unaware of the presentations, but in the other direction, the presentations know about the model. Between them, the view and the controller for the dialog presentation can therefore send a message to the model that the height has doubled. The model can then update the other photo attributes: the width must double and the pixels from the original image have toe be mapped onto the pixels for the doubled image (both the height and width double, so the number of pixels quadruples).

when the model changes, both presentations must synchronize with the model to ensure that they display the current state of the photo object. □

Observing the Model for Changes

To synchronize a presentation with the model, MVC architectures use the observer pattern introduced in Section **??**. Between them the view and the controller observe the model. When they are notified of changes, they retrieve the current information from the model. Is it the view or is it the controller that observes the model? The answer depends on the specific MVC architecture, as we shall see in Section 8.5.2. Looking ahead, in the classic solution, the view observes, but there are practical solutions in which the controller observes the model.

Basic Model-View-Controller (MVC) Pattern

The Basic MVC pattern builds on the above discussion of separated presentation and observer synchronization. It provides a close-to-complete solution to the problem. Since patterns describe properties of all possible solutions, the Basic MVC pattern leaves some specifics of views and controllers to be filled in by individual solutions. Section 8.5.2 fills in some of the details.

Name. Basic Model-View-Controller Pattern.

Context. User interaction with computers is predominantly through graphical and touch interfaces.

Problem. Given an application, design a user interface for it that supports user interaction through devices such as a mouse and keyboard for input and a display for output. Allow the same information to be presented in multiple ways.

Core of a Solution. The solution approach is to have three kinds of modules:

- *Model*. The key design decision is to hold the application-domain objects in the model, separate from their presentation. There is one model per user interface.

- *View.* Each view is paired with a controllers (see below).. There is a view-controller pair for each presentation of information through the user interface. A view is responsible for managing the display (for output) and for detecting mouse clicks, keyboard input, and touch gestures ((for input).
- *Controller.* A controller mediates between a view and the model. For example, in he dialog view in Fig. 8.6, when the user types 9, then 0, then 0 in a certain text box, the controller interprets the numeric value 900 as the height, and informs the model of the change in the photo object. The controller's role includes the following:

  a) *Interpret user activity.* When a view detects user activity—click, keystroke, or gesture–it consults its controller. The controller knows how to interpret the significance of the activity.
  b) *Handle updates to the model.* The controller decides on what to do when it interprets user activity. In particular, it decides whether the model needs to be told about the user activity.

The next example illustrates the roles of the model, view, and controller in a conferencing application.

Example 8.8. Consider a conferencing app, where a participant can multitask and be in multiple sessions at the same time; e.g., video-only in one session, audio and text messaging in another. Each session has a host who can manage participation. The host can add someone to the session by dragging their contact card into the session's window. The addition of a contact card to a window triggers the opening of a media connection to the new participant. To drop a participant, the host simply drags their contact card out of the window, which triggers the closing of the media connection to the participant.

Model. Using a model-view-controller architecture, one possible approach is as follows. The model holds state information, such as the sessions that are open and who is in which session. The model merely holds information: it does not open or close media connections.

View. The view displays session windows, which contain the contact cards for the participants in a session. The view gets sessions and participant information from the model. The view also detects gestures; e.g., it detects that a contact card has been dragged in or out of a session window. The view is dumb: it does not know the significance of dragging a card in or out of a window.

Controller. The controller interprets gestures and updates the model, as appropriate. When a card is added to a window, the view notifies the controller. The controller then opens a media connection for the added participant. It also sends a message to the model to update the state of the app; that is, to record that the session has a new participant. □

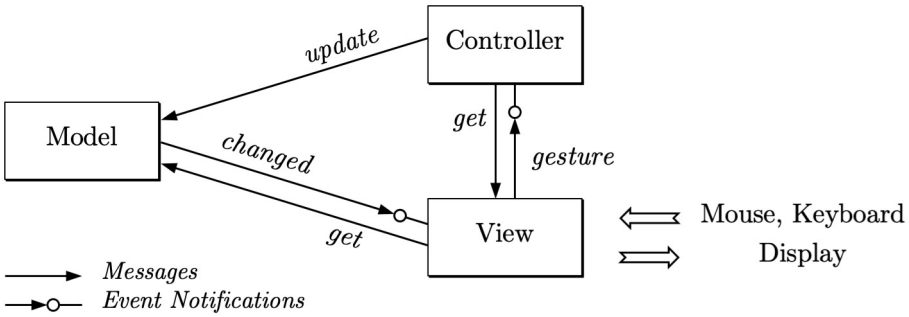


Figure 8.7: A Model-View-Controller architecture with a classic view that observes the model.

8.5.2 Selected MVC Architectures

The Basic MVC pattern does what patterns do: provide a core of a solution, as opposed to a complete solution. Here are some questions that it leaves unanswered:

- How exactly does a presentation synchronize with the model? Each presentation has a view-controller pair, so the two options are (1) the view leads and (2) the controller leads the synchronization.
- How do we test an implementation of an MVC architecture? Interactive user interfaces are notoriously had to test. This section includes a partial answer in the form of some suggestions.
- How is view-specific logic handled? For example of view-specific logic, consider a network map in which the links turn red if they are congested. Where do we put the decision about whether to change the color of a link. An MVC architecture in this section illustrates how this problem has been addressed before.

Classic Views Observe the Model

Classic views get their information directly from the model; see the architecture in Fig. 8.7. Specifically, classic views observes the model. When they are notified of a change, they call the model to get the latest information to display.

Example 8.9. The scenario in this example begins when the user doubles the height of a photo through the dialog view in Fig. 8.6. Again, we assume that the height and width of a photo are linked, so the width must double when the height doubles. The MVC architecture is as in Fig. 8.7.

The sequence diagram in Fig. 8.8 shows a sequence of messages and event notifications between three objects view, controller, and model. These objects

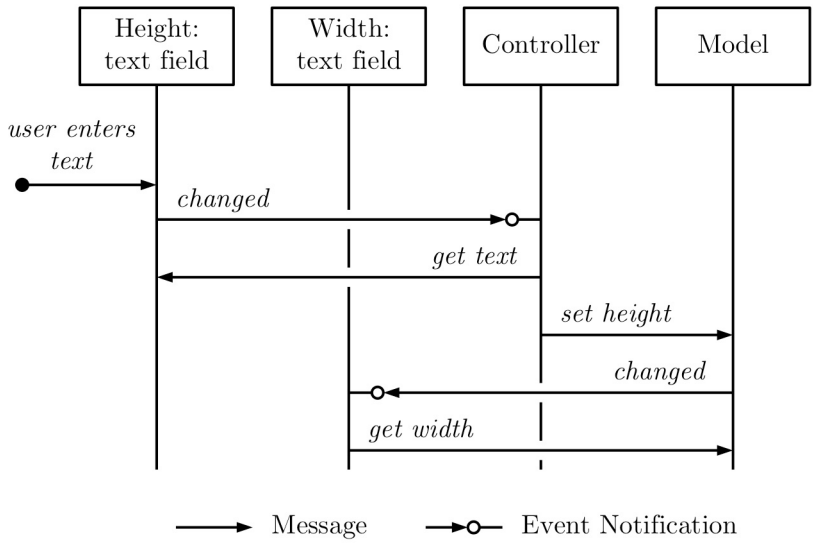


Figure 8.8: Detailed sequence diagram illustrating the flow of messages that begins when a user enters text in the height field. When the model changes, a classic view observes the change and updates its width field.

are represented by boxes and vertical lines. For simplicity we assume that the flow of control through the objects follows the flow of messages. ((In general, the model, view, and controller could execute in separate threads, with asynchronous messages between them.)

The scenario begins when the user enters text in the height field. The view raises an event to notify the controller. The controller sends a message to the view to get the text entered by the user in the height field. The controller then interprets the text as an integer 900 and sends a message to the model to double the size of the photo in the model. How the model doubles the photo object is not shown.

The model then notifies its observers that the photo object has changed. The view observes the event and sends a message to the model to get the width, which it renders in the width field. Note that when the height of a photo changes in the dialog view, the resulting change in the width comes through the model. □

With a classic view, the view and its controller are loosely coupled. Changes to a view come through the model.

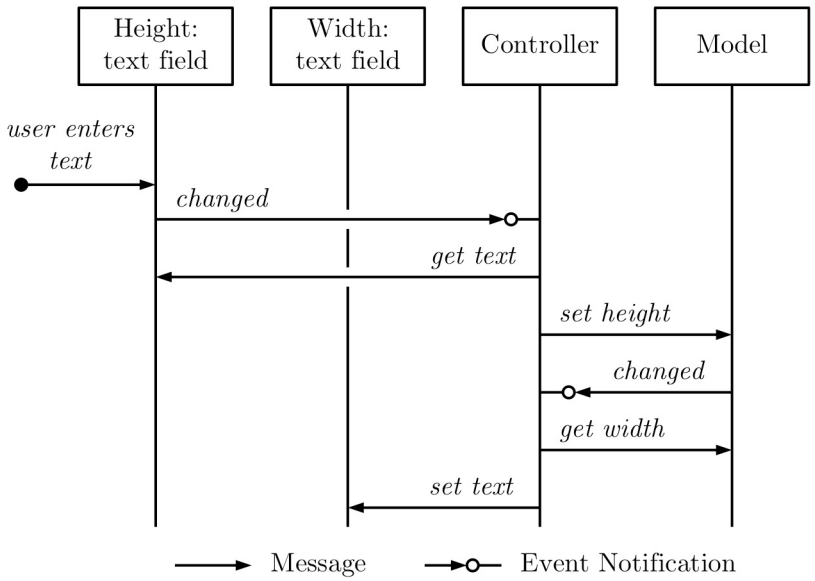


Figure 8.9: A humble view concentrates on detecting user gestures and rendering information. The controller does the rest of the work of synchronizing with the model.

Humble Views: Testing

The *humble* approach is to minimize the behavior of any object that is hard to test. Interactive user interfaces are hard to test since they do not lend themselves to automated ted testing (see Chapter 10 for unit test frameworks). It is hard to automate the comparison of what is rendered with what is expected on a display.

A *humble view* is limited to detecting actions and rendering information on a display. Humble views get the information they display through their controller. The controller is the observer of the model.

Example 8.10. The message sequence in Fig. 8.9 begins like the one in Fig. 8.8: the user enters text in the height field; the view notifies the controller; the controller gets the entered text; the controller sends a message to the model to change the height of the photo object.

With a humble view, the controller observes the model. When notified of a change, the controller retrieves the photo's height and width. It then tells the view what to render. □

The limited functionality of humble views aids testing. It helps to move behavior from the view to its controller, since controllers are easier to test

than views. Controllers can be more readily tested using automated unit-test frameworks.

With a jumble view, testing can be partitioned into two parts: (1) verifying that the view detects user actions correctly and renders information correctly; and (2) the right messages flow to and from the view. The first part—detecting and rendering—unavoidably belongs with the view. The burden of the second part—verifying the messages—can be eased by making the controller responsible for information flows to and from the view.

In summary, with humble views, the controller does as much of the work as possible. The controller observes the model and retrieves information for the view to render. Messages to and from the view can be intercepted at the controller for unit testing.

The hard part of testing a view is therefore reduced to verifying that views detect user actions and render information correctly.

View-Specific Logic and State Information

A complex user interface may involve more than a passive display of information from the model; it may involve some decisions and computations that are specific to a view. We begin with an example that illustrates view-specific decisions. Then comes the question: where do such decisions belong? They are specific to a view, not to the underlying object, so they do not really fit in model.

Example 8.11. A communications company maintains a network of links between sites across the country. It has engineers who monitor the network to ensure that data traffic is flowing smoothly. If a link is overloaded, they take action to reroute traffic around the overloaded link.

The engineers want a display of the network that highlights overloaded links. One option is to provide a list of such links. Another option is to show a network map that includes all the links, but highlights the overloaded ones by making them thicker and coloring them red.

In a model-view-controller architecture, the model holds the state of the network, including the links and their loads. The model also has logic to decide when a link is overloaded.

Decisions about how to highlight are view-specific, however. Thickening and coloring of links apply only to the network-map view. Thus, thickening and coloring of links do not belong in the model. □

The architecture in Fig. 8.10 introduces a logical component called a *presentation model* to hold view-specific logic and state information.[9] Logically, view-controller pairs become triples: view, controller, and presentation model. view and the controller interact with the presentation model as they did with the model. For the network map example, the presentation model holds the decision that overloaded links are to be thickened and colored red.

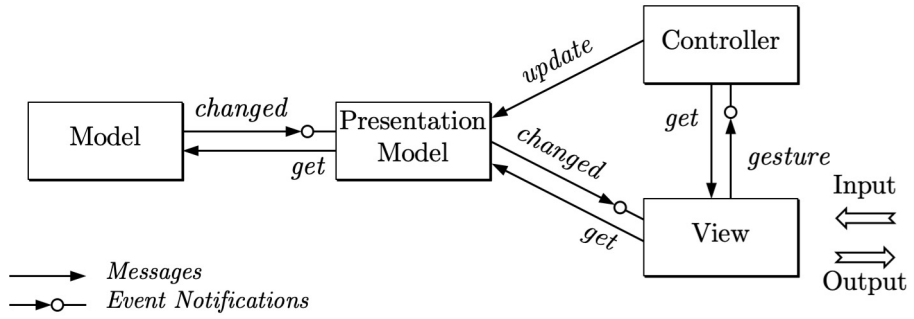


Figure 8.10: The presentation model is a logical component that holds view-specific logic and state information. In practice, the presentation model can be merged into the controller.

Note the emphasis above on presentation model being a "logical" component. In practice, the functionality of the presentation model can be merged into the controller, so we are left again with models, views, and controllers. For clarity and perhaps for testing, it helps to keep the presentation model as a distinct subcomponent of the controller.

8.6 Dataflow Pipelines and Networks

A *dataflow network* is made up of a set of independent components or processes that transform a stream of data as it flows through the network. The connections between the components represent the flow of data from one component to another. If the components are in a line, one after the other, the network is called a *dataflow pipeline* or simply *pipeline*.[10] The components of a dataflow network are often processes, so we use both terms "component" and "process," while discussing dataflows.

Example 8.12. The pipeline in Fig. 8.11 is a high-level view of a compiler. The two components of the compiler are represented by boxes. The input to the Syntax Analyzer is a stream of source code in some programming language. The output of the Syntax Analyzer is an intermediate representation of the source program. The intermediate representation then becomes the input to the Code Generator. The output of the Code Generator is a stream of machine instructions that constitute the the target code for the program.[11] □

Applications of dataflow networks include:

- Quickly assembled Unix or Linux pipelines that transform character streams.
- Extract-transform-load (ETL) processes that extract data from one database and reshape and reformat it so that it can be loaded into another database.

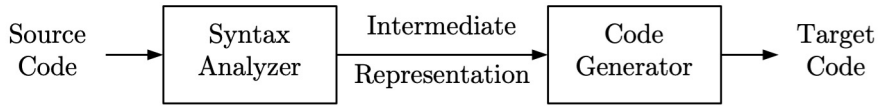


Figure 8.11: A two-stage pipeline for a compiler.

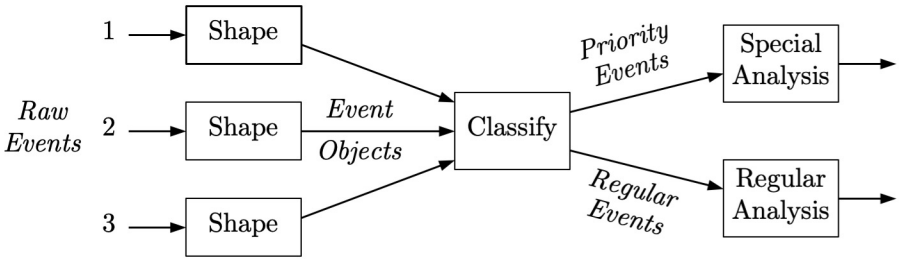


Figure 8.12: Streams of raw events are shaped into time-stamped event objects. The object streams are merged and then classified into high-priority and regular events for data analysis.

- Data mining of the massive streams of events that pour endlessly from search engines, video streaming services, sensors, and more.

Example 8.13. The dataflow network in Fig. 8.12 is inspired by a music streaming service that has users across the world. The app for the service generates an event each time a user searches for a song, requests a song, or takes some other action. The app sends a raw text description of the event to one of the service's gateways, which time-stamps and shapes the event into an object. The gateways send event objects to a classifier, which separates priority events from regular events. (Priority events include billing events.) Priority events are sent, reliable delivery, for special analysis. Regular events are sent, best effort, for regular analysis.

Analysis is used not only to respond to requests, but to make recommendations and maintain lists of popular songs.[12] □

8.6.1 The Dataflow Pattern

Context. The high-level goal is to assemble software applications from independently developed components. The dataflow pattern addresses a useful class of applications that transform streams of data. Unix pipelines are a special case—they transform streams of characters; see examples below.

Problem. Structure software applications that take as input one or more streams, and produce as output one or more data streams.

Core of a Solution. Define independent components, where (a) each component does one job well, and (b) the components work well together.

Combine the components into networks, where the output of one component becomes the input to the next. A component in the network may have multiple inputs and/or multiple outputs. If each component has one input and one output, the network is called a pipeline. □

On Windows, dataflow networks of components with multiple inputs or outputs can be implemented by using named pipes: "*Named pipes* are used to transfer data between processes that are not related processes and between processes on different computers."[13] Windows pipes can also be used to set up two-way communication between processes.

8.6.2 Unix Pipelines

Pipelines are named after the Unix *pipe* operator "|": if p and q are processes, then $p \mid q$ connects the output of p with the input of q. The pipe operator can be used to quickly assemble pipelines by writing expressions of the form

$$p \mid q \mid r \mid s$$

The linear syntax of such expressions is for readability and ease of use. (The underlying implementation can support duplication of streams, so the output of one process can become the input for more than one process.)

Example 8.14. The processes in the pipeline in Fig. 8.13 transform the lines in a document into a sorted list of words, one per line. Suppose that a document has two lines:

```
Omit needless words!
Omit redundant words!
```

The first process converts the document into a list of word occurrences, with each occurrence on a separate line. It does so by translating all non-alphabetic characters into newline characters. Thus, the translation of each blank starts a new line, and so does the translation of each exclamation point, "`!`". The resulting eight lines are shown between the first and second boxes in Fig. 8.13.

The second process translates uppercase letters into lowercase. The two occurrences of `Omit` are therefore each translated into `omit`. The third process in the pipeline sorts the lines. The last process removes adjacent duplicate lines.

The output of the last component in Fig. 8.13 is a sorted list of words, preceded by an empty line. We could add another component to remove the empty line. □

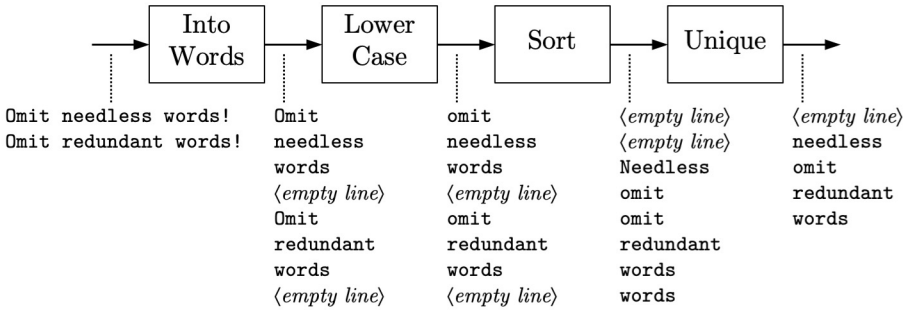


Figure 8.13: Pipeline for making a list of words in a document.

Example 8.15. For completeness, here is the Unix pipeline that corresponds to the transformations in Fig. 8.13:

```
tr -C a-zA-Z '\n' | tr A-Z a-z | sort | uniq
```

The Into Words box is implemented by using the translate command, `tr`, with suitable parameters:

```
tr -C a-zA-Z '\n'
```

The flag `-C` represents "complement," so the command translates non-alphabetic characters to newline characters, represented by `'\n'`. In other words, the command translates any character that is neither a lowercase letter nor an uppercase letter into a newline character.

The translation from uppercase to lowercase letters is done by

```
tr A-Z a-z
```

The `sort` command sorts its input and `uniq` removes adjacent duplicate lines from its input. If desired, the empty line can be deleted by adding a final phase to the right end of the pipeline. □

8.6.3 Unbounded Streams

A data stream is said to be *bounded* if it has a beginning and an end. The input stream to a compiler is bounded: it consists of a source program, which has a beginning and an end. When a Unix pipeline is applied to a file, the character stream that flows through the pipelines is bounded.

A stream is said to be *unbounded* if it has no end. The streams of data from business and entertainment transactions are unbounded for all practical purposes.

Filtering and Aggregating Transformations

With unbounded streams, transformations such as counting and sorting are problematic. Why? Consider the problem of sorting the unbounded stream

$$1, -2, 3, -4, 5, -6, ...$$

What is the smallest number in this stream? What is the largest? How many times do even numbers appear in this stream? Odd numbers?

Example 8.16. Several billion videos are viewed on YouTube every single day. The stream of usage data from these videos is unbounded. Yet, advertisers want to know how many times a video is viewed. The following quote from Google is about video services in general:

> "Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. ... They want all of this information as quickly as possible, so that they can adjust ... in as close to real time as possible."[14]

The solution is to carve up an unbounded stream into buckets or windows, where each window contains a bounded stream. □

Informally, transformations on individual elements carry over readily from bounded to unbounded streams; transformations that aggregate some property of an entire stream are problematic.

- *Filter, elementwise.* The processes in a pipeline are sometimes called *filters*. Filters that transform individual elements or finite sequences of elements work for both bounded and unbounded streams. Three of the four filters in Fig. 8.13 are elementwise—sorting is not. Elementwise filters are also referred to as *map* functions.
- *Aggregate or Accumulate.* Transformations that compute some property of an entire stream are problematic for unbounded streams. Aggregators are also referred to as *reduce* functions. The count of viewings of a specific videos accumulates as more and more of the usage data is read: the count is initially 0 and is incremented with each viewing. The count is well defined for any finite segment of the usage-data stream, but is potentially unbounded for an unbounded stream.

The names map and reduce, for elementwise filters and aggregators, are motivated by the *map* and *reduce* functions in Lisp and other functional programming languages.[15]

Windowing

Slicing a data stream into finite segments is called *windowing*; the segments are called *windows*. Spotify, the music service, has been using hourly windows for

usage events. Each window is finite, so it can be treated as a bounded stream. In general, windows can be of any size. Furthermore, for different kinds of analysis, the same data stream can be sliced in different ways.

8.6.4 Big Dataflows

The significance of dataflow networks has grown with the rise of big data streams, such as those from music and video services. But, scale and geographic distribution introduce engineering issues:

Parallelism

Elementwise filtering lends itself to parallel execution by multiple processors. The elements of the stream are transformed independently of each other, so they can be split into sub-streams that are transformed by separate parallel processors. The popular MapReduce programming model divides an application into two stages: a map stage for elementwise filtering and a reduce stage for aggregation. The experience with MapReduce is summarized by:

> A large class of massive-scale "computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time."[16]

Apache Hadoop is a reverse-engineered version of MapReduce.

Reliability

If the processes in a pipeline are physically running entirely within the same datacenter, then their designer can ignore the possibility of data loss between process in the pipeline. Data transfers within a datacenter can be safely assumed to be reliable. (This observation applies to any dataflow network, not just pipelines. We talk of pipelines to avoid any confusion between network as in "dataflow network" and network as in a "data communications network" like the Internet.)

With geographically distributed pipelines, we need to look more closely at the connectors between processes in a pipeline. The processes in a pipeline have a producer-consumer relationship, since the output of one process becomes the input to the next process in line. If the producer is faster than the consumer, the producer's output needs to be buffered until the consumer is ready for it. The connecters between processes in a pipeline therefore represent queues.

Who ensures the reliability of queues? Do queues *have* to be reliable? Reliability across a network comes with delays. Lost data needs to be transmitted, as we saw in the discussion of reliable (TCP) and best effort (IP) in Section 8.2.3. Reliable delivery is needed for applications such as billing; however, best effort or even sampling data may suffice for a poll of customer preferences.

Example 8.17. Spotify switched from best-effort to reliable queues for their Event Delivery pipeline. The earlier design assumed that the elements of the data stream could be lost in transit. The producer processes were responsible for queueing their output until their consumers acknowledged receipt of transmitted elements. This design worked, but it complicated design of the producers.

The later design is based on reliable queues, which simplifies the design of the producers. The processes in the pipeline can therefore concentrate on transforming inputs to outputs, without worrying about buffers and queues.[17]
□

Network Bandwidth

With missive scale data streams, the capacity of a connector becomes an issue. Can a connecter between two processes handle the volume of the flow between the processes? The scale or volume of a data stream becomes an issue if the two processes are in two separate datacenters connected by a network. Network bandwidth or carrying capacity is limited.

Pipelines need to be designed to conserve data transfers across a network. Is a large stream of raw data being transferred across a continent, only to be reduced to a count or an average on the other side? If so, consider reducing the large-scale raw stream at the source and transferring the much smaller-scale reduced stream to the destination.

8.7 Connecting Clients with Servers

Conceptually, clients request services from servers, and servers reply with the results of processing the client requests. A web server might reply with a web page, a map server with a map, a music server with an audio stream, an academic server with courses that match given criteria, and so on. Clients and servers typically connect over a network.

Here, clients and servers are logical (software) components. If the need arises, we can use the terms "physical client" and "physical server" for the related pieces of hardware.

This section begins with the client-server pattern: clients connect directly with a known server. In practice, there are too many clients for a single server to handle. The client load is distributed across multiple servers. Instead of connecting directly with a specific server, a client is dynamically matched with a server. Dynamic matching is done by adding components like name servers and load balancers. The broker pattern provides a component that provides the functionality needed to dynamically match clients with servers. See Fig. 8.14 for brief notes on the client-server and broker patterns.

As we shall see, dynamic matching of clients and servers is also useful for continuous deployment, where new or updated software components can be added to a production

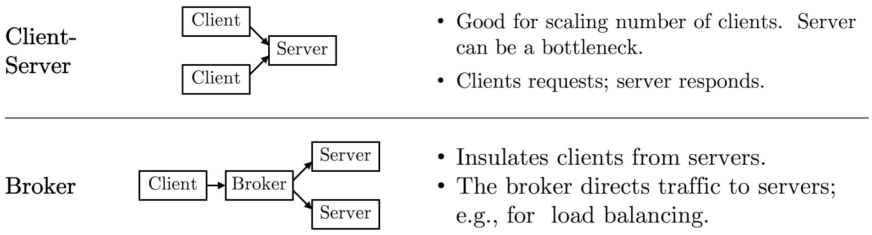


Figure 8.14: Ways of connecting clients and servers.

8.7.1 The Client-Server Pattern

Context. The sequence of patterns in this section builds up to the general problem of matching clients with servers.

Problem. Connect multiple clients directly with a specific server.

Core of a Solution. The connection in this pattern is many-to-one: many clients connect with one server. The clients request services; the server replies with a response.

The clients know the server's identity, so they know how to send a request. The request contains information about the client, so the server knows where to send a reply. With the reply, the connection between client and server is closed. A new request opens a new connection. The server does not initiate a connection with the client.

Typically, the request-reply interaction is synchronous. That is, upon sending a request, the client waits until a reply arrives. Alternatively, the interaction between client and server can be asynchronous. In the asynchronous case,

> the client sends a request; the server acknowledges the request; the server works on a response; the server replies with a response.

After sending an asynchronous request, the client can continue to work. □

A Dynamic Variant of Client-Server

The architecture in Fig. 8.15 goes beyond the client-server pattern in three ways:

- Clients use a name as keys to look up the location(s) of the servers.
- At the location, a load balancer assigns a server to handle the client request. (Large systems need multiple servers to handle the load from large numbers of clients.)

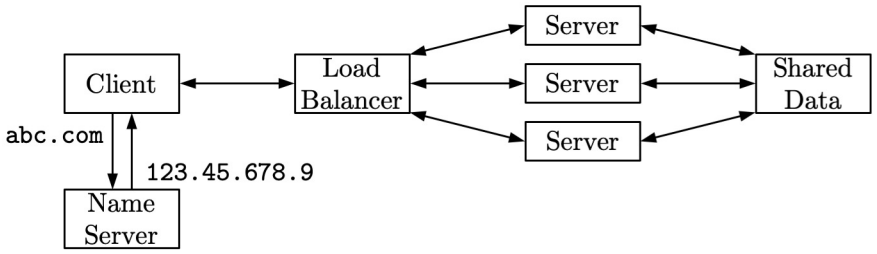


Figure 8.15: A dynamic variant of a client-server architecture.

- All client-related data is held in a persistent data store shared by the servers.

With the architecture in Fig. 8.15, a request-reply interaction goes as follows. The client queries a name server, using a name (e.g., `abc.com`) to look up an address (e.g., `123.45.678.9`) for the location of the servers. The client then sends a request to the address it looked up. At that location, the request goes to a load balancer. The load balancer picks an available server and forwards the client request to that server. The server processes the request and replies to the client.

A key point to note is that servers need to be stateless. That is, the servers must not keep any client-related information. The reason is that the next client request could be assigned to a different server.

So, any client information is kept in a persistent shared data store, accessible by all the servers. Client-related information includes any preferences, any history of past interactions, any data from the current interaction that might be helpful for future interactions—all such information belongs in the shared data store.

Deploying Test Servers

The architecture with the name server (Fig. 8.15) scales readily to multiple locations. Suppose that for `abc.com` there are two locations, with addresses `123.45.678.9` and `135.79.246.8`. (The approach works for more locations as well.) For key `abc.com`, the name server returns a list with both these addresses. The client can then go down the list, one by one. If one address times out, the client tries the next one on the list.

The architecture with the name server also allows new or updated software to be eased into production. Suppose that the servers are running version m and we have just created a new version n of the software. Version n has been fully tested in development and it is time to deploy it in a production environment. Caution demands that the new version n be deployed in stages, leaving open

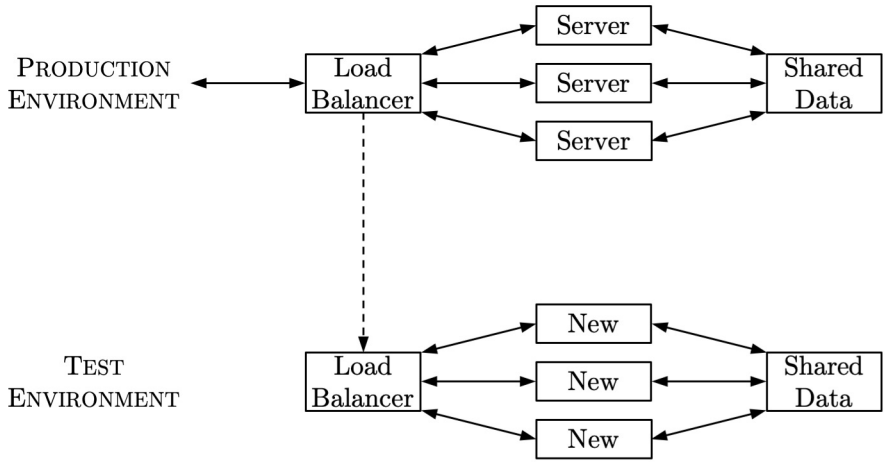


Figure 8.16: Deploying a new server software in a test environment. The dashed arrow indicates that the production load balancer sends copies of all client requests to the test environment. The boxes labeled "New" represent the new version of the server software.

the option of rolling back to the previous version m, if there is a hitch with the new version n at any stage. Here are some possible stages:

1. *Test Environment.* Run the new version on a copy of the incoming stream of client requests. In Fig. 8.16, the production load balancer (above) passes a copy of each client request to the load balancer (below) in the test environment. The test environment mirrors the production environment, except that it runs the new version n.

2. *Partial Production.* If the new version performs well in a test environment it can be put into production, but The new version runs in a separate location, with its own address. The name server directs a small percentage of the client traffic to the new version. The flow of traffic is still to the location with the old version m. If requests for the same client can be directed to either version m or version n of the software, then both versions m and n must share the persistent data store.

3. *Trial Period.* After successfully handling some of the real client load, the new version n is ready for full production; that is, ready to handle all client requests. During a trial period, the earlier version m remains on standby.

The time frames for the above stages are measured in hours or days.

8.7.2 The Broker Pattern

Context. We continue with the theme of large numbers of clients seeking services that are provided by servers. The goal is to connect clients with servers, dynamically. The architecture in Fig. 8.15 provides a solution using building blocks: name servers and load balancers. The broker pattern uses these same building blocks in a way that is not visible to clients.

Problem. Ensure that clients get the services they want through a service interface, without needing to know about the infrastructure (including servers) that is used to provide the services.

Core of a Solution. Introduce an intermediary called a *broker* that clients deal with. Clients send the broker a service request, and the broker does the rest.

The broker uses a name server and balancer to select a server to process the client request. Once the server prepares a response, the broker forwards the response to the client.

(The following observation is about servers, not about brokers. It is included because it applies to an overall solution that includes brokers. If servers are drawn from a pool, the servers need to be stateless. As in the architecture in Fig.‘8.15, all client-related information belongs in a shared persistent data store.)

In this description, name service and load balancing are logical functions. They can be part of the broker.

The broker may also include a client proxy. A *proxy*, P, acts on behalf of another component, A. All communication intended for component A go through the proxy P. In a broker, the role of a client proxy is to manage the communication protocol for receiving and sending messages to the client.

Cautions With Brokers. All communication between clients and servers flows through the broker, which brings up some concerns:

- *Performance Bottleneck.* The broker needs to have enough capacity to handle the expected client load. The silver lining is that load balancing can aid performance because it can direct client requests to the least busy server.
- *Latency.* The forwarding of communications between clients and servers adds latency (delay) compared to direct communication between clients and servers. The benefits of dynamically matching clients and servers outweighs the cost of latency.
- Single Point of Failure. The broker needs to be designed for fault tolerance, with respect to both hardware and software faults. Fault tolerance is a topic in its own right. The basic idea is to have a primary and a backup for each critical component. If a primary does the work. The backup takes over if it detects that the primary has failed.

- *Security Vulnerability.* Since all client-server communications flow through the broker, it represents a security vulnerability. The broker therefore needs to be designed to resist a security attack. It also needs to be protected by network security measures.

8.8 Families and Product Lines

When there are several versions of a product, the versions surely share some common properties. The ways in which they vary from each other must also be bounded. This idea extends to a set of products.

A *software product family* is a set of products where it is worth

- first studying the common properties of the set and
- then determining the special properties of the individual family members.[18]

A *software product line* is a set of products that are specifically designed and implemented as a family. The annual Software Product Lines Conference has a Hall of Fame that honors organizations with commercially successful product lines. The application domains for the honorees include software for automotive systems, avionics, financial services, firmware, medical systems, property rentals, telecommunications, television sets, and training,[19]

The common properties of a family are called *commonalities* and the special properties of the individual family members are called *variabilities*. Using the terminology of patterns, the family follows a pattern; the family's commonalities and variabilities are part of the pattern's core of a solution.

Product lines arise because products come in different shapes, sizes, performance levels. and price points, all available at the same time. Consider iOS and Android versions of an app that runs on smartphones, tablets, and desktops, not to mention languages from different regions. Without a family approach, each version would need to be developed and maintained separately. A family approach can lead to an order of magnitude reduction in the cost of fielding the members of the family.

Example 8.18. HomeAway, a startup in the web-based vacation home rental market, grew quickly through acquisitions. Each acquired company retained the look and feel of its website.[20]

HomeAway's first implementation approach was to lump the systems for the various web sites together, with conditionals to guide the flow of control. This umbrella approach proved unwieldy and unworkable due to the various websites having different content management, layouts, databases, and data formats.

The second approach was to merge the various systems onto a common platform, while still retaining the distinct look and feel of the different websites. This approach had its limits:

> "A thorough code inspection eventually revealed that over time 29 separate mechanisms had been introduced for managing variation among the different sites."
>
> "Testing ... impoverished though it was, discovered 30 new defects every week—week after week—with no guarantee that fixing one defect did not introduce new ones."

The company then turned to a software-product line approach. Within weeks, the product-line approach paid for itself. The software footprint went down, quality went up, deployment times went down. Modularity meant that changes to one site no longer affected all the other sites. □

8.8.1 Software Architecture and Product Lines

Software architecture plays a key role in product-line engineering. To the guidelines for defining modules in Section **??** we can add:

- Address commonalities before variabilities, when designing modules.
- Hide each implementation decision about variabilities in a separate module. Related decisions can be grouped in a module hierarchy.

One of HomeAway's goals for a product line approach (see Example 8.18) was to make the cost of implementing a variation proportional to that variation, as opposed to the previous approaches, where the cost was proportional to the number of variations.

Support for the significance of architecture in product-line-engineering comes from SEI's experience with helping companies implement product lines:

> "The lack of either an architecture focus or architecture talent can kill an otherwise promising product line effort."[21]

8.8.2 Economics of Product-Line Engineering

Product-line engineering requires an initial investment. Here are some areas for investment: identify commonalities and variabilities; build a business case that encompasses multiple products; design a modular architecture that hides variabilities; create test plans that span products; and train developers and managers. One of the keys to the success of product-line engineering at Bell Labs was a small dedicated group that worked with development groups on their projects.

Management support is essential. Product-line engineering projects that have lacked management support or initial investment have failed to deliver the promised improvements in productivity, quality, cost, and time to market. The schematic in Fig. 8.17 illustrates the economic tradeoffs.[22] With the traditional approach, each family member is of built separately, so costs rise in proportion to the number and complexity of the family members. For simplicity, the schematic shows costs rising linearly with the number of family members.

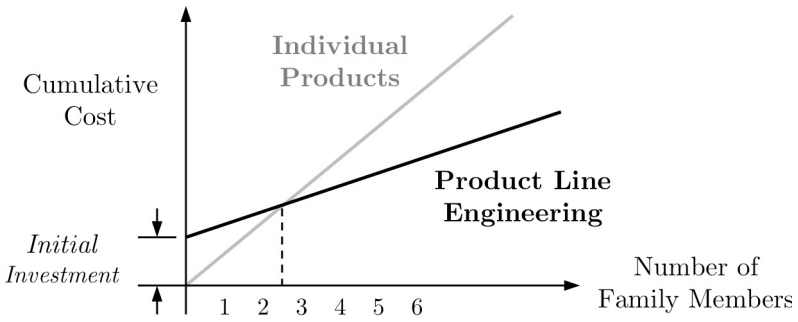


Figure 8.17: Schematic of the economics of software product lines.

With product-line engineering, there is an initial investment, which adds to the cost of the first product. The payoff begins as more products are delivered, since the incremental cost of adding a product is lower. Based on the Bell Labs experience, the crossover point is between 2 and 3 family members. The greater the number of family members, the greater the savings, past the crossover point.

8.9 Conclusion

Exercises for Chapter 8

Exercise 8.1. Relate the following excerpt about beds and tables to architectural patterns:

> " there are beds and tables in the world—plenty of them, are there not?
>
> "Yes.
>
> "But there are only two ideas or forms of them—one the idea of a bed, the other of a table.
>
> "True.
>
> "And the maker of either of them makes a bed or he makes a table for our use, in accordance with the idea ...."

This excerpt is from Plato's *The Republic* (circa 380 BCE).[23]

Chapter 9

Static Validation and Verification

> "While the classical definition of product quality must focus on customer needs ... removing software defects consumes such a large proportion of our efforts that it overwhelms everything else."
>
> — *Wattts S. Humphrey, from one of his columns while he was at the Software Engineering Institute.*[1]

9.1 Introduction

The combination of reviews, static analysis, and testing is highly effective for defect detection. By themselves, the individual techniques are much less effective.[2] Reviews and static analysis are static techniques; they check code without running it. In fact, reviews can be used not only for code, but for artifacts that cannot be run; e.g., designs and documents. Testing is dynamic; it requires code that runs.

This chapter deals with static techniques; see the upper row in Fig. 9.1. The lower row is for testing, which is covered in Chapter 10. The definitions of validation and verification are repeated here, for convenience:

- *Validation: Build the Right Product.* The purpose of validation is to check whether an artifact meets its user requirements; that is, does it do what users want?

	VALIDATION	VERIFICATION
STATIC	Architecture Reviews	Code Reviews Static Analysis
DYNAMIC	Testing	Testing

Figure 9.1: Selected techniques for validation and verification and.

- *Verification: Build the Product Right.* The purpose of verification is to check whether an artifact meets its specification; that is, is it implemented correctly?

A *review* is a process for examining a software artifact for comments, defects, or approval. Reviews range from formal inspections that take days or weeks to open-source code reviews that are completed in hours. Let us use the term *inspection* for a formal review by a carefully selected group of independent experts who meet, physically or virtually, to provide their collective feedback. Open-source code reviewers not only look for defects, they suggest fixes. Many companies have "review-before-commit" policies for code.[3]

Section 9.2 explores guiding principles for architecture and design reviews. Options for conducting an architecture review are discussed in Section 9.3.

Code can be both reviewed by experts and analyzed by automated tools. Code reviews focus on design and style; see Section 9.4. Automated tools perform static analysis, which checks code without running it. Static-analysis tools flag violations of pre-defined rules and guidelines. They strike a balance: too many false warnings and developers will ignore the checker; too few few warnings and some severe defects might slip through. Section 9.5.

The list of techniques in Fig. 9.1 is a a good starting point for a quality improvement program. The list is not intended to be complete. For example, missing from the list are specialized techniques like model checking, which is very useful for finding concurrency failures such as deadlocks.

9.2 Architecture Reviews

The primary purpose of an architecture review is to

a) clarify the goals of the proposed architecture and

b) confirm that a solution based on the architecture will meet those goals.

Figure 9.2: Guiding principles for architecture reviews.

The goals of an architecture follow from user requirements, so architecture reviews are a validation technique.

Far too often, either the architecture does not adequately address the goals or the goals are not completely or clearly defined.[4] In addition a review may uncover other issues related to the project, such as the lack of management support or the inadequacy of the team's tools and domain knowledge.

9.2.1 Guiding Principles for Architecture Reviews

This section explores the guiding principles in Fig. 9.2. The principles are based on over two decades of experience with architecture reviews.[5] See Section 9.3 for how to conduct a review. The full benefits of reviews are realized when they follow a formal process. The guiding principles are also helpful for informal peer reviews, where developers go over each other's work.

The Review is For the Project Team's Benefit

The purpose of a review is to provide a project team with objective feedback. It is then up to the project team and its management to decide what to do with the feedback. The decision may be to continue the project with minor changes, to change the project's direction, or even to cancel the project.

Reviews have been found to be cost effective for both projects that are doing well and for projects that need help. Reviews are not meant to be an audit on behalf of the project's management. They are not for finding fault or assigning blame.

Since the project team will be the one to act on reviewer feedback, members of the development team have to participate in the review. The team's participation also helps to build trust between the team and the reviewers, which

increases the likelihood that the team will act on the feedback from the review.

The Expert Reviewers are Independent

For the review to be objective, the reviewers need to be independent of the project and its immediate management. For the review to be credible, the reviewers need to be respected subject-matter experts.

The independent experts may be either from outside the company or from other parts of the company. A side benefit of drawing reviewers from other parts of the company is that (a) they spread best practices to other projects and (b) the company builds up a stable of experienced reviewers.

The Project has a Clear Problem Definition

During requirements development, user needs are identified and analyzed, with the intent of defining the problem to be solved. Revisiting an example from Section 3.3, consider the following need and some options for a problem definition:

Customer Need:	Listen to music
Option 1:	Offer songs for purchase and download
Option 2:	Offer a free streaming service with ads

During an architecture review, the independent experts respectfully provide feedback on the clarity and completeness of the problem definition.

Issues can arise with the problem definition if there are multiple stakeholders with conflicting needs or goals. For example, one stakeholder may be fanatical about keeping costs low, while another is equally passionate about maximizing performance. As another example, conflicts can arise when balancing convenience and security.

The Architecture Fits the Problem

The reviewers confirm that the architecture will provide a reasonable solution to the problem. Developers often focus on what the system is supposed to do; that is, they focus on the functional requirements for the system. They tend to pay less attention to quality attributes, such as performance, security, and reliability. Reviewers therefore pay particular attention to quality attributes. Early reviews help because quality attributes need to be planned in from the start. Otherwise, they can lead to redesign and rework later in the project.

For example, with iterative and agile processes, early iterations focus on a minimal viable system, in order to get early customer feedback on the basic functionality. Special cases, alternative flows, and error handling get lower priority and are slated for later iterations. Concerns about quality attributes may not surface until late in the project. An early architecture review can prevent later rework.

Problem Definition

- How will the customer benefit?
- What is the rationale for choosing this opportunity?

System Architecture

- What are the prioritized requirements?
- What are the main components of the system? How do they support the basic scenario?
- What is the desired performance? Scale? Availability?

Team

- Has the team built something like this before?
- Is the team co-located or distributed?

Constraints and Risks

- Are there any business constraints? Time to market?
- Are there any ethical, social, or legal constraints?
- What are the risks associated with the external technology and services?

Figure 9.3: A short checklist for an architecture discovery review.

The Project has a System Architect

Projects that are important enough to merit a formal architecture review are important enough to have a system architect. The "architect" may be a person or a small team. The reviewers rely on the architect to describe the architecture and provide the rationale for the design decisions.

The reviewers also assess the team's skills. Does anyone on the team have prior experience with such a system? Are the tools new or known to the team?

9.2.2 Discovery, Deep-Dive, and Retrospective Reviews

The focus of a review varies from project to project and, for a given project, from stage to stage in the life of a project. The following three kinds of reviews are appropriate during the early, middle, and late stages of a project, respectively:

- A discovery review for early feedback.
- A deep dive for an evaluation of a specific aspect of the architecture.
- A retrospective for lessons learned.

An *architectural discovery review* assesses whether an architectural approach promises a suitable solution. A discovery review can begin as soon as preliminary design decisions are made, before an architecture fully exists. The reviewers focus on the problem definition, the feasibility of the emerging design, and the estimated costs and schedule. A short checklist of questions for a discovery review appears in Fig. 9.3.

The benefit to the project team of an early discovery review is that design issues are uncovered before the architecture is fully developed.

An *architectural deep dive* evaluates the requirements, the architecture, and high-level design of either the entire project, or of some aspect of the project. The project team may identify specific areas for feedback. The following is a small sample of focus areas from actual architecture reviews:[6]

user experience	performance	interoperability
user interface	security	software upgrades
disability access	reliability	deployment

Deep dives are conducted during the planning phase, before implementation begins.

An *architectural retrospective* is a debriefing to identify lessons learned that could help other projects. The reviewers ask what went especially well and what did not. Retrospectives are useful for sharing best practices and recommendations for problems that other projects might encounter.

9.3 Conducting Software Inspections

Roughly speaking, an inspection is a formal review by a groups of experts who provide feedback on a project or architecture or other significant software artifact. Inspections can take days. The code reviews in Section 9.4 are applied to relatively short segments of code; the review is completed within hours. The two approaches complement each other and can be used together as part of a quality improvement program.

Inspections are widely used since they were introduced at IBM in the 1970s; their benefits outweigh their costs. IBM experienced "substantial" improvements in software quality and productivity.[7] For a dramatic example of their use, consider the flight-control software for a 2012 soft-landing on Mars: Quality was a prime concern, so the development team conducted 145 inspections that produced 10,000 comments that were individually tracked and addressed.[8]

The main phases of an inspection has not changed since they were introduced. The boxes in Fig. 9.4) represent the four phases:

a) planning, including reviewer selection;

b) individual preparation by reviewers;

c) moderated group examination of materials; and

d) Follow up rework by the project team to address reviewer feedback.

For convenience, this section address the question of how to conduct an inspection by starting with traditional inspections, also known as *Fagan inspections*. The goals of the phases in a traditional inspection appear on the left in Fig. 9.4. The section then explores the questions on the right in the figure, to indicate possible improvements.

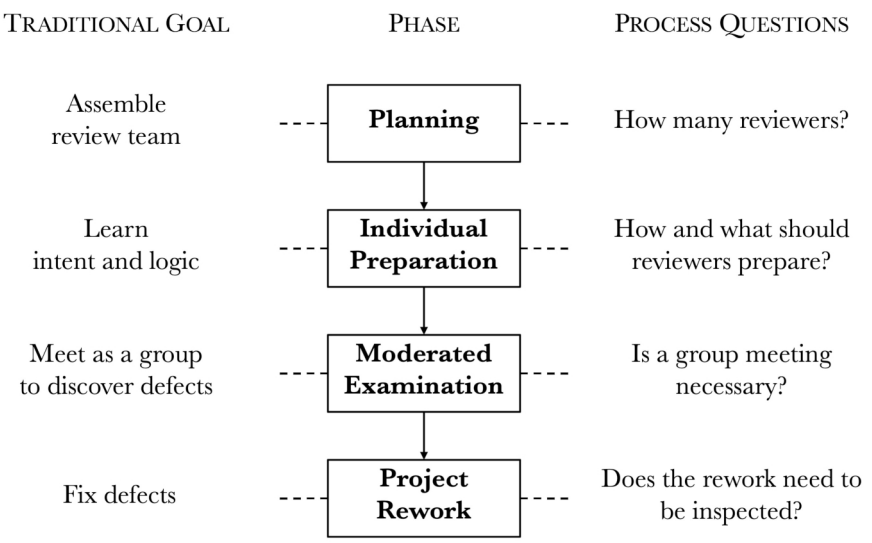


Figure 9.4: Phases of a software inspection. Goals from a traditional inspection are on the left. The questions on the right are based on empirical studies.

9.3.1 Traditional Inspections

In a traditional Fagan inspection, the main event is a group meeting to detect and collect defects. The group meeting corresponds to the moderated examination phase in Fig. 9.4. The earlier phases—planning and preparation—are simply to prepare the reviewers, so they will be effective in the moderated group meeting.[9] (Later studies have questioned the role of the group meeting, as we shall see below. Changes to the goal of the group meeting affect how reviewers prepare; they do not affect the basic flow of an inspection.)

The basic flow of a traditional inspection appears in Fig. 9.5. The phases of the flow are explored below.

Screening Projects for Inspection

Since the purpose of an inspection is to provide the project team with objective feedback, the request for an inspection must come from the project team. They are the ones to benefit from and act on the findings. As part of the request, the project team may indicate specific areas of focus for the inspection; e.g., usability, security, or reliability.

The inspection is not for assessing performance or for assigning blame. If the project team is pressured into having an inspection or is resistant to acting on the findings, the inspection could turn out to be a waste of everyone's time.

Companies typically have a screening process to prioritize requests for in-

Planning
A project team requests an inspection.
The moderator assembles a team of independent reviewers.
The moderator confirms that the materials meet entry criteria.

Overview and Preparation
The moderator spells out the objectives.
The author provides an overview of the materials.
The reviewers study the intent and logic individually.

Group Meeting
The moderator facilitates and sets the pace.
The reviewers examine the materials for defects.
The reviewers conclude with preliminary findings.

Rework and Follow Up
The reviewers compile a report with significant findings.
The author reworks the materials to fix defects.
The moderator verifies that that issues are addressed.

Figure 9.5: The basic flow of a traditional software inspection.

spections. Inspections have a cost: they require a time commitment by the reviewers and the project team. Screening is based on the perceived cost effectiveness of an inspection.

Roles in a Traditional Inspection

The main roles associated with an inspection are as follows:

- *Moderator*. The moderator organizes the inspection and facilitates group interactions to maximize effectiveness. Moderators need special training; they need to be objective. Hence they must be independent of the project and its immediate management.
- *Author*. The author may be a person or a small team. The author prepares the materials for review and answers questions about the project.
- *Reviewer*. Reviewers need to be independent, so they can be objective. They can be drawn from other projects within the same company. See also the comments about reviewers in Section 9.2.1.

The moderator role can be split into two: organizer of the inspection and moderator of the group interactions. Similarly, the author role can be split into two: author of the artifact and reader who paraphrases the content to be reviewed during group meetings.

The Planning Phase

The moderator assembles a team of independent reviewers. The moderator also ensures that the project team provides clear objectives and adequate materials for inspection. For example, for an architecture review, the project must provide a suitable architectural description. For a code inspection, the code must compile without syntax errors.

Overview and Individual Preparation

Individual preparation by the reviewers may optionally be preceded by a briefing session to orient the reviewers. During the briefing, the project team provides an overview of the project, drawing attention to the areas of focus for the inspection.

The reviewers then work on their own to prepare for the group meeting. While they may discover defects during preparation, the emphasis in this phase of a traditional review is on understanding—the group meeting is for defect detection and collection.

The Group Meeting

Ideally, the group meeting is face-to-face. The moderator sets the pace and keeps the meeting on track. The entire review team goes over the materials line-by-line to find defects. The detected defects are recorded: the meeting is for finding defects, not for fixing them. At the end of the review, the reviewers may confer privately and provide preliminary feedback to the project team.

Inspection meetings are taxing, so the recommended length of a meeting is two hours. Meetings for complex projects may take multiple day.

Rework and Follow Up

After the group meeting, the reviewers prepare a report with significant findings. The report classifies issues by severity: major issues must be addressed for the project to be successful; minor issues are for consideration by the project team. Finally, the moderator follows up to verify that the reported issues get addressed. The issues identified by the reviewers may include false positives—a false positive is an reported defect that turns out not to be a defect on further examination. The author must respond to all issues, even if it is to note that no rework is needed.

9.3.2 What Makes Inspections Work?

Software inspections have been studied extensively since they were introduced several decades ago. The cost of an inspection rises with the number of reviewers: how many are enough? A group meeting causes delays: is a meeting necessary? The rest of this section considers some questions about the process for conducting inspections.

How Many Reviewers?

The number of reviewers depends on the nature of the inspection: with too few, the review team may not have the required breadth of expertise; with too many, the inspection becomes inefficient. The fewer reviewers the better, not only for the cost of the reviewers' time, but because it takes longer to coordinate schedules and collect comments from the reviewers.

A typical inspection may have three to six reviewers. For code inspections, there is evidence that two reviewers find as many defects as four.[10]

Is a Group Meeting Really Necessary?

The group meeting is the main event of a traditional inspection. Reviewers are instructed to study the materials prior to the meeting. The meeting is where defect detection is expected to take place.

Experiments at Bell Labs in the 1990s questioned the necessity of a group meeting. In one study, 90% of the defects were found during individual preparation; the remaining 10% were found during the group meeting.[11] This data argues against group meetings. Such meetings are expensive. Schedule coordination alone can take up a third of the time interval for an inspection.[12]

How Should Reviewers Prepare?

Reviewers are often given checklists or scenarios to guide their individual preparation. Such guidance is to avoid two problems: duplication of effort and gaps in coverage. Duplication of effort occurs when multiple reviewers find the same defects. Gaps in coverage occur if defects remain undiscovered: no reviewer finds them.

In one study, reviewers who used scenarios or use cases were more effective at finding defects than reviewers who used checklists.[13]

9.4 Code Reviews

Code reviews have changed along with the tools and methods of software development. Instead of checking for defects, the primary motivation has changed to checking for intent—to checking that the code is readable and that it can be trusted to do what it is intended to do.[14]

Code reviews remain relevant, however. People are better than automated tools at getting to the root cause of a problem and in making judgements about design and style.

9.4.1 What has Changed?

Many of the consistency checks that were once done by human inspectors have been automated. Changes in programming languages and compilers have eliminated the need for questions like[15]

	Traditional Inspections	Open-Source Code Reviews
Frequency	per Phase	per Commit
Code Size	Tens of lines	Hundreds of lines
Reviewers	Independent (3-6)	Invested (1-3)
Goal of the Review	Detect defects	Detect and Fix defects
Meet	Face-to-Face	Asynchronously
Elapsed Time	Days	Hours

Figure 9.6: Differences between traditional inspections and open-source code reviews.

> "Have all variables been explicitly declared?"
>
> "Are there any comparisons between variables having inconsistent data types (e.g., comparing a character string to an address)?"

Static program analysis (see Section 9.5) eliminates the need for

> "Is a variable referenced whose value is unset or uninitialized?"

Run-time checking can address questions like

> "When indexing into a string, are the limits of the string exceeded?"

Design and style questions, however, still require human review and may never be automated. For example, consider

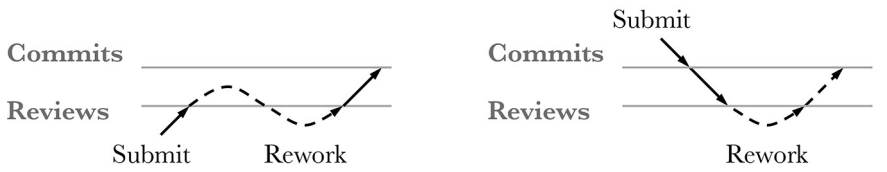
> "Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate."

9.4.2 Code Reviews Today

The differences between traditional code inspections and modern code reviews touch every aspect of a review. For concreteness the comparison in Fig. 9.6 is with open-source code reviews. Similar comments apply to companies like Google. Note that there may be exceptions to the general observations in Fig. 9.6.[16]

Invested Expert Reviewers

Open-source projects have hundreds of contributors, who are geographically dispersed.[17] A trusted group of core developers is responsible for the integrity of the code base.



(a) REVIEW THEN COMMIT (b) COMMIT THEN REVIEW

Figure 9.7: The Review-then-Commit and the Commit-then-Review processes.

Contributions are broadcast to a developer mailing list for review. Reviewers self select, based on their expertise and interests. They respond with comments and suggested fixes. While the number of reviewers for each contribution is small, 1-3, a larger community of developers is aware of the review.[18]

At Google, the main code repository is organized into subtrees with owners for each subtree.

> "All changes to the main source code repository MUST be reviewed by at least one other engineer."[19]

The author of a change chooses reviewers; however, anyone on the relevant mailing is free to respond to any change. All changes to a subtree must be approved by the owner of the subtree.

In general, self-selected reviewers have considerable expertise.

Review Early and Often

It is a good practice to review all code before it is committed; that is, before it goes live in a production setting. Major companies, like Google, require a review before a commit. Open-source software projects require code from new contributors to be reviewed before it is committed.

Projects that review all code have frequent reviews of small pieces of code: tens of lines, say 30-50, instead of the hundreds of line in a traditional inspection. Code for review is self-contained, so reviewers see not only the change, but the context for the change.

The *Review-then-Commit* process is illustrated in Fig. 9.7(a). A contributor submits code for review. The reviewers examine the code for defects and suggest fixes. The contributor reworks the code and resubmits. Once the reviewers have no further comments, the code is committed and become part of the production code base. The dashed line indicates that rework is conditional on reviewers have comments that need to be addressed.

An alternative review process is followed if a change is urgent or if the contributor is known and trusted. The *Commit-then-Review* process is illustrated in Fig. 9.7(b). A contributor commits the code and notifies potential reviewers,

who examine the change. Based on their comments, the commit either holds or the the change is rolled back and reworked. The dashed line represents the case in which the change is reworked until it is approved by the reviewers.

Asynchronous Rapid Responses

With self-selected geographically-distributed reviewers, code reviews are *asynchronous*: there is no group meeting. Reviewer comments are collected by email or though an online tool. Reviewers respond within hours.

9.5 Static Analysis

Static analysis examines a program without running it. The purpose of static analysis is to find anomalies, be they potential defects or deviations from guidelines about programming practices. Static analysis is effective enough that it has become an essential verification technique, along with reviews and testing.

What static analysis cannot do is to prove significant properties of a program, such as correctness, or predict the value of a variable at run time. The problem of proving such properties is equivalent to solving the famous "halting problem," which is known to be undecidable.

For example, we cannot predict whether the value of `x` will be `0` or `1` at the end of the following program fragment:

```
x = 0;
if( f() ) x = 1;
```

There can be no general algorithm to decide whether an arbitrary computation `f()` will halt, so we cannot predict whether control will get to the assignment `x = 1`.

What static analysis can do is to handle special cases. It can identify specific kinds of questionable program constructions. For example, by examining execution paths through a program, a static analyzer might identify lines of code that cannot be reached at run time. As another example, for some loops, a static analyzer might determine whether the loop is infinite—by examining the boolean expressions in the loop, the static analyzer might deduce that once control enters the loop, it can never leave.

In practice, static analyzers can exhaustively identify questionable constructions in millions of lines of code. A complete scan of the source code allows static analyzers to find defects that reviews and testing might miss. With the heightened interest in security, there is renewed interest in exhaustive static checking.

In short, some static analyzers can detect enough of the defects all of the time.

A drawback of static analysis, is that it can raise false alarms, along with flagging critical defects. In the past, the volume of false alarms was a barrier to adoption. Now, static analyzers use heuristics and deeper analysis to hold down

the number of false alarms. They also prioritize their warnings: the higher the priority, the more critical the defect.

9.5.1 A Variety of Static Checkers

Automated static analysis tools consist of a set of *checkers* or *detectors*, where each checker looks for specific questionable constructions in the source code. Questionable constructions include the following:

- A variable is used before it is defined.
- A piece of code is unreachable (such code is also known as *dead code*).
- A resource *leaks*; that is, the resource is allocated but not released.
- A loop never terminates; e.g., its control variable is never updated.

This list of constructions is far from complete. New checkers continue to be defined, inspired by real problems in real code. The examples in this section are drawn from real problems in production code.

Static analyzers rely on compiler techniques to trace the flow of control and data through a program. *Control-flow analysis* traces execution paths through a program. *Data-flow analysis* traces the connections between the points in a program where the value of a variable is defined and where that value could potentially be used. The two are related, since data-flow analysis can help with control-flow analysis and vice versa.

Checking for Infinite Loops

The general problem of detecting infinite loops is undecidable, as noted earlier in this section. Many loops have a simple structure, however, where the value of a variable, called a *control variable*, determines when control exits the loop. Simple deductions may suffice for deciding whether such a loop is infinite.

Example 9.1. The following code fragment is adapted from a commercial product:

```
for ( j = 0; j < length; j-- ) {
    ...  // j is not touched in the body of the loop
}
```

If the value of `length` is positive, `j<length` will remain true as `j` takes on successively larger negative values.

An infinite loop is probably not what the programmer intended. □

Checking for Unreachable Code

Control-flow analysis is needed for detecting unreachable code. Such code is typically a sign of a bug in the program.

```
1)  if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
2)          goto fail;
3)  if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
4)          goto fail;
5)  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
6)          goto fail;
7)  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
8)          goto fail;
9)          goto fail;
10) if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
11)         goto fail;
```

Figure 9.8: A bug in the form of an extra "`goto fail;`" introduced a vulnerability into an implementation of SSL.

Example 9.2. For 17 months, between September 2012 and February 2014, a security vulnerability lay undetected in code running on hundreds of millions of devices. The code had been open sourced, available for all to see.

The vulnerability was in the Secure Sockets Layer (SSL) code for the operating systems for iPhones and iPads (iOS), and Macs (OS X).[20] The vulnerability was significant, since it left the door open for an attacker to intercept communication with websites for applications such as secure browsing and credit-card transactions.

The vulnerability was due to a bug, in the form of an extra unwanted `goto`; see Fig. 9.8. The indentation of the second circled `goto` on line 9 is deceptive. Lines 7-11 have the following form:

```
7)  if( condition1 )
8)      goto fail;
9)  goto fail;          // bug: unwanted goto
10) if( condition2 )
11)     goto fail;
```

Note that the unwanted goto on line 9 prevents control from reaching the conditional on line 10.

A static analyzer would have detected the unreachable code on lines 10-11. □

Checking for Null-Dereference Failures

Null is a special value, reserved for denoting the absence of an object. A *null-dereference failure* occurs at run time when there is an attempt to use a null

```
1)  Logger logger = null;
2)  if (container != null)
3)      logger = container.getLogger();
4)  if (logger != null)
5)      logger.log(... + container.getName() + ...);
6)  else
7)      System.out.println(... + container.getName() + ...);
```

Figure 9.9: A program fragment from the open-source Apache Tomcat Server with a null-dereference bug.

value. Static analysis can detect potential null references.

The following Java program fragment assigns the value `null` to variable `logger` of class `Logger` and then promptly proceeds to use null value:

```
Logger logger = null;
...         // code that does not change the value of logger
logger.log(message);
```

Using data-flow analysis, we can deduce that the value of `logger` will be `null` when control reaches the last line of the above program fragment. At that point, a failure will occur: there will be no object for `logger` to point to, so the method call `log(message)` will fail.

Before reading the next example, can you find the potential null dereferences in Fig. 9.9?

Example 9.3. The real code fragment in Fig. 9.9, avoids a null dereference for `logger`, but it introduces a null dereference for `container`.

Data-flow analysis would discover that `logger` in Fig. 9.9 is defined in two places and used in two places. The two definitions are on lines 1 and 3. The two uses are on lines 4 and 5. As for `container`, there are no definitions; there are four uses, on lines 2, 3, 5, and 7.

A null-dereference failure will occur if `container` is `null` when line 1 is reached. Control then flows from the decision on line 2 to line 4, leaving `logger` unchanged at `null`. From the decision on line 4, control therefore flows to line 7, which has a use of `container`. But `container` is `null`, so we have a null-dereference failure.

There is no null dereference if `container` is non-null when line 1 is reached, even if `container.getLogger()` returns `null`. □

The open-source static analyzer FindBugs would warn about two potential null dereferences for the uses of `container` on lines 5 and 7.[21] From Example 9.3 only the null dereference on line 7 is possible. The FindBugs warning about the use of `container` on line 5 is therefore a false alarm. Such false alarms are called false positives.

9.5.2 False Positives and False Negatives

A warning from a static analyzer about a piece of code is called a *false positive* if the code does not in fact have a defect. False positives arise when a static analyzer errs on the side of safety and flags a piece of code that *might* harbor a defect. If the piece of code does not in fact have a defect, then the warning is a false positive.

A *false negative* is a defect that is not detected by static analysis.

Static analysis tools choose to hold down the number of false positives at the expense of introducing some false negatives. The designers of a commercial static analyzer, Coverity, observe

> "In our experience, more than 30% [false positives] easily cause problems. People ignore the tool. ... We aim for below 20% for 'stable' checkers. When forced to choose between more bugs or fewer false positives we typically choose the latter."[22]

From the above quote, the designers of Coverity choose fewer false positives over fewer false negatives. More false negatives means more bugs missed. Other static analysis tools make the same choice.

9.6 Conclusion

- A *static* property of a program can be analyzed without executing the program. By contrast, a *dynamic* property is a run-time property of the behavior of a program.
- *Software quality* is a general term for any of the following forms of quality:
  - *functional quality* is the degree to which a system meets user requirements for functionality;
  - *process quality* refers to the effectiveness of a process in organizing software development activities and teams;
  - *product quality* refers to inherent properties of a product that cannot be altered without altering the product itself;
  - *operational quality* refers to a customer's operational experience after the product is delivered;
  - *transcendental quality* refers to the indefinable goodness of a product;
  - the *value* notion of quality refers to a customer's willingness to pay for a product.
- A *fault* is a flaw in the source code, the design, the documentation, or some other software artifact. A *failure* occurs when the behavior of an implementation does not match the specification. Faults are a measure of product quality. Failures are a measure of operational quality.

- A *defect* is a fault or an omission from a software artifact. *Error* is a broad term for a fault, a failure, or an omission.
- The severity levels of defects are as follows (the lower the number, the more severe the defect):

  1. *critical* for total stoppage;
  2. *high* for major error that cripples some functionality;
  3. *medium* if there is a problem, but there is a workaround; and
  4. *low* for a cosmetic error.

- *Validation* refers to checking that a software artifact meets customer requirements; or, "Am I building the right product?" *Verification* refers to checking that an implementation is correct; or, "Am I building the product right?" In terms of overall effort, verification is a much bigger problem than validation.
- A *review* is a process or meeting for examining a software artifact for comment or approval. An *inspection* is a formal review by a carefully selected group of independent experts, with the purpose of finding "anomalies, including errors and deviations from standards and specifications."
- The primary purpose of an *architecture review* is to clarify the goals of the proposed architecture and confirm that a solution based on the architecture will meet those goals. The guiding principles for architecture reviews are:

  - The review is for the project team's benefit.
  - The expert reviewers are independent.
  - The project has a clear problem definition.
  - The architecture fits the problem.
  - The project has a system architect.

- In a *traditional* or *Fagan inspection*, a group of 3-6 independent experts prepare in advance for a group meeting to detect and collect defects in a software artifact. Subsequent studies have shown that two committed expert reviewers may be enough and that a group meeting is not necessary. Instead, reviewer comments can be collected asynchronously by email or through an online tool.
- The primary motivation for *code reviews* has changed from checking for defects to checking for whether the code is clean and whether it can be trusted. It is a good practice to review all code before it is committed; that is, before it goes live in a production setting. *Asynchronous code reviews* are conducted using email and online tools. Small, say 30-50 line, contributions are examined by either named reviewers or by self-selected reviewers, based on their expertise and interest.

  - With a *Review-then-Commit* process, a contribution must be approved before it is committed.
  - If a change is urgent or if a reviewer is trusted, the *Commit-then-Review* process allows a contribution to be committed first and then reworked, if needed.

- *Static analysis* examines a program for defects without running the program. Static analysis is essentially an automated review.
- Automated static analysis tools consist of a set of *checkers* or *detectors*, where each checker looks for specific questionable constructions in the source code. For example, there are checkers for undefined variables, unreachable code, null-dereferences, and infinite loops.
- A warning from a static analyzer about a piece of code is a *false positive* if the code does not in fact have a defect. A *false negative* is a defect that is not detected by static analysis. Developers ignore static analyzers that produce too many false positives, so static analyzers hold down false positives at the risk of missing some defects; that is, at the risk of having some false negatives.

Exercises for Chapter 9

Exercise 9.1. Explain the distinction between the following pairs of concepts:

a) failure and fault

b) process quality and product quality

c) validation and verification

d) a traditional inspection and an open-source code review

Exercise 9.2. For each of the following words,

a) anomaly	b) bug	c) defect	d) flaw	e) failure
f) fault	g) error	h) glitch	i) omission	j) problem

- Look up the word in a dictionary and write down its dictionary meaning.
- Classify the dictionary meaning as being closer to that of fault; closer to that of failure; or not a fit with either fault or failure. Explain your answer.

Exercise 9.3. For each of the following forms of software quality, associate two metrics to measure quality relative to that view: functional, process, product, and operational.

Exercise 9.4. For a deep-dive architecture review, come up with 10 separate security-related questions.

Exercise 9.5. For a deep-dive architecture review, come up with 10 separate performance-related questions.

Chapter 10

Testing

> "Individual programmers: Less than 50% efficient in finding bugs in their own software."
>
> — *Capers Jones, in a survey of the state of the art of software quality in 2013. He also observed that normal testing is often less than 75% efficient.*[1]

Software testing is the process of running a program in a controlled environment to check whether the program behaves as expected. The purpose of testing is to improve software quality. If a test fails—that is, the program does not behave as expected—there must be a fault, either in the program or in the specification of expected behavior. Either way, the test has provided feedback that can be used to remove the fault and improve quality.

At one time, coding and testing were distinct phases of software development. Testing accounted for half of the time and cost of development.[2] Coding was done by developers; testing was done by testers. Testers prided themselves on their ability to trigger failures and track down faults. Defects were more likely to be in the code for special or edge cases, since developers tended to pay more attention to the basic functionality.

Since then, test automation has blurred the distinction between coding and testing. Batteries of tests can be run automatically every time a change is made. Developers can thereby produce code of sufficient quality that the code can go straight from development to deployment. Automated tests are run as part of the deployment process to ensure that applications that used to work continue to work.

```
1)  year = ORIGINYEAR; /* = 1980 */
2)  while (days > 365)
3)  {
4)      if (IsLeapYear(year))
5)      {
6)          if (days > 366)
7)          {
8)              days -= 366;
9)              year += 1;
10)         }
11)     }
12)     else
13)     {
14)         days -= 365;
15)         year += 1;
16)     }
17) }
```

Figure 10.1: Where is the fault?

Although when and how tests are run may have changed, testing remains a significant part of software development. The principal techniques for defect detection and removal have remained the same. They are reviews, static analysis, and testing. Reviews and static analysis were covered in Chapter 9. This chapter explores the process of testing.

10.1 Overview of Testing

The code in Fig. 10.1 is from a digital music and video player. On December 31, 2008, owners of the player awoke to find that it froze on startup. On the last day of a leap year, the code in Fig. 10.1 loops forever.[3]

What went wrong?

Example 10.1. Suppose that the code in Fig. 10.1 is reached with variable `days` representing the current date as an integer. January 1, 1980 is represented by the integer 1; December 31, 1980 by 366 since 1980 was a leap year; and so on.

Variable `year` is initialized to 1980 on line 1. On exit from the while loop on lines 2-17, variable `year` represents the current year. The body of the loop computes the current year by repeatedly subtracting 366 for a leap year (line 8) or 365 for a non-leap year (line 14). Each subtraction is followed by a line that increments `year` (lines 9 and 15). In other words, the body of the loop counts down the days and counts up the years.

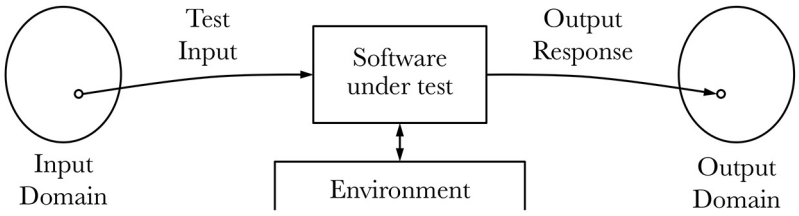


Figure 10.2: Software can be tested by applying an input stimulus and evaluating the output response.

On the 366th day of 2008, a leap year, line 6 is eventually reached with value 366 for `days`. Control therefore loops back from line 6 to line 2 with `days` unchanged. Ad infinitum. □

How is it that "simple" bugs escape detection until there is an embarrassing product failure? The rest of this section explores the process of testing and its strengths and limitations.

10.1.1 Issues During Testing

The issues that arise during testing relate to the four main elements in Fig. 10.2:

- *Software Under Test.* The software under test can be a code fragment, a component, a subsystem, a self-contained program, or a complete hardware-software system.
- *Input Domain.* A tester selects an element of some input domain and uses it as test input.
- *Output Domain.* The output domain is the set of possible output responses or observable behaviors by the software under test. Examples of behaviors include producing integer outputs, as in Example 10.1, and displaying a web page.
- *Environment.* Typically, the software under test is not be self contained, so an environment is needed to provide the context for running the software.

If the software under test is a program fragment, the environment handles dependencies on the rest of the program. The environment also includes the operating system, libraries, and external services that may be running either locally or in the cloud. In the early stages of development, external services can be simulated by dummy or mock modules with controllable behavior. For example, an external database can be simulated by a module that uses a local table seeded with known values.

Example 10.2. Suppose that the software under test is the code on lines 1-17 in Fig. 10.1. The input domain is the set of possible initial integer values for the variable `days`. The output domain is the set of possible final integer values for the variables `year` and `days`.

The code in Fig. 10.1 cannot be run as is, because it needs a definition for `ORIGNYEAR` and an implementation for function `IsLeapYear()`. These things must be provided by the environment. (We are treating the initial value of variable `days` as an input, so the environment does not need to provide a value for `days`.) □

The following questions capture the main issues that arise during testing:[4]

- How to stabilize the environment to make tests repeatable?
- How to select test inputs?
- How to evaluate the response to a test input?
- How to decide whether to continue testing?

10.1.2 Test Selection

The judicious selection of test inputs is a key problem during testing. Fortunately, reliable software can be developed without exhaustive testing on all possible inputs—exhaustive testing is infeasible.

The Input Domain

The term *test input* is interpreted broadly to include any form of input; e.g., a value such as an integer; a gesture; a combination of values and gestures; or an input sequence, such as a sequence of mouse clicks. In short, a test input can be any stimulus that produces a response from the software under test.

A set of tests is also known as a *test suite*.

The *input domain* is the set of all possible test inputs. For all practical purposes, the input domain is typically infinite. Variable `days` in Fig. 10.1 can be initialized to any integer value and, machine limitations aside, there is an infinite supply of integers.

Some faults are triggered by a mistimed sequence of input events. Therac-25 delivered a radiation overdose only when the technician entered patient-treatment data fast enough to trigger a fault; see Section 1.6. Other faults are triggered by an unfortunate combination of values. Avionics software is tested for interactions between multiple inputs; e.g., a decision may be based on data from a variety of sensors and a failure occurs only when the pressure crosses a threshold and the temperature is in a certain range.

It is important to test on both valid and invalid inputs, for the software must work as expected or valid inputs and do something sensible on invalid inputs. Crashing on invalid input is not sensible behavior.

Input domains can therefore consist a single values, (b) combinations of values, or (c) scenarios consisting of sequences of values.

Black-Box and White-Box Testing

During test selection, we can either treat the software under test as a black box or we can look inside the box at the source code. Testing that depends only on the software's interface is called *black-box* or *functional testing*. Testing that is based on knowledge of the source code is called *white-box* or *structural testing*. As we shall see in Section 10.2, white-box testing is used for smaller units of software and black-box testing is used for larger subsystems that are built up from the units.

Test design and selection is a theme that runs through this chapter.

10.1.3 Test Adequacy: Deciding When to Stop

Ideally, testing would continue until the desired level of software quality is reached. Unfortunately, there is no way of knowing when the desired level of quality is reached because, at any time during testing, there is no way of knowing how many more defects remain undetected. If a test fails—that is, the output does not match the expected response—the tester has discovered that there is a defect somewhere.

But, if the test passes, all the tester has learned is that the software works as expected on that particular test input. The software could potentially fail on some other input. As Edsger Dijkstra put it,

> "Testing shows the presence, not the absence of bugs."[5]

Stopping or Test-Adequacy Criteria

A *test adequacy* criterion is a measure of progress during testing. Adequacy criteria support statements of the form, "Testing is x% complete." Test adequacy criteria are typically based on three kinds of information: code coverage, input coverage, and defect-discovery data.

- *Code coverage* is the degree to which a construct in the source code is touched during testing. For example, statement coverage is the proportion of statements that are executed at least once during a set of tests. Code coverage is discussed in Section 10.3 on white-box testing.
- *Input coverage* is the degree to which a set of test inputs is representative of the whole input domain. For example, in Section 10.4 on black-box testing, the input domain will be partitioned into equivalence classes. Equivalence-class coverage is the proportion of equivalence classes that are represented in a test set.
- *Defect-discovery data* includes data about the number and severity of the defects discovered in a given time interval. When combined with historical data from similar projects, the rate of defect discovery is sometimes used to make predictions about product quality.

Test adequacy criteria based on coverage and defect-discovery data are much better than arbitrary criteria such as stopping when time runs out or when a certain number of defects have been found. They cannot, however, guarantee the absence of bugs.

While testing alone is not enough, it can be a key component of an overall quality-improvement based on reviews, static analysis, and testing.[6]

10.1.4 Test Oracles: Evaluating the Response to a Test

Implicit in the above discussion is the assumption that we can readily tell whether an output is "correct;" that is, we can readily decide whether the output response to an input stimulus matches the expected output. This assumption is called the oracle assumption.

The *oracle assumption* has two parts:

1. There is a specification that defines the correct response to a test input.
2. There is a mechanism to decide whether or not a response is correct. Such a mechanism is called an *oracle*.

Most of the time, there is an oracle, human or automated. For values such as integers and characters, all an oracle may need to do is to compare the output with the expected value. An oracles based on a known comparison can be easily automated.

Graphical and audio/video interfaces may require a human oracle. For example, how do you evaluate a text-to-speech system? It may require a human to decide whether the spoken output sounds natural to a native speaker.

Questioning the Oracle Assumption

The oracle assumption does not always hold. A test oracle may not be readily available, or may be nontrivial to construct.

Example 10.3. Elaine Weyuker gives the example of a major oil company's accounting software, which "had been running without apparent error for years."[7] One month, it reported $300 for the company's assets, an obviously incorrect output response. This is an example of knowing that a response is incorrect, without knowing the right response.

There was no test oracle for the accounting software. Even an expert could not tell whether "$1,134,906.43 is correct and $1,135,627.85 is incorrect." □

Example 10.4. Consider a program that takes as input an integer n and produces as output the nth prime number. On input 1 the program produces 2, the first prime number; on 2 it produces 3; on 3 it produces 5; and so on. On input 1000, it produces 7919 as output.

Is 7919 the 1000th prime? (Yes, it is.)

It is nontrivial to create an oracle to decide if a number p is the nth prime number.[8] □

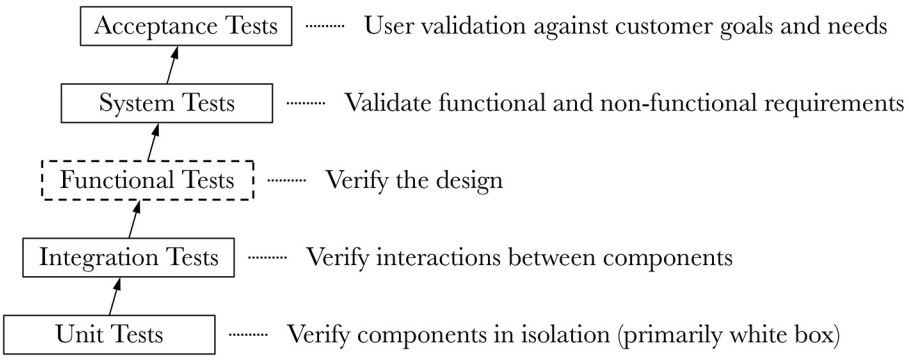


Figure 10.3: Levels of testing. Functional tests may be merged into system tests; hence the dashed box.

10.2 Levels of Testing

Testing becomes more manageable if the problem is partitioned: bugs are easier to find and fix if components are debugged before they are assembled into a larger system. A components-before-systems approach motivates the levels of testing in Fig. 10.3. Testing proceeds from bottom to top; the size of the software under test increases from bottom to top.

Each level of testing plays a different role. The terms "verify" and "validate" were introduced as follows in Section 9.1:

Validation: "Am I building the right product?"
Verification: "Am I building the product right?"

The top two levels in Fig. 10.3 validate that customer needs and requirements are met. The lower levels verify the implementation. System and functional testing may be combined into a single level that tests the behavior of the system; hence the dashed box for functional tests. The number of levels varies from project to project, depending on the complexity of the software and the importance of the application.[9]

Based on data from hundreds of companies, each level catches about one in three defects.[10]

10.2.1 Unit Testing

A *unit* of software is a logically separate component that can be tested by itself. It may be a module or part of a module. *Unit testing* verifies a unit in isolation from the rest of the system. With respect to the overview of testing in Fig. 10.2, the environment simulates just enough of the rest of the system to allow the unit to be run and tested.

```
public class Date {
   ...
   public int getYear(...) {
      ...
   }
   ...
}
```

(a) Software Under Test

```
public class DateTest {
   @Test
   public void test365() {
      Date date = new Date();
      int year = date.getYear(365);
      assertEquals(year, 1980);
    }
}
```

(a) A JUnit 4 Test

Figure 10.4: A JUnit test.

Unit testing is primarily white box testing, where test selection is informed by the source code of the unit. White-box testing is discussed in Section 10.3.

xUnit: Automated Unit Testing

Convenient automated unit testing profoundly changes software development. A full suite of tests can be run automatically at any time to verify the code. Changes can be made with reasonable assurance that the changes will not break any existing functionality. Code and tests can be developed together; new tests can be added as development proceeds. In fact, automated tests enable test-first or test-driven development, where tests are written first and then code is written to pass the tests.

Convenience was the number one goal for *JUnit*, a framework for automated testing of Java programs. Kent Beck and Erich Gamma wanted to make it so convenient that "we have some glimmer of hope that developers will actually write tests."[11]

JUnit quickly spread. It inspired unit testing frameworks for other languages, including CUnit for C, CPPUnit for C++, PyUnit for Python, JSUnit for JavaScript, and so on. This family of testing frameworks is called *xUnit*.

An xUnit test proceeds as follows:

set up the environment;
run test;
tear down the environment;

From Section 10.1, the environment includes the context that is needed to run the software under test. For a Java program, the context includes values of variables and simulations of any constructs that the software relies on.

Example 10.5. The pseudo-code in Fig. 10.4(a) shows a class `Date` with a method `getYear()`. The body of `getYear()` is not shown—think of it as implementing the year calculation in Fig. 10.1.

The code in Fig. 10.4(b) sets up a single JUnit test for `getYear()`. The annotation `@Test` marks the beginning of a test. The name of the test is `test365`.

A descriptive name is recommended, for readability of messages about failed tests. Simple tests are recommended to make it easier to identify faults.

The test creates object `date` and calls `getYear(365)`, where 365 represents December 31, 1980. JUnit supports a range of assert methods; `assertEquals()` is an example. If the computed value `year` does not equal 1980, the test fails, and JUnit will issue a descriptive message.

For more information about JUnit, visit `junit.org`. □

10.2.2 Integration Testing

Integration testing verifies interactions between the components of a subsystem. Integration testing is typically black-box testing.

Prior unit testing increases the likelihood that a failure during integration testing is due to interactions between components instead of being due to a fault within some component.

With *big bang* integration testing, the whole system is assembled all at once from individually unit tested components. *Incremental* integration testing is a better approach: interactions are verified as components are added, one or more at a time, to a previously tested set of components.

Dependencies Between Modules

Incremental integration testing must deal with dependencies between modules.

Example 10.6. In a model-view-controller architecture, the view displays information that it gets from the model. The view depends on the model, but not the other way around.

The model in Example 8.6 held information about a picture of the Mona Lisa, including a digital photo and the height, width, and resolution of the photo. There were two views: one displayed the digital photo; the other displayed the height, width, and resolution of the photo.

Both views got their information from the model, so they depended on the model. The model, however, was not dependent on the views. □

Dependencies between modules can be defined in terms of a uses relationship: module M *uses* module N, if N must be present and satisfy its specification for M to satisfy its specification.[12] Note that the used module need not be a subcomponent of the using module. In Example 10.6, the views used the model, but the model was not a subcomponent of either view.

During incremental integration, suppose module M uses module N. Then, either M must be added after N or there must be a "stub" that can be used instead of N for testing purposes. More precisely, module N' is a *stub* for N if N' has the same interface and enough of the functionality of N to allow testing of modules that use N.

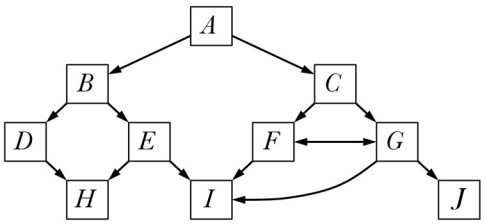


Figure 10.5: Modules to be integrated. The horizontal line between F and G means that they use each other.

Example 10.7. The edges and paths in Fig. 10.5 represent the uses relation between the modules in a system. Module A uses all the modules below it. A uses B and C directly; it uses the other modules indirectly.

A uses B, so B must be present and work for A to work. But, B uses D and E, so D and E must also be present and work for A to work. □

Top-Down Integration Testing

With stubs, integration testing can proceed top down. Testing of module A in Fig. 10.5 can begin with stubs for B and C. Then, testing of A and B together can begin with stubs for C, D, and E. Alternatively, testing A and C together can begin with stubs for B, F, and G.

A disadvantage with top-down integration testing is that stubs need to provide enough functionality for the using modules to be tested. Glenford J. Myers cautions that

> "Stub modules are often more complicated than they first appear to be."[13]

Bottom-Up Integration Testing

With bottom-up integration testing, a module is integrated before any using module needs it; that is, if M uses N, then M is integrated after N. If two modules use each other, then they are added together.

Bottom-up integration testing requires drivers: a *driver* module sets up the environment for the software under test. Automated testing tools like the xUnit family set up the environment, so there is no need for separate drivers.

Example 10.8. Consider the modules in Fig. 10.5. Any ordering that adds a child node before a parent node can serve for incremental integration testing, except for modules F and G, which use each other. They must be added together.

Here are two possible ordering for bottom-up testing:

H, D, I, E, B, J, F and G, C, A
J, I, F and G, C, H, E, D, B, A

□

10.2.3 Functional and System Testing

Functional testing verifies that the overall system meets its design specifications. *System testing* validates the system as a whole with respect to customer requirements, both functional and nonfunctional. The overall system may include hardware and third-party software components.

Functional testing may be merged into system testing. If the two are merged, then system testing performs a combination of verification and validation.

System testing is a major activity. Nonfunctional requirements include performance, security, usability, reliability, scalability, serviceability, documentation, among others. These are end-to-end properties that can be tested only after the overall system is available.

Testing for properties like security, usability, and performance are important enough that they may be split off from system testing and conducted by dedicated specialized teams.

10.2.4 Acceptance Testing

Acceptance testing differs from the other levels of testing since it is performed by the customer organization. Mission-critical systems are usually installed in a lab at a customer site and subjected to rigorous acceptance testing before they are put into production. Acceptance tests based on usage scenarios ensure that the system will support the customer organization's business goals.

10.2.5 Case Study: Test Early and Often

Testing of a component can begin as soon as the component is coded, while other components are still in an earlier stage of development.

The following example is motivated by the development process for a highly reliable communications product. The development team for the product was made up of small subteams that worked independently on separate components called features. The code for the features was then integrated to form the product. Testing was a priority. The development process called for extensive unit, functional, system, and performance testing.

Example 10.9. The process in Fig 10.6 focuses on the building phase of a project. The process does not show the initial activities to identify customer needs and opportunities and to define a product.

The process in Fig. 10.6 begins with system design. During system design, the components of the system are designed and plans are drawn to develop the

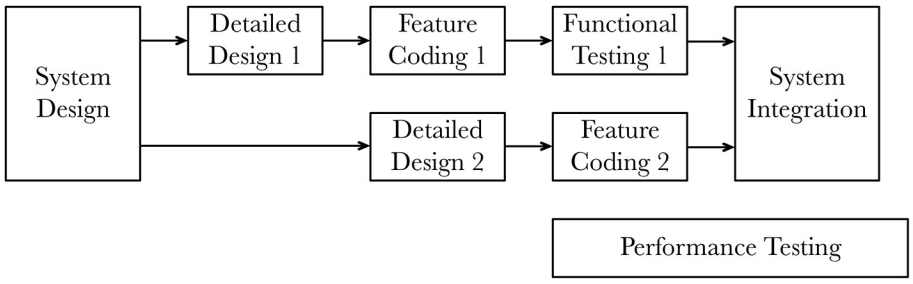


Figure 10.6: Simplified version of a plan-driven process with an emphasis on testing.

components independently. Between initial system design and final system integration are two parallel subprocesses for developing the components, referred to as Feature 1 and Feature 2.

The development of the two features is staggered in time: coding for Feature 1 overlaps with the detailed design of Feature 2; functional testing of Feature 1 overlaps with the coding of Feature 2.

The roles for the process in Fig. 10.6 are project manager, designer, coder, and tester. Parallel development of features allows the same designers and coders to work on both features. After the initial system design, the designers work first on the detailed design for Feature 1 and then on the detailed design for Feature 2. Similarly, the coders work sequentially on coding for Feature 1 and Feature 2. Functional testing of Feature 1 begins as soon as the code for the feature is available.

Once both features have been coded, the testers begin system integration and testing. Functional testing of Feature 2 is combined with system integration. Another group of testers begins performance testing once the first feature has been coded.

The project manager is responsible for assembling the team and coordinating the work of designers, coders, and testers. □

Instead of two features, the product that motivated Example 10.9 had multiple features. Overall planning and system design were done up front, before detailed design, coding, and testing.. Careful planning ensured that the independently developed components came together as designed during system integration. Not only did the project meet its functionality, reliability, and performance goals, it was on time and under budget.

10.3 Testing for Code Coverage

Since program testing cannot prove the absence of bugs it is unrealistic to expect that testing must continue until all defects have been found. The test-adequacy criteria in this section are based on code coverage, which is the degree to which a construct in the source code is executed during testing. In practice, the adequacy of test sets is measured relative to coverage targets.

Code coverage is closely, but not exclusively, associated with white-box testing, where test selection is based on knowledge of the source code.

10.3.1 Control-Flow Graphs

A *control-flow graph*, or simply *flow graph*, represents the flow of control between the statements in a program. Flow graphs have two kinds of nodes:

- *basic nodes* with one incoming and one outgoing edge; and
- *decision nodes* with one incoming and two or more outgoing edges.

Basic nodes represent assignments and procedure calls. Decision nodes result from Boolean expressions, such as those in conditional and while statements. The two outgoing edges from a decision node are called *branches* and labeled T for *true* and F for *false*. If the Boolean expression in the decision node evaluates to *true* control flows through the T branch; otherwise, control flows through the F branch.

For convenience, a sequences of one or more basic nodes may be merged to form a node called a *basic block*. As a further convenience, assume that a flow graph has a single *entry* node with no incoming edges and a single *exit* node with no outgoing edges.

Example 10.10. The flow graph in Fig. 10.7 represents the flow of control between the statements in Fig. 10.1. The flow graph has three basic blocks, B_1-B_3, and three decision nodes, D_1-D_3. Basic block B_1 has one assignment, which initializes `year`. (We take the liberty of initializing `year` to `1980`, instead of `ORIGINYEAR`.) B_2 has two assignments:

```
days -= 365;
year += 1;
```

Decision node D_1 corresponds to the decision in the while statement

```
while (days > 365) { ... }
```

D_2 corresponds to the decision in the conditional statement

```
if (IsLeapYear(year)) { ... } else {...}
```

□

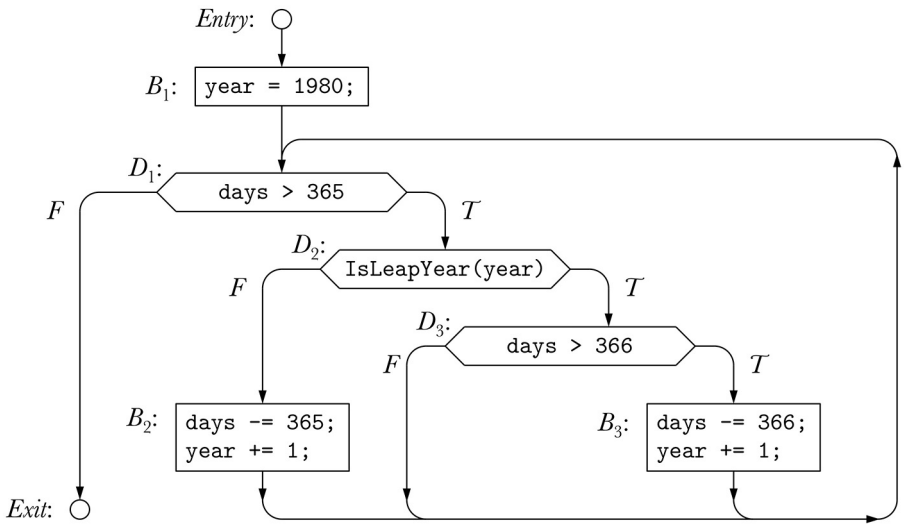


Figure 10.7: Control-flow graph for the code fragment in Fig. 10.1.

The flow graph for a program can be constructed by applying rules like the ones in Fig. 10.8. The rules are for a simple language that supports assignments, conditionals, while loops, and sequences of statements. Variables E and S represent expressions and statements, respectively. The rules can be applied recursively to construct the flow graph for a statement. It is left to the reader to extend the rules to handle conditionals with both then and else parts.

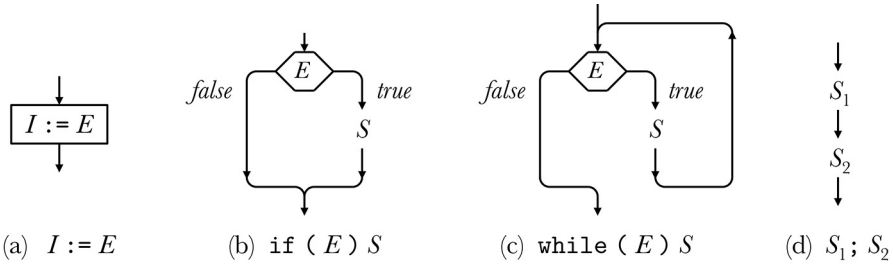


Figure 10.8: Rules for constructing a control-flow graph.

Paths Through a Flow Graph

A test corresponds to a path through a flow graph. Testing is done by running a program, so a test corresponds to an execution of the program. Each execution corresponds to a path through the flow graph for a program. Hence the statement that a test corresponds to a path through a flowgraph.

More precisely, a *path* through a flow graph is a sequence of contiguous edges from the entry node to the exit node. Two edges are *contiguous* if the end node of the first edge is the start node of the second edge. A path can be written as a sequence of nodes $n_1, n_2, ...$, where there is an edge from node n_i to node n_{i+1}, for all i.

A *simple path* is a path with no repeated edges. A simple path through a loop corresponds to a single execution of the loop. If there was a second execution, one or more of the edges in the loop would repeat.

Example 10.11. This example relates tests of the code in Fig. 10.1 with paths through the flow graph in Fig. 10.7. A test of the code consists of an execution, where the test input is an initial value for variable `days`.

Consider the test 365; that is, the initial value of `days` is 365. The Boolean expression `days > 365` evaluates to false, so control skips the body of the while-loop. This execution corresponds to the simple path

$$Entry, B_1, D_1, Exit$$

With test 367, control goes once through the body of the while loop, before exiting. Variable `year` is initialized to `1980`, a leap year, so this test traces the simple path

$$Entry, B_1, D_1, D_2, D_3, B_3, D_1, Exit$$

Although node D_1 appears twice, this path is simple because no edge is repeated. □

10.3.2 Control-Flow Coverage Criteria

The correspondence between tests and paths is one-to-one. Each test of a program corresponds to a path through the program's flow graph, and vice versa. Code-coverage criteria can therefore be expressed either in terms of program constructs like statements, decisions, and executions or in terms of nodes, branches, and paths, respectively.

Coverage tracking is a job best done by automated tools that build flow graphs and keep track of coverage information as tests are run. Given the close correspondence between flow graphs and the source code, the tools display coverage information by annotating the source code. For example, statement coverage is typically displayed by showing the number of times each line of code is executed by a test set. The tools also produce reports that summarize the coverage achieved by a test set.

All-Paths Coverage: Strong But Unrealistic

With *all-paths coverage*, each path is traced at least once.

The set of all paths through a flow graph corresponds to the set of all possible executions of the program. All paths coverage therefore corresponds to exhaustive testing. Exhaustive testing is the strongest possible test-adequacy criterion: if a program passes exhaustive testing, then we know that the program will work for all inputs.

Exhaustive testing is also infeasible. A flow graph with loops has an infinite number of paths: given any path that goes through a loop, we can construct another longer path by going one more time through the loop.

Since all-paths coverage is an unattainable ideal, many more or less restrictive coverage criteria have been proposed. The rest of the this section considers some widely used code coverage criteria.

Node or Statement Coverage

Node coverage, also known as *statement coverage*, requires a set of paths (tests) that touch each node at least once. Node coverage is the weakest of the commonly used coverage criteria.

Example 10.12. For the flow graph in Fig. 10.7, 100% node coverage can be achieved without triggering the known leap-year bug: on the last day of a leap year the code loops forever.

From Example 10.11, the paths traced by the test inputs 365 and 367 are as follows (for readability, the entry and exit nodes are omitted):

Test Input	Path
365	B_1, D_1
367	$B_1, D_1, D_2, D_3, B_3, D_1$

The test set $\{365, 367\}$ covers the nodes

$$B_1, B_3, D_1, D_2, D_3$$

Note that node B_2 is not yet covered. For B_2 to be covered, control must take the F branch of decision node D_2 in Fig. 10.7. The first time through D_2, control will take the T branch, since the initial value of `year` is 1980, a leap year.

The path for test 732 takes the F branch to B_2 the second time the path reaches D_2, The path is

$$B_1, D_1, D_2, D_3, B_3, D_1, D_2, B_2, D_1$$

The singleton test set $\{732\}$ happens to covers all nodes in Fig. 10.7. In general, multiple tests are needed to achieve the desired level of node coverage. □

While 100% node coverage is a desirable goal, it may not be achievable. If the software under test has dead code, by definition, the dead code cannot be reached during any execution. No test can therefore reach it. While dead code can be removed by refactoring, legacy systems are touched only as needed.

In practice, companies set stricter node coverage thresholds for new than for legacy code.

Branch Coverage is Stronger than Node Coverage

Branch coverage, also known as *decision coverage*, requires a set of paths (tests) that touch both the true and false branches out of each decision node. Branch coverage is a stronger criterion than node coverage. (As we shall see, branch coverage does uncover the leap-year bug in Fig. 10.7.)

Specifically, branch coverage *subsumes* node coverage, which means that any test set that achieves 100% branch coverage also achieves 100% node coverage. The converse is false, as the next example demonstrates.

Example 10.13. From Example 10.12, the tests 365, 367, and 732 correspond to the following paths

Test Input	Path
365	B_1, D_1
367	$B_1, D_1, D_2, D_3, B_3, D_1$
732	$B_1, D_1, D_2, D_3, B_3, D_1, D_2, B_2, D_1$

The branch coverage of these tests is as follows:

Test Input	D_1	D_2	D_3
365	F		
367	T, F	T	T
732	T, F	T, F	T

These tests achieve 100% node coverage, but they do not achieve 100% branch coverage because they do not cover the F branch from D_3 to D_1. (In fact, test 732 alone covers all branches covered by the other tests.)

This F branch out of D_3 is covered only when the code is in an infinite loop. Here's why. For the branch (D_3, D_1) to be taken, the following must hold:

`days > 365`	*true*	at node D_1
`IsLeapYear(year)`	*true*	at node D_2
`days > 366`	*false*	at node D_3

Together, these observations imply that, when the branch (D_3, D_1) is taken, `days` must have value 366 and `year` must represent a leap year. With these values for `days` and `year`, the program loops forever.

Thus, a test suite designed to achieve 100% branch coverage would uncover the leap-year bug.

The smallest test input that triggers the infinite loop is the value 366 for `days`—the corresponding date is December 31, 1980. □

Other Control-Flow Coverage Criteria

In practice, node and branch coverage are the most popular control-flow-based adequacy criteria. Branch coverage subsumes node coverage: any test set that achieves 100% branch coverage also achieves 100% node coverage. A number of other control-flow-based criteria have ben proposed; e.g.,[14]

- *Loop count coverage*: exercise each loop up to k times, where k is a parameter.
- *Length-n path coverage*: cover all sub-paths of length n.

10.3.3 MC/DC: Modified Condition/Decision Coverage

Branch coverage is adequate for the vast majority of decisions. Branch coverage is not enough, however, for decisions involving complex Boolean expressions containing operators such as `&` (logical and) and `|` (logical or) For example, suppose the value of the following decision is *false*:

```
(pressure > 32) & (temperature <= LIMIT)
```

Is it *false* because `pressure` is less than or equal to `32` or is it *false* because `temperature` is greater than `LIMIT`?

In discussing Boolean expressions, it is helpful to distinguish between atomic expressions which do not contain Boolean operators and general expressions which could. A *condition* is an atomic Boolean expression; e.g., `days > 365` or `IsLeapYear(year)`. A *decision* is a Boolean expression formed by applying Boolean operators to one or more conditions.

Branch coverage is adequate for decisions involving a single condition, as in

```
while (days > 365) { ... }
```

One study found that almost 85% of the decisions in a collection of software tools were based on just one "condition."[15]

For system safety and security, however, it is not enough for a coverage criterion to be adequate 85% of the time. All a hacker needs is one security vulnerability. One flaw may be enough to jeopardize a mission-critical system. Avionics software can have complex Boolean expressions with multiple conditions. The same study found that 17% of the decisions in a flight-instrumentation system had two or more conditions; over 5% had three or more conditions.[15]

For avionics software, the US Federal Aviation Administration recognizes a criterion called MC/DC, which is stronger than branch coverage.[16] MC/DC has also been recommended for detecting security backdoors.[17]

Condition and Decision Coverage are Independent

Let T and F denote the Boolean truth values, *true* and *false*, respectively. Let the lowercase letters $a, b, c, ...$ denote conditions; e.g.,

Condition a: `pressure > 32`
Condition b: `temperature <= LIMIT`

The expression a `&` b is a decision, representing

```
(pressure > 32) & (temperature <= LIMIT)
```

Condition coverage requires that each condition in a decision take on both truth values, T and F. *Decision coverage* requires each decision to take on both truth values. As we shall see, condition coverage does not ensure decision coverage, and vice versa.

While discussing condition and decision coverage, it is convenient to summarize tests by writing tables like the following for decision $a \,|\, b$:

TEST	a	b	$a \,\vert\, b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Each row of this table represents a test. The columns represent the values of the conditions a and b and the decision $a \,|\, b$. In test 2, a is T and b is F, so $a \,|\, b$ is T.

Example 10.14. Tests 2 and 3 above provide condition coverage, but not decision coverage. Condition coverage follows from the observation that in tests 2 and 3, a is T and F, respectively; b is F and T, respectively. But, the two tests do not provide decision coverage, since $a \,|\, b$ is T in both tests.

Tests 2 and 4 provide decision coverage, but not condition coverage. While the value of $a \,|\, b$ flips from T to F in these tests, b is not covered, since b is F in tests 2 and 4. □

MC/DC Pairs of Tests

MC/DC is a stronger criterion than condition and decision coverage put together. The following two tests provide both condition and decision coverage:

TEST	a	b	$a \,\vert\, b$
1	T	T	T
4	F	F	F

The limitation of these tests is that both a and b vary together: they both flip from T to F together.

Modified Condition/Decision Coverage (MC/DC) requires each condition to independently affect the outcome of the decision. Independently means that for each condition x, there is a pair of tests such that

- the outcome of the decision flips from T to F, or vice versa,

TEST	a	b	$a \mid b$	a	b
1	T	T	T		
2	T	F	T	4	
3	F	T	T		4
4	F	F	F	2	3

TEST	a	b	$a \,\&\, b$	a	b
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

Figure 10.9: Pairs tables for decisions $a \mid b$ and $a \,\&\, b$.

- the value of condition x also flips, and
- the values of the remaining conditions stay the same.

Such a pair of tests is called an *MC/DC pair* of tests for x.

Example 10.15. For decision $a \mid b$, the following tests form a pair for a:

TEST	a	b	$a \mid b$
2	T	F	T
4	F	F	F

The following tests form a pair for b:

TEST	a	b	$a \mid b$
3	F	T	T
4	F	F	F

□

A *pairs table* for a decision succinctly identifies the MC/DC pairs for the conditions in the decision. The pairs tables for $a \mid b$ and $a \,\&\, b$ appear in Fig. 10.9. A pairs table is an extension of a truth table for the decision. A truth table has a column for each condition and a column for the decision. The rows of the truth table show all combinations of values for the conditions and the corresponding value for the decision. A pairs table adds a column for each condition x. If (i, j) is a pair for x, then the added column for x has j in row i and i in row j.

Example 10.16. The pairs table for the decision $(a \,\&\, b) \mid c$ appears in Fig. 10.10.

There is only one MC/DC pair for a: it is $(2, 6)$. Since $(2, 6)$ is a pair, there is a 6 in row 2 and a 2 in row 6. In the other pairs for a the outcome of the decision does not flip. The decision remains T for the pairs $(1, 5)$ and $(3, 7)$; it remains F for the pair $(4, 8)$.

The only pair for b is $(2, 4)$. There are three pairs for c: they are $(3, 4)$, $(5, 6)$ and $(7, 8)$. □

TEST	a	b	c	$(a \,\&\, b) \mid c$	a	b	c
1	T	T	T	T			
2	T	T	F	T	6	4	
3	T	F	T	T			4
4	T	F	F	F		2	3
5	F	T	T	T			6
6	F	T	F	F	2		5
7	F	F	T	T			8
8	F	F	F	F			7

Figure 10.10: Pairs tables for the decision $(a \,\&\, b) \mid c$.

MC/DC Tests

The MC/DC tests for a decision can be deduced from its pairs table. The fewer the tests the better.

A direct approach is

for each condition x, pick a pair for x, any pair.

The resulting number of tests is at most twice the number of conditions. The actual number may be smaller, since some tests may appear in more than one pair. With the direct approach, the number of tests grows linearly with conditions, since each condition adds at most two tests. The number of possible tests grows exponentially, however, since the number of possible tests doubles with each condition.

Fewer tests result from the following approach

for each condition with only one pair, pick that pair;
for the remaining conditions x,
  pick the pair that adds the fewest tests;

Example 10.17. For the decision $(a \,\&\, b) \mid c$ in Fig. 10.10, $(2, 6)$ is the only pair for a and $(2, 4)$ is the only pair for b, so they must be picked. Of the three pairs for c, either $(3, 4)$ or $(5, 6)$ would be a better choice than $(7, 8)$, since they overlap with the tests needed for a and b. Choosing the pair $(3, 4)$ for c, we get four tests:

TEST	a	b	c	$(a \,\&\, b) \mid c$
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
6	F	T	F	F

□

10.3.4 Data-Flow Coverage

10.4 Testing for Input-Domain Coverage

From the overview of testing in Section 10.1, testing proceeds roughly as follows:

- Select a test input for the software under test.
- Evaluate the software's response to the input.
- Decide whether to continue testing.

Test inputs are drawn from a set called the input domain. The input domain can be infinite. If not infinite, if is typically so large that exhaustive testing of all possible inputs is infeasible. Test selection is therefore based on some criterion for sampling the input domain.

The selection criteria in this section are closely, but not exclusively, associated with black-box testing, which treats the software under test as a black box that hides the source code. Test selection is based on the software's specification.

10.4.1 Equivalence-Class Coverage

Equivalence partitioning is a heuristic technique for partitioning the input domain into subdomains with inputs that are equivalent for testing purposes. The subdomains are called equivalence classes. If two test inputs are in the same equivalence class, we expect them to provide the same information about a program's behavior: they either both catch a fault or they both miss the fault.

A test suite provides *equivalence-class coverage* if the set includes a test from each equivalence class.

There are no hard and fast rules for defining equivalence classes—just guidelines. The following example sets the stage for considering some guidelines.

Example 10.18. Consider a program that determines whether a year between 1800 and 2100 represents a leap year. Strictly speaking, the input domain of this program is the range 1800-2100, but let us take the input domain to be the integers, since the program might be called with any integer. Integers between 1800 and 2100 will be referred to as valid inputs; all other integers will be referred to as invalid inputs.

As a first approximation, we might partition the input domain into two equivalence classes, corresponding to valid and invalid integer inputs. For testing, however, it is convenient to start with three equivalence classes: integers up to 1799; 1800 through 2100; and 2101 and higher.

These equivalence classes can be refined, however, since some years are leap years and some years are not. The specification of leap years is as follows:

> "Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400."[18]

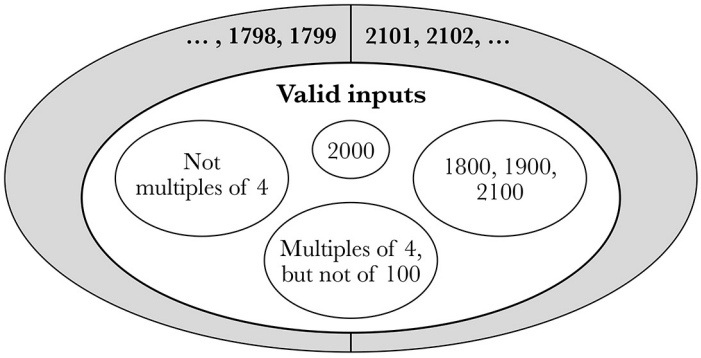


Figure 10.11: Equivalence classes for a leap-year program. The two shaded regions are for invalid test inputs.

This specification distinguishes between years divisible by 4, 100, 400, and all other years. These distinctions motivate the following equivalence classes (see Fig. 10.11):

- Integers less than or equal to 1799,
- Integers between 1800 and 2100 that are not divisible by 4.
- integers between 1800 and 2100 that are divisible by 4, but not by 100.
- The integers 1800, 1900, and 2100, which are divisible by 100, but not by 400.
- The integer 2000, which is divisible by 400.
- Integers that are greater than or equal to 2101.

The leap-year program can now be tested by selecting representatives from each equivalence class. □

Equivalence classes are designed. Two people working from the same specification could potentially come up with different designs. The following are some guidelines for getting started with equivalence partitioning:[19]

- If the inputs to a program are from a range of integers m-n, then start with three equivalence classes. The first class contains invalid inputs that are less then or equal to $m-1$; the second contains the valid inputs $m, m-1, ..., n$; the third contains the invalid inputs greater than or equal to $m+1$. This guideline can be generalized from integers to other data types.
- If an equivalence class has two or more inputs that produce different outputs, then split the equivalence class into subclasses, where all of the inputs in a subclass produce the same output.

- If the specification singles out one or more test inputs for similar treatment, then put all inputs that get the same "same" treatment into an equivalence class.
- For each class of valid inputs, define corresponding classes of invalid inputs.

Once equivalence classes are designed, tests can selected to cover them; that is, select a test input from each equivalence class.

10.4.2 Boundary-Value Coverage

Suppose that the input domain is ordered; that is, for two inputs i and j, it makes sense to talk of i being less than or before j, written $i < j$. Then, a value is a *boundary value* if it is either the first or the last—alternatively, the least or the greatest—with respect to the ordering.

Based on experience, errors are often found at boundary values. For example, the date-calculation code in Fig, 10.1 fails on December 31st of a leap year, a boundary value.

Boundary-value testing leverages the equivalence classes defined during equivalence partitioning.

Example 10.19. In Fig. 10.11, consider the equivalence classes for valid and invalid inputs. The valid inputs are the years between 1800 and 2100. The boundaries of this equivalence class are 1800 and 2100. There are two classes for invalid inputs: the smaller class of years less than 1800 and the bigger class of years greater than 2100. The upper boundary for the smaller class is 1799, but there is no lower boundary, since there are an infinite number of integers smaller than 1800. Similarly, the lower boundary for the bigger class is 2101 and there is no upper boundary.

For the equivalence class of years between 1800 and 2100 that are not multiples of 4, lower boundary is 1801 and the upper boundary is 2099. □

A test suite provides *boundary value coverage* if it includes the upper and lower boundaries of each of the equivalence classes. For an equivalence class with one element, the lower and upper boundaries are the same. In Fig. 10.11, the year 2000 is in a class by itself.

10.4.3 Combinatorial Testing

Combinatorial testing is an efficient technique for detecting failures that result from a combination of factors—a *factor* is a quantity that can be controlled during testing. At the unit level, failures can result from complex decisions involving multiple conditions. Two factors, pressure and temperature, are involved in the following pseudo-code:

```
if (pressure > 32 & temperature > LIMIT) {
    // faulty code
} else {
    // good code
}
```

The faulty code is reached at specific combinations of pressure and temperature.

At the system level, failures can result from combinations of components, as in the following example.

Example 10.20. Consider the problem of testing a software product that must support multiple browsers (Chrome, Safari, Explorer, Firefox), run on multiple platforms (Linux, Windows, Mac OS), and interface with multiple databases (MongoDB, Oracle, MySQL).

With 4 browsers, 3 platforms, and 3 databases, the number of combinations is $4 \times 3 \times 3 = 36$. That's 36 test sets, not tests, since the full test set must be run for each combination. The problem gets worse if the product must support not only the current version of a browser, but previous versions as well. □

A software failure that results from a specific combination of factors is called an *interaction failure*. A 2-way interaction involves two factors; a 3-way interaction involves three factors; and so on. An empirical study found that roughly 60% of web-server failures involved two or more factors. Incidentally, web-server and browser failures involved more complex interactions than either medical devices or a NASA application.[20].

Combinatorial testing is based on the empirical observation that

> "most failures are triggered by one or two factors, and progressively fewer by three, four, or more factors, and the maximum interaction degree is small."[21]

Pairwise Interactions

Pairwise testing addresses 2-way interactions. The idea is to test all combinations of values for each pair of factors. For the system in Example 10.20 the pairs of factors are

$$(\text{browser}, \text{platform}),\ (\text{browser}, \text{database}),\ (\text{platform}, \text{database})$$

Some conventions will be helpful in organizing sets of tests. Let the letters $A, B, C, ...$ represent factors; e.g, A represents browser, B represents platform, and C represents database. Let the integers $0, 1, 2, ...$ represent factor values; e.g., for factor B (platform), 0 represents Linux, 1 represents Windows, and 2 represents Mac OS. Let two-letter combinations $AB, AC, BC, ...$ represent the pairs of factors (A, B), (A, C), $(B, C), ...$, respectively.

Consider tests involving two factors A and B, each of which can take on the two values 0 and 1. With two factors and two values, there are four possible combinations of values for the two factors. A test consists of a specific

Test	A	B	C
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

(a) All combinations

Test	A	B	C
2	0	0	1
3	0	1	0
5	1	0	0
8	1	1	1

(b) A 2-way covering array

Figure 10.12: Tables for three factors, each of which can have two possible values. See Example 10.21.

combination of values. The following table represents an exhaustive set of tests involving the two factors:

Test	A	B
1	0	0
2	0	1
3	1	0
4	1	1

In such tables, columns represent factors, rows represent tests, and table entries represent values for the factors.

A set of tests is a *2-way covering array* if the tests include all possible combinations for each pair of factors. The next two examples illustrate covering arrays.

Example 10.21. Consider a simplified version of Example 10.20, where each factor can have one of two values. Factor A (browser) can take on the two values 0 and 1 (Chrome and Safari); factor B (platform) the values 0 and 1 (Linux and Windows); and factor C (database) the values 0 and 1 (MongoDB and Oracle).

The table in Fig. 10.12(a) shows all possible combinations for three factors, where each factor can have one of two values. Tests 1-8 therefore constitute an exhaustive test set for three factors.

The four tests in Fig. 10.12(b) constitute a covering array for pairwise testing of three factors. The three pairs of factors are AB, AC, and BC.

Let us verify that the four tests cover all combinations of values for each of these pairs. Consider the pair AC. All possible combinations of values for AC are 00, 01, 10, and 11. These combinations are covered by tests 3, 2, 5, and 8, respectively. Test 3 has the form $0x0$, where both A and C have value 0. The

A	B
0	0
0	1
0	2
1	0
1	1
1	2

A	C
0	0
0	1
1	0
1	1

B	C
0	0
0	1
1	0
1	1
2	0
2	1

Test	A	B	C
1	0	0	1
2	0	1	0
3	0	2	1
4	1	0	0
5	1	1	1
6	1	2	0

Figure 10.13: All combinations for pairs AB, AC, and BC, and a covering array. See Example 10.22.

value of B is ignored for now, since we are focusing on the pair AC. Tests 2, 5, and 8 have the form $0x1$, $1x0$, and $1x1$, respectively.

The combinations of values for the pair AB, are covered by the tests 2, 3, 5, and 8. For the pair BC, consider the tests 5, 2, 3, and 8. □

Example 10.22. Now, suppose that factor B can take on three values 0, 1, and 2 (for Linux, Windows, and Mac OS), and that factors A and C can have two values, as in Example 10.21. The total number of combinations for the three factors are $2 \times 3 \times 2 = 12$.

For pairwise testing, 6 tests are enough. All combinations for the pairs AB, AC, and BC appear in Fig. 10.13, along with a covering array with 6 tests. For pair AB, tests 1-6, in that order, correspond to the rows in the combinations-table for AB. For pair AC, see tests 2-5, in that order. For pair BC, tests 4, 1, 2, 5, 6, and 3, correspond to the rows in the combinations-table for BC. □

Multiway Covering Arrays

The discussion of 2-way interactions generalizes directly to the interaction of three or more factors. More factors need to be considered since 2-way testing finds between 50% and 90% of faults, depending on the application. For critical applications, 90% is not good enough. Three-way testing raises the lower bound from 50% to over 85%.

The benefits of combinatorial testing become more dramatic as the size of the testing problem increases. The number of possible combinations grows exponentially with the number of factors. By contrast, for fixed t, the size of a t-way covering array grows logarithmically with the number of factors.[22] For example, there are $2^{10} = 1024$ combinations of 10 factors, with each factor having two values. There is a 3-way covering array, however, that has only 13 tests; see Fig. 10.14.[23]

Algorithms and tools are available for finding covering arrays. The general problem of finding covering arrays is believed to be a hard problem. A naive heuristic approach is to build up a covering array by adding columns for the

Test	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

Figure 10.14: 3-way covering array for 10 factors.

factors, one at a time. Entries in the new column can be filled in by extending an existing test, or by adding a row for a new test.

```
start with an empty array;
for each factor F:
    add a column for F;
    mark F;
    for each 3-way interaction XYF, where X and Y are marked:
        for each combination in the combinations table for XYF:
            if possible:
                fill in a blank in an existing row to cover the combination.
            else:
                add a row with entries in the columns for X, Y, and F;
                comment: leave all other entries in the row blank
```

10.5 Conclusion

Exercises for Chapter 10

Exercise 10.1. What is the distinction between the following:

a) fault and failure?

b) validation and verification?

c) domain and code coverage?

d) branch and decision coverage?

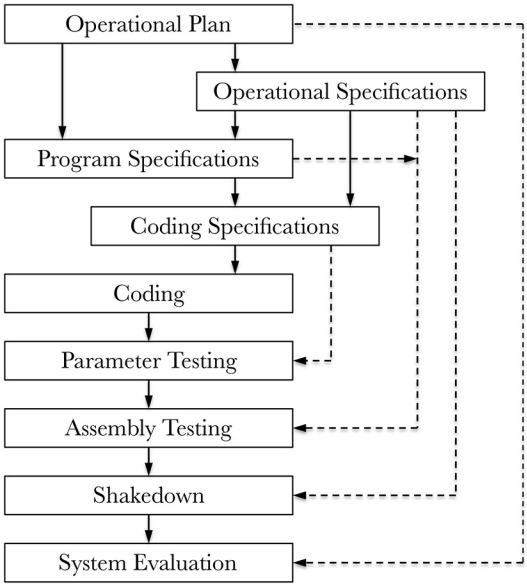


Figure 10.15: The development process for the SAGE air-defense system.

e) integration and sysem testing?

Be specific. Give examples where possible.

Exercise 10.2. What is the best fit between the testing phases of the development process in Fig. 10.15 and unit, functional, integration, system, and acceptance testing? Explain your answer.

Here are brief descriptions of the testing phases:

- *Parameter Testing.* Test each component by itself guided by the coding specification.
- *Assembly Testing.* As parameter testing completes, the system is gradually assembled and tested using first simulated inputs and then live data.
- *Shakedown.* The completed system is tested in its operational environment.
- *System Evaluation.* After assembly, the system is ready for operation and evaluation.[24]

Exercise 10.3. For each of the following, define a minimal set of MC/DC (Modified Condition/Decision Coverage) tests. Why is your answer a minimal set?

a) `(!a)|(!b)|c`

Exercise 10.4. Suppose that there are 4 possible effects for text formatting and that you want to test all two-way interactions.

a) Create no more than 6 tests to test all 2-way interactions. Explain why your solution works.

b) Find a solution that requires no more than 5 tests.

Chapter 11

Measurement

> 'I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be."
>
> — *William Thomson, Lord Kelvin.*[1]

11.1 Introduction

Measurement and metrics provide a quantitative basis for decisions and predictions during every stage in the life of a software product, from the initial concept for the product through development, delivery, and maintenance to the end of support for the product. Informally, measurement is the process of associating metrics with something we want to measure. Decisions and predictions are based on data, in the form of values for the metrics.

For example, number of staff-days is a metric for the effort needed to implement a work item—if 2 developers work for 3 days, the effort is 6 staff-days. During iteration planning, effort estimates are used to choose work items to be implemented during the iteration. (The choice/decision of work items is actually based on two metrics: effort and priority.)

As another example, consider product failures after delivery, which can damage a company's z. Support teams carefully monitor metrics such as the number of trouble reports from customer sites and the time it takes to respond

to a trouble report. Customer-found defects trigger corrective maintenance of the product. In addition to fixing defects, the development team may need to measure and improve its practices so future products have better quality.

In previous chapters, the focus was on development, leading up to product delivery. This chapter introduces measurement, using examples from customer support and corrective maintenance after delivery. The term *maintenance* refers to two broad classes of post-delivery activities: *corrective maintenance* to respond to and fix defects; and *enhancements* to evolve the product.[2] With iterative development, all iterations evolve the product, so enhancement is an extension of evolutionary development.

As we shall see, product failures after delivery can be reduced if we

1. improve the verification processes,
2. which will lead to better products with fewer remaining defects
3. that will have fewer failures at customer sites,
4. resulting in fewer trouble reports and hence less need for corrective maintenance.

Lines 1-4 describe a quality improvement program. Section 11.3 introduces the various forms of software quality. Lines 1-3 touch on three forms of quality: process, product, and operations.

Metrics and measurement are essential elements of a quality improvement program. Section **??** deals with the measurement process. Metrics are used to set targets for improvement and track progress towards meeting the targets. See also Chapter **??**, where "How much?" questions are used to refine soft words like "better" and "fewer" into quantitative targets. The goal hierarchy in Fig. **??** touches on various forms of software quality.

To put things in perspective, better design and clean code may have a greater impact on customer satisfaction than improving verification processes. The point here is that, all other things being equal, improved verification practices can contribute to reducing the need for corrective maintenance.

11.2 What is Measurement?

Informally, a metric assigns numbers or symbols to properties of something we care about. This section begins with definitions of the terms measurement and metric. The selection of a suitable metric can be a challenge, as we shall see.

11.2.1 The Measurement Process

An overview of the measurement process appears in Fig. 11.2. Starting with the example in the figure, a program has a size that is measured in lines of code. That specific program has 750 lines. The program is an entity, size is an

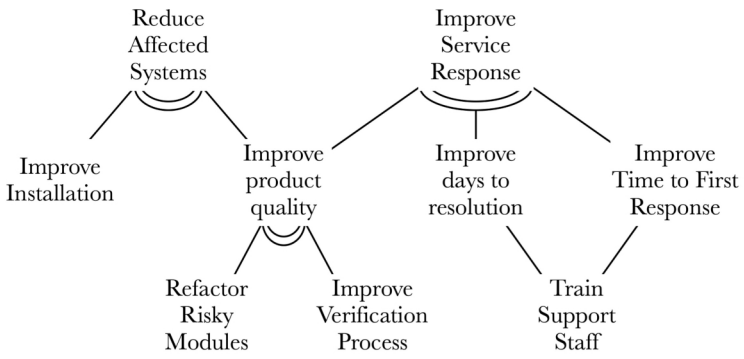


Figure 11.1: A goal hierarchy that touches on the various forms of software quality.

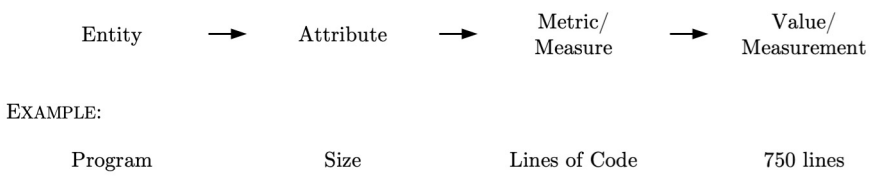


Figure 11.2: An overview of the measurement process.

attribute of the program, lines of code is a metric (also known as a measure), and 750 is the value of the metric.

Entities, attributes, and metrics are chosen based on the purpose of measurement. We design a *measurement process* as follows:

1. Identify a set of *entities* that represent things of interest. Such things include artifacts (e.g., code), activities (e.g., testing), and events (e.g., reviews).

2. Define *attributes* for each entity, representing properties to be measured; e.g., attributes effort and priority for an entity user story.

3. Select one or more *metrics* that associate numbers or symbols with an attribute; e.g., metric lines of code for attribute size of entity program.

4. Associate *values* with metrics, where the values reflects the real world.

Example 11.1. Height is an attribute of a person. Number of inches and number of centimeters are two different metrics for attribute height. With height, there is a clear distinction between the attribute and its associated metrics. □

We use the terms metric and measure interchangeably, as in

> Lines of code is a measure of program size.
> lines of code is a metric for program size.

Note that the term "measurement" is overloaded. As a process, *measurement* defines metrics, so numbers or symbols can be assigned to attributes. As a value, a *measurement* is a specific value for a metric. For example, lines of code is a metric; 500 lines and 750 lines are measurements.

Metrics for goals are useful for quantifying progress toward the gaol. A *criterion* is a specific value of a metric that serves as a reference for determining whether a goal has been achieved. If we want to increase revenues by 15%, then +15% is the criterion for metric revenues.

Direct and Derived Metrics

The value of a *direct* metric is independent of the value of any other attributes. The value of a *derived* metric depends on the values of one or more other attributes.

Example 11.2. Consider an entity representing a product that has been released and installed at customer sites. During the first three months after release, several defects have been found and fixed. The product team now wants to estimate the quality of the product. The team selects the following attributes and metrics:

Attribute	Metric	Kind	Description
installs	N	Direct	Number of installs
defects	D	Direct	Number of defects
quality	R	Derived	$R = D/N$

Metric R for attribute *quality* is a derived metric because it is defined in terms of metrics for two other attributes, *defects* and *installs*. Metrics D and$N$ are direct because they depend only on the measurement of a single attribute.

Note that the ratio of defects to installs is one of many possible metrics for product quality. A different metric, defect-removal efficiency, appears in Section 11.4. □

11.2.2 Three Common Metrics

To be useful, a metric must be consistent with our intuition about the real world. That is, a metric need to be chosen or designed to serve as a reliable basis for decisions and predictions. For example, if a suitably designed defect-rate metric assigns value 2% to product A and value 3% to product B, then we want these values to reflect our experience with products A and B. If not, we need to design a different metric.

The examples below illustrate some of the issues that arise with metrics. Lines of code is widely used, despite its limitations. Customer satisfaction

cannot be measured directly. Downloads are easy to measure, but do not predict customer satisfaction with a product. The corresponding lessons are as follows:

- A roughly right metric may be good enough; e.g., lines of code. Is there a better alternative?[3]
- An attribute may not be directly measurable; e.g., customer satisfaction. Is there another attribute that can serve as a proxy?
- An metric may be accurately measurable, but it may not fit the purpose/goal for making measurements; e.g., downloads and customer satisfaction. Choose another metric.

Measurement presupposes that the relevant data is available for associating values with metrics. Otherwise, we need to design derived metrics that can be evaluated using the available data.

Lines of COde: Flawed, But Useful

The main issue with lines of code is its dependence on the programming language. The "same" task can take more lines if it is written in C than it does in an object-oriented language like Java.

Line counts penalize code that is elegant and short, compared to code that is sloppy and verbose. There are also lesser issues about differences in programming style. Comments are sometimes handled by counting only non-comment lines.

Despite its limitations, lines of code is a helpful metricd for estimating program size. , Teams that use the same programming language and similar style guidelines can use line of code for meaningful comparisons.

Net Promoter Score is a Proxy for Customer Satisfaction

Net Promoter Score (NPS) is often used as a proxy for customer satisfaction. *NPS* is a measure of the willingness of customers to recommend a product, on a scale of 0 to 100. Higher scores are better. NPS can be measured through interviews. Example 11.3 illustrates the use of NPS.

Downloads Do Not Predict Satisfaction

Example 11.3. The Microsoft Access team was delighted that downloads of their new database product had jumped from 1.1 million to 8 million over 18 months. Then, they observed that customers were not using the product.

The team followed up with phone interviews and found that the Net Promoter Score was "terrible." The verbatim comments were "brutal."

The team fixed the product, including 3-4 design flaws and some pain points. They also made some training videos. The videos turned out to be most popular. The Net Promoter Score jumped an impressive 35 points.[4] □

- Total number of fixed customer-found defects in code or documentation
- Number that were dubbed "genuine" and reported for the first time
- Number that were rejected by the Level 3 support personnel
- Number of pointers to components touched by a first-time genuine defect
- Number of defects found in previous "fixes"
- Number of defects in code or documentation that were received
- Backlog of defects in code or documentation
- Total number of customer service requests that were closed
- Number that were for preventive service
- Number that were for installation planning
- Number that were for code or documentation; e.g., 2nd+ defect reports
- Number that were not related to IBM code or documentation
- Number of customer service requests handled by Level 2 support
- Number of days to resolution for requests handled by Level 2
- Number of users, a measure of the size of the installed base

Figure 11.3: Which of these customer-service metrics has the most effect on customer satisfaction? All 15 were managed and tracked.

11.2.3 Case Study: Customer-Support Metrics

The case study in Example 11.4

The case study in Example 11.4 serves two purposes. First, it illustrates that the right for the job may not be obvious. Second, it illustrates metrics related to corrective maintenance.

The authors of an IBM study note that, within the company, you were "likely to get two conflicting answers" if you asked which of the following metrics had a greater impact on customer satisfaction:[5]

- the number of problems on a product, or
- service call response time.

The study set out to address the larger question of which of the many metrics in use within IBM had the greatest impact on customer satisfaction.

Example 11.4. The IBM study was based on three years of actual data from service centers that handled customer calls. It examined 15 different metrics, for an operating systems product. The metrics related to customer-found defects and to customer service requests; see Fig. 11.3.

The study examined the correlation between the 15 metrics and customer satisfaction surveys. It found the greatest correlation between the following two factors and satisfaction surveys:

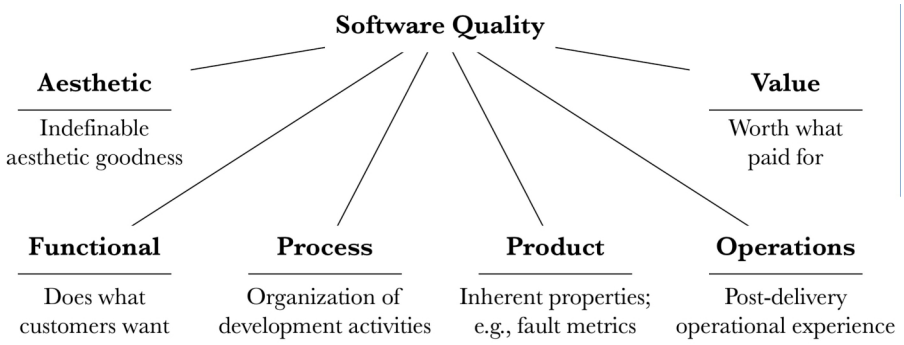


Figure 11.4: A model for software quality.

1. Number of defects found in previous "fixes" to code or documentation.

2. Total number of customer service requests that were closed.

The next two metrics were much less significant than the first two:

3. Total number of fixed customer-found defects in code or documentation.

4. Number of days to resolution for requests handled by Level 2.

□

Defective fixes during corrective maintenance are also referred to as *breakage*. Breakage is a significant dissatusfuer, Customers do not like it when something stops working.

11.3 Forms of Software Quality

The term "quality" is inherently difficult to define because different stakeholders view quality differently. Users may focus on whether a product does what they want; whether it works reliably; and whether it is worth its price. Developers may focus on the product's code: whether it meets its specification; whether it is free of defects; and whether it is clean or sloppy. Thus, while users on the behavior of the product as a black box, developers focus on the code inside the product. It might therefore be more appropriate to talk about "qualities" (plural), instead of "quality" (singular).

For our purposes, the term *software quality* is a general term for any of the following six forms of quality: functional, process, product, operations, aesthetic, and value. See Fig. 11.4.[6]

Functional Quality

Functional quality is the degree to which a software product does what the user wants. Specifically, it is the degree to which the product meets user requirements for functionality. For example, functional quality might be measured in terms of how well the product supports the use cases or scenarios for the product.

Process Quality

Process quality has two possible interpretations:

- *Effectiveness*. With this interpretation, process quality refers to the effectiveness of a process in organizing software development activities and teams. For example, iterative processes tend to be more effective than waterfall processes at delivering the right product on time and within budget.
- *Compliance*. With the compliance interpretation, process quality is the degree to which a process conforms to or follows the documented process. For example, is code being subjected to the level of testing specified by the documented process? If not, the actual process is not in compliance.

Note that process quality is relatively independent of the product being developed. Consider the process decision that all code must be reviewed before it becomes part of the code base. This decision is independent of the product.

Product Quality

Product quality refers to inherent properties of a product that cannot be altered without altering the product itself.[7] For example, the number of known defects in a product is a measure of product quality.

Operations Quality

Operations quality refers to a customer's experience with operating a product after the product is delivered. If a product fails during operation, the customer's operations experience will suffer and the product will be said to have poor operations quality.

Example 11.5. This example illustrates the distinction between product and operations quality. It mentions testing and code coverage, which are discussed in Chapter 10.

A team at Nortel spent months improving product quality:[8]

> "We spent almost 6 months in test between manual and automated testing. Then we went to the customer base, with [a code coverage] tool turned on ... what we found was that we only tested 1/10 of 1% of what they use. And of course it was a bad release in the customers' eyes."

In other words, the team spent 6 months improving product quality as measured by the number of defects in the source code. However, the defects they fixed through testing were largely in portions of the code that were not reached during operation. The operations quality was poor because defects remained in the portions of the code that customers did use.

For the next release the team focused on improving operations quality:

> "We changed the automation tools and manual testing to test what the customer used, about 1% of the code ... and the next release was a fantastic release in the customers' eyes.
>
> "Through this tool we also learned that the customer base rarely (aka almost never) used new features. It was all about everything that they currently use still working in the new release."

The conclusion is that customers care about operations quality, not overall product quality. (Developers typically focus on product quality.) The defects that matter to customers are the defects that they encounter when they use a product. □

Aesthetics

Aesthetic quality refers to the indefinable "I'll know it when I see it" goodness of a product. It is about perceptions and taste and cannot be measured directly. Aesthetic quality is included here for completeness.

Value

Value, *worth-what-paid-for*, and *customer satisfaction* are related notions of quality. They all relate to a person's willingness to pay for a product. Companies care a great deal about the extent to which their customers are satisfied. Like aesthetics, value (from the customer's perspective) is hard to measure directly.

In Example 11.3, Net Promoter Score (NPS) was used as a proxy for value. NPS is a measure of the willingness of customers to recommend a product, on a scale of 0 to 100; higher is better. Example 11.3 also illustrates that value can be related to other aspects of quality, such as product quality.

11.4 Product Quality: Metics for Defects

There is data to support the claim that better reviews, static analysis, and testing can lead to products that have fewer defects upon delivery. This claim will be quantified below, using a metric called defect-removal efficiency. This section begins, however, with a closer look at defects. In particular, the term "customer-found defects" bears scrutiny. What happens if two customers find the same defect? Customer support teams pay close attention to all defect

reports from customers. For development teams, however, a defect is a defect, no matter how many customers find it. The metrics in this section measure product quality.

Recall that a *defect* is a fault: a flaw or omission in source code, a design, a test suite, or some other artifact.

Severity of Defects

All defects are not equal: some are critical, some are harmless. Defects are often classified into four levels of decreasing *severity*

1. *Critical*. Total stoppage. Customers cannot get any work done.
2. *Major*. Some required functionality is unavailable and there is no workaround.
3. *Minor*. There is a problem, but there is a workaround, so the problem is an inconvenience rather than a roadblock.
4. *Cosmetic*. All functionality is available.

The severity levels may be identified either by number or by name. The lower the number, the more severe the defect. The levels are sometimes referred to as critical, high, medium, and low. Companies tend not to ship a product with known critical or major defects.

Customer-Found Defects (CFDs)

extensive list of metrics in Fig. 11.3 illustrates the care and attention paid to support requests from customers. Many of the metrics on list relate to defects. Project managers and developers, however, tend to prefer a short list of relevant metrics.

Example 11.6. The following list of metrics is from Fig 11.3:

- "Total number of fixed customer-found defects in code or documentation"
- "Number [of defects] that were dubbed 'genuine' and reported for the first time"
- "Number that were rejected by the Level 3 support personnel"
- "Number of defects in code or documentation that were received"
- "Backlog of defects in code or documentation"

In the second metric on this list, the phrase "dubbed 'genuine' " is open to interpretation. Presumably, it is there to ensure that the defect is genuinely about the given product. In the third metric, "Level 3" refers to levels in a customer-support organization, before a trouble report reaches the product developers. t strongly with customer satisfaction. □

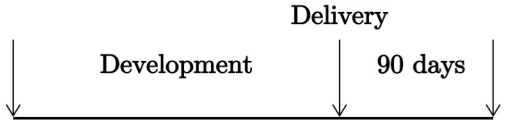


Figure 11.5: Defect-removal efficiency is the ratio of defects removed during development and the total number of defects removed during development and detected in the 90 days after delivery.

Having established that all defects are tracked, let us turn to CFDs. Let us classify a defect as a *customer-found defect (CFD)* if it satisfies the following:

1. *It is a product defect.* Not all customer trouble reports are about a defect in a given product. For example, the trouble may be that the product was not installed properly; say, it was misconfigured or has an interoperability issue with a third party product.

2. *It is the first report of this product defect.* Even if there are multiple reports, they are about the same defect in the product. A fix to the defect will handle all of the trouble reports. It therefore makes sense to classify only the first report as a CFD. (Meanwhile, the number of reports about a defect is a measure of operations quality, since it reflects the customer experience with the product.)

3. A potential third condition is that the defect is either a critical or a major defect.

Defect-Removal Efficiency

The idea is to measure how efficiently defects are detected and removed before the product is delivered. Here, efficiency is measured by the ratio *before*/*total*, *before* is the number of defects removed before delivery and *total* is the total number of defects before and after delivery. If 90 defects were removed before delivery and 10 defects remain in the product upon delivery, then the efficiency is 90%, from $90/(90 + 10)$.

The challenge with the above idea is that, at the time of delivery, we do not know the number of undiscovered defects in a product. Nor do we know the number of defects that will be detected in the future. What we can measure is the number of defects that are detected in a fixed interval after delivery. In Fig. 11.5, the fixed interval is the 90 days after delivery.

In practice, there is an initial surge of trouble reports after delivery, as customers use the product in ways that the developers may not have anticipated. The initial surge dies down as the early trouble reports are addressed, and

Techniques	Median DRE
Reviews alone	65%
Static Analysis alone	65%
Testing alone	53%
Reviews, Static Analysis, and Testing	97%

Figure 11.6: Defect-removal efficiency (DRE) for reviews, static analysis, and testing. Source: Capers Jones.

defects are fixed. The 90-day interval in Fig. **??** strikes a balance between getting early feedback and waithing long enough for trouble reports to die down.

Let the "total" number of defects be the number removed during development and the number detected during the 90 days after delivery. *Defect-removal efficiency* is the ratio of defects removed during development and the total number of defects.

Measuring Defect Removal Techniques

Defect-removal efficiency provides insights into the effectiveness of techniques for finding defects. Based on the data in Fig. 11.6, the combination of reviews, static analysis, and testing has a 97% removal efficiency. The individual techniques by themselves are no more than 65% efficient. The obvious conclusion is that we need to use a combination of techniques for defect removal.

11.5 Scales of Measurement

The following metrics have different kinds of values:

- Name of a product
- Severity level of a defect
- Release date
- Defect-removal efficiency
- Days to resolution of a trouble report

These metrics are on what we would call different "scales of measurement."

A *scale of measurement* consists of a set of values and a set of operations on the values. Note that "release date" and "days to resolution" belong on different scales becayse the former is a date and the latter is a number. We can subtract two dates to get a number (of days), but what does it mean to add two dates? A scale of measurement spells out the operations that are meaningful.

The *Stevens system* has four scales or levels of scales:

- *Nominal Scales.* Also known as *categorical scales* because values are typically used to group entities into categories; e.g., product names can be used to group trouble reports.
- *Ordinal Scales.* Ordinal values also correspond to categories, except that the categories are ranked; e.g., we can sort defects by severity level.
- *Interval Scales.* Dates are example. Values represent offsets from a chosen origin; e.g., the freezing point of water is the chosen origin (0 degrees) of the Celsius scale for termperature.
- *Ratios.* Ratios are used to normalize (adjust) measurements for comparisons across entities; e.g., defect-removal efficiency is a ratio that allows comparisons between products with different numbers of total defects.

Note: While the Stevens system has been challenged, it continues to be widely used.[9]

Absolute Scale

The Stevens scales, above, are in addition to the *absolute scale*, where values correspond to integers. Counting is on the absolute scale; e.g., the number of lines of code in a program. The term "absolute" comes from the scale having an absolute zero. It makes sense to talk of zero defects. Depending on the context, the absolute scale may or may not support negative values. It does not make sense for a program to have a negative number of defects.

Nominal or Categorical Scales

Values on a *nominal scale* form an unordered set. The only operation is comparison for equality between values. Let us refer to nominal values as *categories* because they are typically used to group entities into categories.

Once entities are categorized we can count the number of entities in a category. The counts are numbers, so they are on the absolute scale, not on the nominal scale. Numbers are ordered, so we can identify the *mode*, which is the category or categories with the highest count.

Example 11.7. Suppose that customer requests for product features come with the name of the requester. For simplicity, suppose requester names are drawn from the set The set of names

$$\{ \textit{Alice}, \textit{Bob}, \textit{Chris}, \textit{Dylan} \}$$

This set forms a nominal scale. Here, entities are feature requests, *requester* is an attribute, *name* is a metric, and the values of he metric are drawn from the set above. □

Variants of Example 11.7 include categories for developer names, implementation languages for modules, and geographic regions for installations of a product.

Caution. Categories are sometimes numbered. In this case, the numbers are simply symbols and have no other significance. For example, suppose we number smartphone colors:

1: Silver
2: Blue
3: Gold

Here, 1, 2, and 3 are alternative symbols for colors. The colors are in an arbitrary order, so the numerals need to be treated as unordered elements of a nominal scale.

Ordinal Scales

Values on an *ordinal scale* are linearly ordered. The only operation is comparison with respect to the ordering.

Example 11.8. Defect severity levels form an ordinal scale with the following set of values:

$$\{\ \textit{critical}, \textit{major}, \textit{minor}, \textit{cosmetic}\ \}$$

The values are linearly ordered by severity level:

$$\textit{critical} > \textit{major} > \textit{minor} > \textit{cosmetic}$$

Here, $x > y$ means that x has higher severity than y. Defect entities have an attribute for severity, These entities can be categorized by severity level. □

Example 11.9. The MoSCoW approach to prioritizing requirements (Section 6.3) is on an ordinal scale with the following categories:

$$\textit{Must-have} > \textit{Should-have} > \textit{Could-have} > \textit{Won't-Have}$$

Here, $x > y$ means that category x has higher priority than category y. □

We refer to ordinal values as categories, since ordinal values can be used to group entities, just like nominal categories. Thus, we can count the number of entities grouped in a category and determine the mode—the category with the most entities.

The ordering on categories allows us to determine the median (middle) category.

The linear ordering of categories permits conclusions about the position of a category relative to the ordering. The median is the middle category: half the categories are below, half are above. It is also meaningful to talk about quartiles, which partition the linear ordering into quarters.

Interval Scales

Dates are a prime example of the use of an interval scale. There is a fixed "distance" of one day between successive dates. All dates are measured relative to an origin or date zero.

Date zero is a chosen reference value. Internally on Unix, date zero is January 1, 1970. Microsoft Excel has used two date scales, one with an origin in 1900 and one with an origin in 1904. Each date therefore represents an offset or distance from the chosen date zero.

Generalizing from dates, an *interval scale* is based on a sequence of elements. The *distance* between successive elements is 1 unit; the distance between the elements at positions i and j in the sequence is $j-i$ units. One of the elements in the sequence is chosen as the *origin*for the scale. A value on this scale represents the distance between an element and the origin.

Example 11.10. Both Celsius and Fahrenheit temperatures are on interval scales. On the Celsius scale, temperatures are reported relative to zero degrees, the freezing point of water. □

Values are linearly ordered by their distance from the origin, positive or negative. Values can therefore be compared and hence sorted. An *interval* is defined by two values: a start value and an end value, with the end value greater than the start value. The length of the interval is obtained by subtracting the start value from the end value. It is meaningful to subtract one value from another and obtain a number (of units), but addition, multiplication, and division are meaningless. It is meaningful, however, to add an interval value and a number and get another value; e.g., think of slipping a deadline by 30 days.

The length of an interval is a number, so arithmetic on lengths is meaningful. We can computer the average number of days to resolve a customer issue. In

Ratio Scales

Values on a *ratio scale* are ratios of the form m/n, where both m and n are on a scale that supports arithmetic operations. Typically, both m and n are on an absolute scale.

Example 11.11. Defect-removal efficiency is the ratio of two numbers: the number of defects removed during development and the number of defects removed during development plus in the number removed during the first 90 days after delivery. (For more on defect-removal efficiency, see Section 11.4.)

Ratios provide useful metrics for making comparisons between entities. Suppose product A has 5 customer-found defects in the first 90 days after delivery and that product B has 10 customer-found defects in the corresponding time period. Did the developers of A do a better job of removing defects than the developers of B? Not necessarily. If A had 95 defects removed during development and B had 190, then the two products have the same defect-removal efficiency:

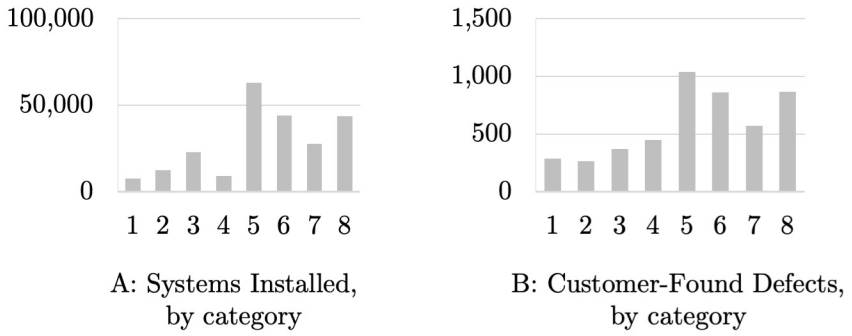


Figure 11.7: Numbers of (a) installs and (b) customer-found defects, by category. Categories are numbered due to space limitations. Both bar charts have the same categories.

$$95/(95+5) = 190/(190+10) = 95\%$$

If does product B has so many defects, is it poorly coded? This is a different question. Another ratio can help: defect density is the ratio of defects to program size. If B is twice the size of A, then the two would have the same defect density. □

Ratios are numeric, so a ratio scale supports numeric operations such as comparison, addition, subtraction, multiplication, and division.

11.6 Displaying Data

Visual representations of data are routinely used in dashboards that summarize the status of a software project. Charts and plots can readily convey information that could get lost in a mass of numeric data.

11.6.1 Bar Charts

Bar charts are useful for displaying information that is classified by category; see Fig. 11.7. In its simplest form, a bar chart has categories are along the horizontal axis. There is a vertical bar for each category and the height of the bar represents a value associated with the category.

Categories are unordered on a nominal scale, so their horizontal position can be changed without losing information. Categories are linearly ordered on an ordinal scale; that linear ordering determines their position along the horizontal axis.

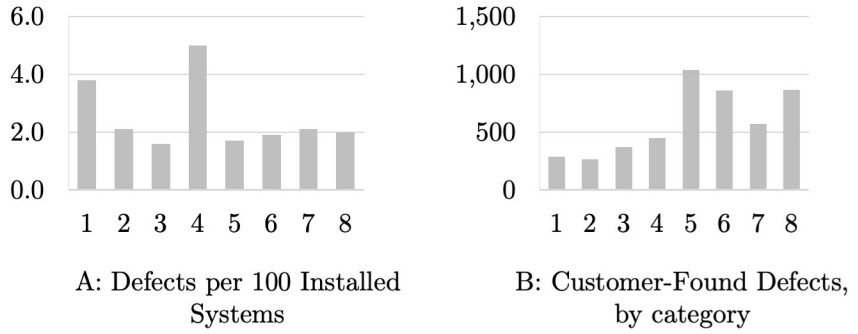


Figure 11.8: Numbers of (a) installs and (b) customer-found defects, by category. Categories are numbered due to space limitations. Both bar charts have the same categories.

The following example uses bar charts to s support the following observations:

- The number of customer-found defects is not a suitable metric of software quality. Instead, the number of defects reflects the number of installations (installs) of a product.
- The ratio of customer-found defects to installs does correlate with quality. It correlates with operations quality, as we shall see in Section 11.7.

Example 11.12. The bar charts in Fig. 11.7 display (a) the number of installations (installs) and (b) the number of customer-found defects for a sequence of releases of a product. Both bar charts are for the same sequence of releases. The releases are represented on the horizontal axis by the categories 1-8. The heights of the vertical bars are proportional to the counts in each category.

Note that the heights of the corresponding bars in both bar charts are very similar. As the number of installs goes up, so does the number of customer-found defects. This relationship between installs and defects has been observed for other products as well.

The relationship between defects and installss is easer to spot if consider the ratio of defects to installs. The bar chart in Fig. **??**(a) shows the ratios for the releases. For ease of comparison, Fig. 11.8(b) repeats Fig. 11.7(b).

Except for categories 1 and 4, the releases have around 2 defects per 100 installs. Sure enough, "dot-oh" releases: category 1 represents R2.0 and category 4 represents R3.0. The team improved its releases: categories 6 and 7 are for R4.0 and R5.0, respectively. □

11.7 Case Study: An Operational Quality Metric

The derived metric in the next example was designed to measure the operational quality of products across Avaya. The metric, called CQM (Customer Quality Metric), allowed quality comparisons between different products because it normalized for variables such as product popularity (the number of installations of a product) and product maturity.

Example 11.13. The CQM metric in this example was used to drive a company-wide effort to improve operational quality at Avaya. We need to distinguish between a product and and an installation of the product at a customer site, so let use the terms *install* and *system* to refer to an installation of a product. It is convenient for readability to have both terms install and system. The metric based on the ratio of *affected* installs, where affected means that the install reports a customer-found defect. The lower the ratio of affected installs, the better the operational quality of the product.

The use of a ratio allows comparisons between products with widely differing numbers of installs. CQM works with the ratio of the number of affected installs to the total number of installs. Based on empirical data. affected installs rise linearly with total installs. The ratio of affected to total installs represents the likelihood (probability) that an install will be affected.

Two key questions remain:

- Product Maturity. Operational quality improves as a product matures. Defects get reported and fixed. Later installs go more smoothly. Let a *maturity period* be the time interval that starts with the release of the product. How do we compare the quality of products that have different maturity periods? The answer is to fix the maturity period to be the first m months after release, for each product. Thus, independent of how long a product has been on the market, the CQM calculation is based on installs that were affected in the first m months after release.
- *Reporting Interval*. How do we count the number of affected installs? As shown in Fig. 11.9, installs occur at different times during the product maturity period. A system that has been installed for 6 months is more likely to encounter a defect than a system that has been installed for just 1 month. For a true comparison, all installs need to have the same *reporting interval*, which is the first n months after an install. In Fig. 11.9, the reporting interval is shown by a thick line.

The above discussion of product-maturity period and reporting interval sets the stage for the definition of CQM. The *n-month CQM* metric for a product is the ratio of installs within the first m months after release that report a customer-found defect within the first n months after installation.

At Avaya, year-over-year improvements in CQM were achieved by improving process quality: by improving static analysis, code coverage, reviews, and

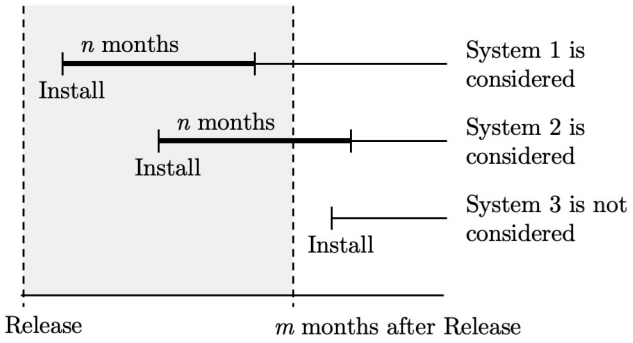


Figure 11.9: To be considered for the n-month CQM operational quality metric for a product, a system must be installed within the product maturity period, which is the first m months after product release. The thick lines indicate the reporting interval, which is the first n months after install.

testing. The product maturity period was $m = 7$ months. There were three reporting intervals: $n = 1$ for an early measure; $n = 6$ months for a relatively stable measure; and $n = 3$ months for an intermediate value.[10] □

Chapter 12

Epilogue

> "... if I'd a knowed what a trouble it was to make a book I wouldn't a tackled it and ain't agoing to no more."
>
> — *Huckleberry Finn*[1]

12.1 Introduction

12.2 Continuous Deployment

12.3 Attack Trees: Identifying Security Threats

Attack trees model security threats. Once security holes are identified, the holes can ideally be closed. If the holes cannot be closed entirely, defensive measures can be taken to reduce the risk of an attack through those holes.

Think Like an Attacker

Technically, an *attack tree* is a tree-structured goal hierarchy, where the top-level goal is to break into a system; see Fig. 12.1. The general advice from security experts is to anticipate a hacker's moves. In short, think like an attacker. The top-level goal in an attack tree is therefore the attacker's goal of breaking into a system.

Hackers typically launch attacks in ways that the system designers never anticipated. Therefore, it pays to make an attack tree as complete as possible.

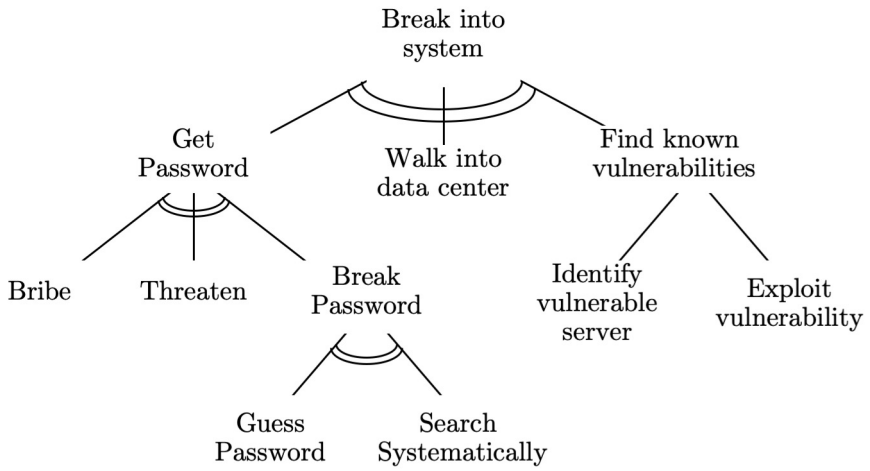


Figure 12.1: An attack tree. For simplicity, this tree includes a representative rather than a complete set of security threats.

Note that security threats can be social as well as technical. Social threats include phishing attacks, bribery, deception, and so on.

Finding Possible Attacks

While building an attack tree, it is better to brainstorm and include ways that are later dismissed as being impossible. Otherwise, we run the risk of missing ways that later turn out to have security holes.

Once an attack tree is built, impossible ways can be pruned by making two passes over the tree: a bottom-up pass to mark possible nodes and a top-down pass to prune impossible ways. The passes are as follows:

1. Mark each leaf p fif that leaf is possible; that is, the goal at the leaf is achievable. Otherwise, mart it i for impossible.

2. Working bottom-up, mark each *or*-node p for possible if any of its children is marked p. Mark each *and*-node p if all of its children are marked p. Otherwise, mark a node i for impossible. Continue until all nodes are marked.

3. Working top-down from the root, delete a node and its subtree if the node is marked i. Continue until all impossible nodes and their subtrees are deleted.

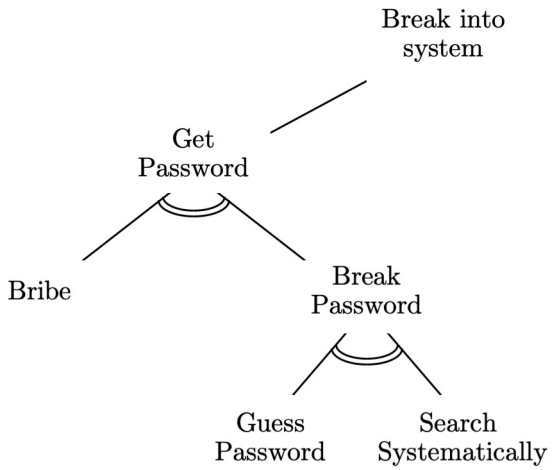


Figure 12.2: The result of deleting impossible nodes from the attack tree in Fig. 12.1.

Example 12.1. Consider the attack tree in Fig. 12.1. Suppose for this example that the following leaf goals are impossible:

- *Threaten.* It is not possible for an attacker to get the password by threatening or blackmailing a system administrator.
- *Walk into data center.* The data center is physically isolated enough that it is not possible for someone to walk into the data center.
- *Identify vulnerable server.* From the context at the parent node, the full version of this leaf goal is "Identify server with known vulnerability." In practice, hackers have exploited servers running old software with known security holes. This data center, however, religiously applies all security updates, so it is not possible for an attacker to fund a server with known vulnerabilities.

Under different circumstances, it may be possible to threaten an administrator into revealing a password, walk up to a server in a data center, or find a vulnerable server. In this example, these are impossible goals. All other leaves represent possible goals. Once impossible nodes are identified and deleted we will be left with the attack tree in Fig. 12.2.

Working bottom-up, "Threaten" is one of three children of an *or*-node. The other two children are marked possible, so "Get password" is possible. Meanwhile, "Identify vulnerable server" is at a child of an *and*-node. Since we have assumed that this leaf goal is impossible, its parent goal, "Find known vulnerabilities," is impossible. The root is an *or*-node, with three children. The

first child, "Get password," is possible, so the root is marked possible. The other two children of the root are impossible.

Working top-down, the root is possible, so it remains. "Get password" also remains, but the other two children of the root are deleted along with their subtrees, since they are marked impossible. The leaf node for "Threaten" is also deleted. All other nodes remain. □

Estimating the Cost of an Attack

Cost information can be used to do a more granular analysis of security threats than doing a binary analysis of possible and impossible threats. The idea is to begin by associating cost estimates with the leaf goals in an attack tree. The estimates can then be propagated bottom-up to the root to estimate the minimum cost of an attack. With this approach, domain expertise is required for making the initial cost estimates for the leaf goals. For example, what is the cost of systematically breaking a password? Meanwhile, a facilities expert may need to be consulted for estimated for someone to gain physical access to a data center.

Once the costs at the leaves are estimated, they can be propagated up the other nodes as follows:

- The estimated cost at an *or*-node is the minimum of the costs at its children.
- The estimated cost at an *and*-node is the sum of the costs at its children.

Example 12.2. Without specific information about the context of the system modeled by the attack tree in Fig. 12.1, we can only guess at the costs of accomplishing the leaf goals in the tree. The costs in the annotated tree in Fig. 12.3 may therefore be unrealistic. The numbers in parentheses represent the costs. The units are thousands of dollars.

In the left subtree, the cost of systematically searching for a password is 30 units, which is lower than the alternative of an ad hoc guesswork approach. The cost at their parent node is therefore 30, since the parent is an *or*-node. In the right subtree, the cost of finding a vulnerable server is 60 and the cost of exploiting a known vulnerability is 10. The cost at their parent node is the sum, 70, since the parent is an *and*-node. The root is an *or*-node, so the cost at the root is 30. □

Once costs have been propagated bottom-up from the leaves to the root, we can work down from the root to find the attack that matches the cost at the root. In general, an attack will be represented by a subtree because all subgoals of an *and*-goal G need to be satisfied for G to be satisfied. Starting at the root, least cost attack can be determined top-down as follows:

1. Begin by selecting the root.

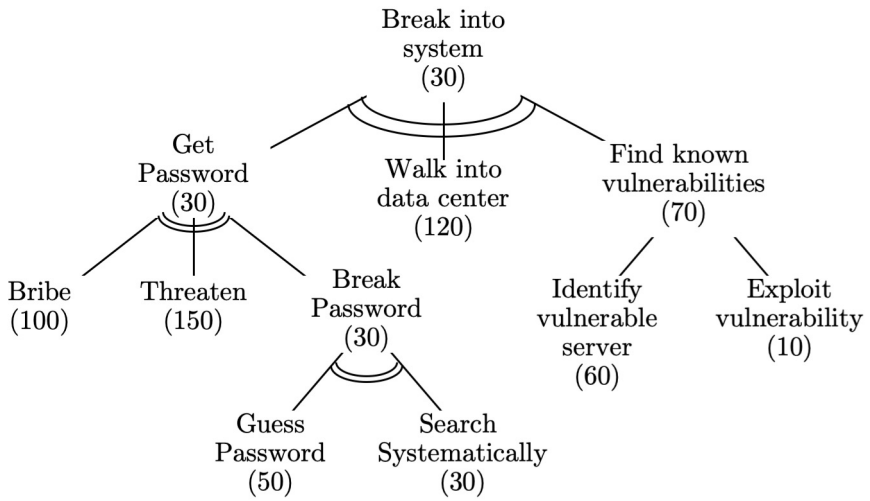


Figure 12.3: Estimating the cost of an attack, based on the attack tree in Fig. 12.1.

2. If a selected node n is an *or*-node, then select one of the children of n with the lowest cost.

3. If a selected node n is an *and*-node, then select all of its children.

4. End when there are no more nodes to select.

Example 12.3. For the annotated attack tree in Fig. 12.3, the attack consists of a path from the root to the leaf for systematically searching for a password. The tree is a path, since all the nodes along the path happen to be *or*-nodes. □

Appendices

Notes

Index

Notes

Notes for the Preface

[1]See Guideline 9 in the 2014 IEEE-ACM software engineering curriculum [99, p. 43].

[2]The ACM and IEEE curriculum guidelines for both computer science [2, p. 174] and software engineering [99, p. 45] strongly recommend a significant team project: "the best way to learn to apply software engineering theory and knowledge is in the practical environment of a [team] project."

Notes for Chapter 1

[1]Margaret Hamilton's remarks about the early use of the term software engineering are from a 2014 interview [88].

[2]See SWEBOK 3.0 [40, p. 29] for the IEEE definition of software engineering. The original source is SEVOCAB, the Software and Systems Engineering Vocabulary `www.computer.org/sevocab`. The Association of Computing Machinery (ACM) and the IEEE Computer Society are the two main professional organizations for computer science.

[3]Example 1.3, with the speech-therapy program, is from an article in 1843 magazine [129].

[4]See [83] for the algorithms behind Netflix's recommender system.

[5]In a classic paper, Brooks [45] argues that software is intrinsically complex.

[6]Miller [143] notes that "the accuracy with which we can identify absolutely the magnitude of a unidimensional stimulus variable ... is usually somewhere in the neighborhood of seven." Unidimensional refers to like chunks of information, such as bits, words, colors, and tones. Faces and objects differ from one another along multiple dimensions, hence we can accurately distinguish hundreds of faces and thousands of objects.

[7]Example 1.8 is based on a congressional hearing into the loss of Mariner 1 [193, p. 100-101].

[8]For more on the issues that arise during testing, see the "practice tutorial" by Whittaker [206].

[9]Software project management builds on project management in general. The Project Management Institute publishes PMBOK, a guide to the project management body of knowledge [165].

[10]Martin Barnes is credited with creating the Iron Triangle for a 1969 course [203]. Trilemmas have been discussed in philosophy for centuries.

[11]The Therac-25 accidents were never officially investigated. Nancy Leveson and Clark Turner [131] relied on court documents, correspondence, and information from regulatory agencies to infer information about the manufacturer's software development, management, and quality control practices. See also [130].

[12]The quotes from the ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice [1] are reprinted by permission. ©1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.

[13]Margaret Hamilton's remarks about the use of the term software engineering are from a 2014 interview [88].

[14]Boehm [34] writes, "On my first day on the job [around 1955], my supervisor showed me the GD ERA 1103 computer, which filled a large room. He said, 'Now listen. We are paying $600 an hour for this and $2 an hour for you, and I want you to act accordingly.'"

[15]The focus of the 1968 NATO conference was on the "many current problems in software engineering." See [151, p. 13-14].

[16]The Bauer definition of software engineering is by Fritz Bauer [18], one of the organizers of the 1968 NATO workshop [151]. The SEI definition was created by the Software Engineering Institute for planning and explaining their own activities [73]. The IEEE definition is from [40, p. 29].

[17]Parnas [159, p. 415] credits Brian Randell with the characterization of software engineering as "multi-person development of multi-version programs."

[18]The list of forces in Exercise 1.3 is from an ACM SIGSOFT webinar by Grady Booch

[19]The code in Exercise 1.5 is based on an example in [128] of function self-application. Apologies to Henry Ledgard for taking his code out of context.

Notes for Chapter 2

[1]The opening quote is from McIlroy, Pinson, and Tague's foreword to a 1978 special issue of the *Bell System Technical Journal* on Unix [142].

[2]"Who will do what by when and why" is a variant of Boehm's [32, p. 76] "Why, What, When, Who, Where, How, How Much."

[3]The Unix maxims are from [142]. The first two are based on the opening quote of the chapter.

[4]Agile Manifesto: ©2001 by Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. The Agile Manifesto may be freely copied in any form, but only in its entirety through this notice. `http://agilemanifesto.org/`

[5]Fowler [77].

[6]The definition of agile method follows the Agile Alliance's description of agile software development [3].

[7]The Agile Alliance provides a short history of "Agile" [3].

[8]The Tidal Wave reference is from the title of an internal Microsoft memo by Gates [79]. The sleeping giant allusion is from Bill Gates's opening comments at Microsoft's December 1995 analysts conference [60, p. 109]

[9]Bill Turpin: "The original way we came up with the product ideas was that Marc Andreessen was sort of our product marketing guy. He went out and met with lots of customers. He would meet with analysts. He would see what other new companies were doing." [60, p. 251]

[10]Iansiti and MacCormack [97] note that companies in a wide range of industries "from computer workstations to banking" had adopted iterative product development to deal with uncertain and unexpected requirements changes.

[11]For the risks faced by the Netscape Communicator 4.0 project, see the remarks by Rick Schell, senior engineering executive [60, p. 187].

[12]The "Definitive Guide" by Scrum's originators, Ken Schwaber and Jeff Sutherland [186] is the primary reference for Section 2.3.

[13]The 2019 State of Agile Survey [56] found that 58% of agile projects used Scrum by itself; 85% used Scrum or a hybrid.

[14]Beck and Andres [20] open with, "Extreme Programming (XP) is about social change." The premise of XP is from [20, ch. 17].

[15]Beck [19] briefly describes the roots of XP and provides references.

[16]The 3C acronym for Card, Conversation, Confirmation is due to Ron Jeffries [107].

[17]The user story template in Fig. 2.7 is attributed to Connextra, a London startup.

[18]Cockburn and Williams [53].

[19]Wray [213] reviews work on pair programming in a position paper prompted by his own experience with pair programming.

[20]Based on interviews, surveys, and expert panels between 1996 and 1998, MacCormack [133] measured (1) *product quality*, defined as a combination of performance, functionality, and reliability; and (2) *team productivity*.

[21]Fowler [75] attributes the acronym *yagni* to a conversation between Kent Beck and Chet Hendrickson, in which Hendrickson proposed a series of features to each of which Beck replied, "you aren't going to need it."

[22]In the second edition of *Extreme Programming Explained* [20, ch. 7] , Kent Beck notes that some of the teams misinterpreted the first edition as recommending deferring design until the last moment—they created brittle poorly

designed systems. The quote is from the same chapter.

[23]See the second edition of *Extreme Programming Explained* [20, ch. 7].

[24]In introducing XP, Beck p19] writes, "Rather than planning, analyzing, and designing for the far-flung future, XP exploits the reduction in the cost of changing software to do all of these activities a little at a time, throughout software development."

[25]Benington [23] is the earliest known published account of a waterfall process. The diagram in Fig. 2.9 is based on an influential 1970 paper by Winston W. Royce [172]. While Royce is often credited as the source for waterfall processes, the paper includes, 'In my experience, however, the simpler [waterfall] method has never worked on large software development efforts."

[26]The grow analogy for iterative processes is due to Brooks [45].

[27]The committee chairman's opening remarks are from [194, p. 2]. For the extent of system and end-to-end testing, see [194, p. 57]. The cost of the website is from [195, p. 19].

[28]The chart on the cost of fixing a defect is due to Boehm [34, 28].

[29]During a 2002 workshop [180], projects from IBM Rochester reported a 13:1 in the cost of a fix between coding and testing, and then a further 9:1 between testing and operation. Toshiba's software factory with 2,600 workers reported a 137:1 ratio for the time needed to fix a sever defect before and after shipment.

[30]Madden and Rone [134] describe the iterative process used for the parallel development of the hardware, software, and simulators for the Space Shuttle.

[31]US government contracts promoted the use of waterfall processes through standards such as Military Standard MIL-STD-1521B dated June 4, 1985 [192].

[32]Herbert Benington's 1956 paper [23] was reprinted in 1983 with a fresh Foreword by the author, in which he noted, "I do not mention it in the [1956] paper, but we undertook the programming only after we had assembled an experimental prototype."

[33]The V-shaped process diagram for SAGE in Fig. 2.11 is a redrawing of the original diagram. Paraphrasing Benington [23], a programmer must prove that the program satisfies the specifications, not that the program will perform as coded. Furthermore, "test specifications ... can be prepared in parallel with coding."

[34]The right-product/product-right characterization of validation and verification is due to Boehm [29].

[35]Barry Boehm introduced the spiral framework in the 1980s [31]. The treatment in Section 2.8 follows [33], where Boehm emphasizes that the framework is not a process model. He describes it as a "risk-driven process model generator."

[36]Example 2.8 is due to Boehm [35], who compares Total (up-front) and Incremental (spiral) commitment of funds. The Incremental approach in Example 2.8 achieved twofold improvement in 42 months for an overall investment of $1B (B for billion). The winning bidder of a Total commitment project promised eightfold improvement in 40 months for $1B, but delivered only twofold improvement in 80 months for $3B.

[37]2019 State of Agile Report [56].

[38]Jalote [106, p. 37] describes the Infosys development process. Infosys is known for its highly mature processes; specifically, at the time, Infosys was a CMM level 5 company.

Notes for Chapter 3

[1]Fred Brooks [45].

[2]The definition of the term requirement is adapted from Zave's description of requirements engineering [**??**].

[3]The iterative process in Fig. 3.2 is adapted from the Fast Feedback Cycle in De Bonte and Fletcher [63].

[4]Mavin??

[5]IEEE Standard 830-1998 includes the characteristics of a good SRS [98, p. 4-8] and several templates for writing an SRS [98, p. 21-26.]

[6]Reliability was an overriding concern during the development of the flight-control software for the mission that took the rover Curiosity to Mars. Holzmann [93] describes the validation and verification techniques that were used for the software.

[7]Hormby [94] traces the twists and turns in the story behind the Sony Walkman.

Notes for Chapter 4

[1]Carl Rogers [171] cites "a natural urge to judge, evaluate, and approve (or disapprove) another person's statement" as a major barrier to communication.

[2]Example 4.1 on the failed BBC Digital Media Initiative is based on a report by the UK National Audit Office [150] and a news report [82].

[3]Before starting Intuit, Scott Cook worked at Proctor & Gamble (P&G), where he "learned an encyclopedia's worth of business. ... P&G's obsession with customers resonated with Cook." [187, p. 6]. The quote about basing future products on customer actions, not words, is from [187, p. 221].

[4]The classification of needs in Section 4.2 is adapted from Sanders [175, 176]. She writes about a "shift in attitude from designing **for** users to one of designing **with** users" [176].

[5]The prototype Lunar Greenhouse at the University of Arizona has a a hydroponic plant-growth chamber. It is intended as a bio-regenerative life support system.

[6]Carl Rogers [171].

[7]Griffin and Hauser [85] refer to Voice of the Customer as an industry practice.

[8]For Intuit's Design for Delight: see for example Ruberto [173].

[9]Rabinowitz [167]. outlines the development of Intuit's QuickBooks app for the iPad.

[10]SMART criteria originated in a business setting; they are attributed to Peter Drucker's management by objectives. The acronym SMART is attributed to George T. Doran [66].

[11]The temporal classification of goals is from Dardenne, Lamsweerde, and Fickas [62]. See Lamsweerde [**??**] for a retrospective account.

[12]"Lamsweerde, Darimont, and Massonet [199] write, "*why* questions allow higher-level goals to be acquired from goals that contribute positively to them. ... *how* questions allow lower level goals to be acquired as sub-goals that contribute positively to the goal considered."

[13]The INVEST checklist for user stories is attributed to William Wake. `https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/` .

[14]Dan North [154] writes, "[Starting with user stories, Chris] Matts and I set about discovering what every agile tester already knows: A story's behaviour is simply its acceptance criteria .... So we created a template to capture a story's acceptance criteria. ... We started describing the acceptance criteria in terms of scenarios, which took the 'Given-when-then' form."

[15]For an in-depth treatment of scenarios, see the book on Scenario-Focused Engineering by De Bonte and Fletcher [63].

Notes for Chapter 5

[1]Jacobson, Spence, and Kerr [105] promote use cases for "not just supporting requirements, but also architecture, design, test, and user experience."

[2]Use cases were first presented at OOPSLA '87 [102]. Jacobson [103] provides a retrospective.

[3]The ATM use case in Example 5.6 is based on a fully worked out use case in Bittner and Spence [26], which Ivar Jacobson, the inventor of use cases, called "*THE* book on use cases" [103].

[4]As part of a discussion of user interface design, Constantine and Lockwood distinguish between intentions, interactions, and interfaces [57].

[5]The suggested order for evolving a use case is motivated by Cockburn [51, p. 17].

[6]Petre [161, p. 728] conducted "interviews with 50 professional software engineers in 50 companies and found 5 patterns of UML use," ranging from no use of UML (70%) to selective use (22%) and wholehearted use (0%). Several users mentioned informal use of use cases. Only one of the 50 developers found use case diagrams to be useful.

[7]In a chapter entitled "Here There Be Dragons," Bittner and Spence [26] note that the behavior of most systems can be specified without inclusions and extensions.

[8]For a discussion of use cases and user stories, see Cockburn [52] and the accompanying comments.

[9]Beck [19] introduced user stories along with XP.

[10]Jacobson, Spence, and Kerr [105].

[11]The description in Exercise 5.4 is adapted from the "Background and functionality" section of the Wikipedia entry for HealthCare.gov. `https://en.wikipedia.org/wiki/HealthCare.gov` . Text used under the Creative Commons CC-BY-SA 3.0 license.

Notes for Chapter 6

[1]Tversky and Kahneman [190]. Kahneman won the 2002 Nobel Prize in Economic Sciences for his insights into "human judgment and decision making under uncertainty."

[2]Sanders [175].

[3]New York Times, June 3, 2017, p. B1.

[4]The terms cognitive bias and anchoring are due to Amos Tversky and Dianiel Kahneman; see 190].

[5]Example 6.2 is based on a case study by Aranda and Easterbrook [10].

[6]The Quote Investigator [166] traces the saying, "It is difficult to make predictions, especially about the future," to the autobiography of the Danish politician Karl Kristian Steincke, where the quote appears (in Danish), without attribution.

[7]The quote about group wisdom is from Aristotle [12, Book 3, Part 11].

[8]Smith and Sidky [182, ch. 14].

[9]Dalkey and Helmer [61] describe the Delphi method, which was designed at RAND corporation in the 1950s.

[10]See Helmer [89] for a retrospective of the Delphi method.

[11]Boehm [30, p. 335] is credited with the Wideband Delphi method.

[12]See Cohn [55, ch. 6] for Planning Poker.

[13]Three-point estimation is is based on a statistical model developed for time estimates in conjunction with an operations research technique called PERT. Moder, Phillips, and Davis [148, ch. 9] discuss the statistical underpinnings of the weighted-average formula for Three-Point Estimation. PERT (Program Evaluation and Review Technique) is a project-management tool that was developed for the US Navy in the 1950s.

[14]Clegg, Barker, and Barker [50, p. 27] describe MoSCoW prioritization.

[15]Some companies—Avaya is an example—have internal standards into prodding product teams into giving priority to a range of quality attributes.

[16]The two-step prioritization based on value, cost, and risk is based on Cohn [55, ch. 9].

[17]Herzberg [91, ch. 6] et al. interviewed 200 engineers and accountants to test the hypothesis that people have two sets of needs: "the need as an animal to

avoid pain" and "the need as a human to grow psychologically." They asked about times when the interviewees felt especially good about their jobs and probed for the reasons behind the good feelings. In separate interviews, they asked about negative feelings about the job. The results from the study were that satisfiers were entirely different from dissatisfiers. Herzberg et al. used the term "motivators" for job satisfiers and the term "hygiene" for job dissatisfiers. The corresponding terms in this chapter are *attractors* and *expected* features.

[18]Kano [115] is the source for the treatment of Kano analysis in Section 6.4. The oft-cited paper by Kano et al. [116] is in Japanese.

[19]Kano analysis is included in De Bonte and Fletcher's [63] description of Scenario Focused Engineering at Microsoft.

[20]Clippy was included in Microsoft Office for Windows versions 97 through 2003 [209]. A USA Today article dated February 6, 2002 noted, "The anti-Clippy site has gotten 22 million hits since launching April 11." [197].

[21]See the TeXShop Change History [188].

[22]Lamsweerde [**??**] provides a guided tour of research on goal-oriented requirements engineering. Basili and Weiss [16] advocate the use of goals to guide data collection and measurement. Schneier [178] introduces attack trees for identifying potential security attacks.

[23]Jørgensen [113] reviews "what we do and don't know about software development effort estimation." See also the friendly debate between Jørgensen and Boehm [114] about expert judgment versus formal models.

[24]Diagrams similar to the Cone of Uncertainty were in use in the 1950s for cost estimation for chemical manufacturing [153]. Boehm [30, p. 311] introduced the Cone as a conceptual diagram for software estimation. Supporting data cane much later; see Boehm et al. [36].

[25]Results from numerous studies, going back to the 1960s, support the observation that there are order of magnitude differences in individual and team productivity. From early studies by Sackman, Erikson, and Grant [174], "one poor performer can consume as much time or cost as 5, 10, or 20 good ones." McConnell [139] outlines the challenges of defining, much less measuring, software productivity.

[26]Boehm and Valerdi's [37] review of the Cocomo family of formal models includes some historical perspective on models.

[27]Walston and Felix [202].

[28]Boehm [30].

[29]Boehm and Valerdi [37] note that "although Cocomo II does a good job for the 2005 development styles projected in 1995, it doesn't cover several newer development styles well. This led us to develop additional Cocomo II-related models."

[30]Jørgensen and Boehm [114] debate which is better: formal models or expert judgment.

Notes for Chapter 7

[1] "A map is not the territory," is a variant of "the map is not the thing mapped." attributed to Eric Temple Bell. [207]"

[2] The SEI has compiled a list of "modern, classic, and bibliographic definitions of software architecture." [183]. The list has 3 modern, 9 classic, and numerous bibliographic definitions.

[3] Kazman and Eden's [118] comments on the usage of the terms architecture, design, and implementation were based on the experience at the Software Engineering Institute (SEI). They propose the following distinction between architecture and design: architecture is non-local and design is local, where non-local means that the specification applies "to all parts of the system (as opposed to being limited to some part thereof)."

[4] Klein and Weiss [120] treat architecture as a subset of design: "Architecture is a part of the design of the system; it highlights some details by abstracting away from others."

[5] The cross section of the Hagia Sophia in Fig. 7.2 is from Lübke and Semrau [**??**]; see `https://commons.wikimedia.org/wiki/File:Hagia-Sophia-Laengsschnitt.jpg`.

[6] The diagrams of the pendentives in Fig. 7.3 is adapted from Viollet-le-Duc [**??**]; see
`https://commons.wikimedia.org/wiki/Category:Pendentives`.

[7] From Vitruvius's *de Architectura*, Book 1, Chapter 3, Verse 2: "All these should possess strength, utility, and beauty. Strength arises from carrying down the foundations to a good solid bottom, and from making a proper choice of materials without parsimony. Utility arises from a judicious distribution of the parts, so that their purposes be duly answered, and that each have its proper situation. Beauty is produced by the pleasing appearance and good taste of the whole, and by the dimensions of all the parts being duly proportioned to each other."
`http://penelope.uchicago.edu/Thayer/E/Roman/Texts/Vitruvius/1*.html`

[8] According to a Wikipedia article, the parable of the blind men and an elephant is an ancient Indian tale. A version appears in a Buddhist text, *Udana 6.4*, dated to about mid 1st millennium BCE.

[9] Rather than add to the proliferation of definitions of software architecture, we follow Bass, Clements, and Kazman [17]. The authors were all at the Software Engineering Institute when they published the first two editions of their book.

[10] Views have been incorporated into the standards IEEE 1471 and ISO/IEC/IEEE 42010.

[11] The 4+1 grouping of architectural views is due to Kruchten [122].

[12] See [38] for a user guide by the original authors of UML. The preface to the user guide includes a brief history of UML. Grady Booch, and James Rumbaugh [38] created an early draft of UML in 1994, based on a combination of their object-oriented design methods. They were soon joined by Ivar Jacobson, and expanded the UML effort to incorporate his methods. UML 1.1 was adopted as a standard by Object Management Group (OMG) in 1997. The UML 2.0 standard was adopted in 2005.

In a 2006 interview, Jacobson observed, "UML 2.0 has become very large, it has become heavy, it's very hard to learn and study. [104]" UML has strong name recognition, but actual usage lags. A 2013 survey by an author of UML books found that, while all 162 respondents had heard of UML, "Only 13% found UML to be very useful, and 45% indicated that UML was useful but they could get by without it. A further 20% found it "more trouble than it was worth." [7]

[13]The concerns in Fig. 7.10 are from a 2005 survey of Microsoft architects, developers, and testers by LaToza, Venolia, and DeLine [127] .

[14]The principle of information hiding was explored by David Parnas in the early 1970s [156].

[15]McIlroy and Pinson [142] note "a number of maxims [that] explain and promote" Unix style.

[16]The first guideline for module design is from Parnas [156]. The remaining guidelines are based on Britton and Parnas [43, p. 1-2].

Notes for Chapter 8

[1]Plato's observations about the ideas of beds and tables are from *The Republic*, Book X. circa 380 BCE.

[2]The definition of patterns is due to Alexander et al. [6]; see their Foreword.

Gamma et al. note that "The Alexandrian point of view has helped us focus on design trade-offs—the different 'forces that help shape a design." [76, p. 356]. Shaw [**??**] writes, "[Alexander's patterns] helped shape my views on software architecture."

[3]Bass, Clements, and Kazman write, "There will never be a complete list of patterns: patterns spontaneously emerge in reaction to environmental conditions, and as long as those conditions change, new patterns will emerge." [17 ch. 13].

[4]The communication layers of the Internet Protocol Suite are described in RFC 1122 [42].

[5]The separation of TCP and IP was motivated by applications such as conferencing and debugging. The cross-Internet debugger, XNET, needed access to all available information when dealing with stress or failure in the network; it mattered if packets were dropped or out of order. Dave Clark [49] traces the evolution of the design philosophy of TCP and IP. The original design of TCP was due to Robert Kahn and Vinton Cerf [48]. Dave Clark "joined the project in the mid 1970s and took over architectural responsibility for TCP/IP in 1981." [49].

[6]Johnson and Ritchie [110] describe the Unix portability project in 1977. At the time, "Transportation of an operating system and its software between non-trivially different machines [was] rare, but not unprecedented." The three main goals were: write a portable C compiler; refine the C language; and port Unix by rewriting it in C,

[7]Trygve Reenskaug's original May 1979 proposal for the Smalltalk user interface is entitled Thing-Model-View-Editor [168]. *Thing* referred to the application, "something that is of interest to the user," *model* to the objects that represent the thing, *view* to a "pictorial representation" of the model, and *editor* to "an interface between the user and one or more views." "After long discussions," the editor was renamed *controller*.

[8]See Fowler [74] for the evolution and variants of model-view-controller architectures.

[9]Fowler [74] notes that the VisualWorks variant of Smalltalk put presentation logic into a component called Application Model. Following Fowler, the component is called Presentation Model in Fig. 8.10.

[10]Doug McIlroy [140] envisioned a catalog of standard components that could be coupled "like garden hose—screw in another segment when it becomes necessary to massage data in another way."

[11]The design of a compiler was a motivating application for coroutines, a form of dataflow pipeline. Conway [58] laid out a compiler so that "all information items flowing between modules have a component of motion to the right."

[12]Example 8.13 is motivated by the Spotify music streaming service [136].

[13]For named pipes on Windows, see
https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574%28v=vs.85%29.aspx .
Accessed July 13, 2015.

[14]Akidau et al. [5] explore issues related to unbounded streams and describe the processing model of Google Cloud Dataflow.

[15]The following is a brief description of the *map* and *reduce* functions in functional programming. Given a unary function f and a list x, $map(f, x)$ constructs a new list y, where the elements of y are obtained by applying f to the corresponding elements of x. One version of *reduce* accumulates a value. Given a binary function f, a value v, and a list x, $reduce(f, v, x)$ is simply v if the list x is empty. Otherwise, apply the function f to the first element of lists x and the result of applying *reduce* to f, v, and the rest of x, without the first element.

[16]Dean and Ghemawat [64],

[17]Need reference

[18]The definition of program family is due to Parnas [157]. David Parnas credits the idea of program family to Dijkstra [65], who referred in passing to "a program as part of a family or 'in many (potential) versions.'"

[19]See `http://splc.net/hall-of-fame/` for the Software Product Line Conferences Hall of Fame.

[20]The HomeAway example is based on Kreuger, Churchett, and Buhrdorf [121].

[21]Northrop [155].

[22]See Weiss and Lai [204] for the economics of product-line engineering at Bell Labs. David Weiss led a group in Bell Labs Research that worked closely with a small dedicated group in the business unit to support product-line engineering across the parent company, Lucent Technologies.

[23]Plato's theory of Forms or theory of Ideas appears in many of the dialogues, including *The Republic*. The excerpt about beds and tables is from Book X.

Notes for Chapter 9

[1]Humphrey [96, p. 123].

[2]According to Capers Jones [112], projects that use a combination of architecture and code reviews, static analysis, and testing have a defect-removal efficiency of 95-99%, with a median of 97%. The individual techniques have defect-removal efficiencies under 75%. See Section 11.4 for the definition of defect-removal efficiency.

[3]IEEE Standard 1028-2008 defines five kinds of reviews: management reviews, technical reviews, inspections, walk-throughs, and audits [100]. A review is for meeting for examining artifacts for "comment or approval." An inspection is a formal review for finding "anomalies, including errors and deviations from standards and specifications."

[4]Maranzano et al. [135] conducted over 700 architecture reviews between 1988 and 2005. Of the issues uncovered during the reviews, 29%-49% of the design issues could be categorized under "The proposed solution doesn't adequately solve the problem," and 10%-18% under "The problem isn't completely or clearly defined."

[5]The guiding principles for architecture reviews are adapted from Maranzano et al. [135] They "estimate that projects of 100,000 non-commentary source lines of code have saved an average of US$1 million each by identifying and resolving problems early". This estimate is based on reviews at companies that share a Bell Labs heritage.

[6]John Palframan, personal communication, September 2014.

[7]See Fagan [71] and his subsequent paper on software inspections [72]. See also the surveys [124, 13].

[8]Holzmann [93] describes the flight-control software for a mission to land a rover on Mars.

[9]Fagan's 1976 paper [71] had five phases: overview, preparation, inspection, rework, and follow-up. Based on experience with "hundreds of inspections involving thousands of programmers," his 1986 paper [72] added an initial planning phase and noted that "Omitting or combining [phases] led to degraded inspection efficiency that outweighed the apparent short-term benefits. Overview is the only [phase] that under certain conditions can be omitted with slight risk."

[10]Porter, Siy, Toman, and Votta found that there "was no difference between two- and four-person inspections, but both performed better than one-person inspections." [162, p. 338]

[11]Eick et al. [69, p. 64] found that 90% of defects were found during individual preparation. This data was collected as part of a study to estimate residual faults; that is, faults that remain in a completed system.

[12]Votta [201] suggests two alternatives to group meetings: (a) "collect faults by deposition (small face-to-face meetings of two or three persons), or (b) collect faults using verbal or written media (telephone, electronic mail, or notes)."

[13]Porter and Votta [163] report on a study that found reviewers who used checklists were no more effective at finding defects than reviewers who used ad hoc techniques. Reviewers who used scenarios were more effective at finding defects.

[14]In a mid-1990s study of code inspections, Siy and Votta [181] found that 60% of all issues related to readability and maintainability, not to behavior or failure.

[15]The checklist questions in Section 9.4.1 are from a seminal book on testing by Myers [149, p. 22-32].

[16]Rigby et al. [170] review the "policies of 25 [open-source] projects and study the archival records of six large, mature, successful [open-source] projects". The six are Apache httpd server, Subversion, Linux, FreeBSD, KDE, and Gnome.

[17]Mockus, Fielding, and Herbsleb found that 458 people contributed to the Apache server code and documentation [144, p. 311]; 486 people contributed code and 412 people contributed fixes to Mozilla [144, p. 333].

[18]Rigby et al. [170, p. 35:11-35:13] counted a median of two reviewers for Review-then-Commit and one reviewer for Commit-then-Review.

[19]Henderson [90].

[20]For the SSL bug in iOS and Mac OS, see Bland [27].

[21]David Hovemeyer "developed FindBugs as part of his PhD research "in conjunction with his thesis advisor William Pugh." [14] Example 9.3 is based on [95].

[22]Bessey et al. [25] describe the challenges in commercializing the static analyzer, Coverity.

Notes for Chapter 10

[1]Capers Jones [111, slide 25] observes that programmaers are less than 50% efficient at at finding bugs in their own software; normal testing is often less than 75% efficient. See also [112].

[2]In his 2002 interview [80], Bill Gates noted, "When we do a new release of [Microsoft] Windows, which is, say, a billion-dollar effort, over half that is going into the quality." The classic 1979 book on software testing by Glenford J. Myers [149, p. vii] begins with, "It has been known for some time that, in a typical programming project, approximately 50% of the elapsed time and over 50% of the cost are expended in testing the program or system being developed."

[3]The defective code in Fig. 10.1 is from the clock driver for the Freescale MC 13783 processor used by the Microsoft Zune 30 and Toshiba Gigabeat S

media players [211]. The root cause of the failure on December 31, 2008 was isolated by "itsnotabigtruck" [101].

[4]For more on the issues that arise during testing, see the "practice tutorial" by Whittaker [206].

[5]This version of Edsger W. Dijkstra's famous quote about testing is from the 1969 NATO Software Engineering Techniques conference [46, p. 16].

[6]Capers Jones provides empirical data on the "defect removal efficiency" of various combinations of reviews, static, analysis, and testing. Hackbarth, Mockus, Palframan, and Sethi [86] describe a software-quality improvement program based on reviews, static analysis, and testing that led to improvements in post-delivery operational experience.

[7]Weyuker [205].

[8]Primality testing is the problem of deciding whether a number n is a prime number. In 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena showed that there is a deterministic polynomial algorithm for primality testing [4].

[9]SWEBOK 3.0 merges system and functional testing into a single level [40, p. 4-5.]. The levels of testing in Section 10.2 assign the validation and verification roles to system and functional testing, respectively. The classic text on testing by Glenford J. Myers separates system and functional testing [149].

[10]Capers Jones has published summary data from 600 client companies [112]: "Many test stages such as unit test, function test, regression test, etc. are only about 35% efficient in finding code bugs, or find one bug out of three. This explains why 6 to 10 separate kinds of testing are needed."

[11]The xUnit family began with Kent Beck's automated testing frameworks for Smalltalk. In 1997, Smalltalk usage was on the decline and Java usage was on the rise, so Kent Beck and Erich Gamma created JUnit for Java. They had three goals for JUnit: make it natural enough that developers would actually use it; enable tests that retain their value over time; and leverage existing tests in creating new ones [21].

[12]David L. Parnas [158] introduced the "uses" relation as an aid for designing systems "so that subsets and extensions are more easily obtained." Incremental bottom-up integration corresponds to building a system by extension.

[13]Myers [149, p. 99-100] notes that a comparison between top-down and bottom-up integration testing "seems to give the bottom-up strategy the edge."

[14]Zhu, Hall, and May survey test coverage and adequacy criteria [216].

[15]Chilensky and Miller 1994

[16]MC/DC is included in RTCA document DO-178C [208]. FAA Advisory Circular 20-115C, dated July 19, 2013, recognizes DO-178C as an "acceptable means, but not the only means, for showing compliance with the applicable airworthiness regulations for the software aspects of airborne systems."

[17]While discussing industry trends in 2017, Ebert and Shankar [68] recommend MC/DC for detecting security backdoors.

[18]The leap-year specification is from the US Naval Observatory]196].

[19]Myers [149, p. 46-47] provides heuristic guidelines for equivalence partitioning.

[20]hagar-2015-testing
[21]Kuhn 2014 keynote
[22]Cohen, Dalal, Fredman, Patton [54].
[23]Kuhn keynote. It's also in the Hagar paper.
[24]The development process in in Fig. 10.15 is from Benington's description of the SAGE air defense system [23].

Notes for Chapter 11

[1]Lord Kelvin began a lecture on May 3, 1883 with, "In physical science a first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. ! often say ..." [**??**, p. 79-80].
[2]SWEBOK
[3]Function points were proposed as a measure for program size, but they were later found to correlate with lines of code. According to Capers Jones, a function point corresponds to roughly 50 lines of Java code. [**??**].
[4]Example 11.3 is based on an account by Clint Covington, Principal Program Manager Lead, Microsoft Office [63].
[5]Buckley and Chillarege [47].
[6]Garvin [78] synthesized the various definitions of quality for physical products into five approaches: "(1) the transcendental approach of philosophy; (2) the product-based approach of economics; (3) the user-based approach of economics, marketing, and operations management; and (4) the manufacturing based and (5) value-based approach of operations management." Kitchenham and Pfleeger [119] applied Garvin's model to software. The model in Fig. 11.4 uses "aesthetic" instead of "transcendental" and splits the user-based approach into two: functional and operations.
[7]Shewhart [179, p. 38] defines product quality in terms of properties that cannot be altered without altering the product.
[8]Example 11.5 is based on email from Gilman Stevens to Audris Mockus, May 14, 2014.
[9]Stanley Smith Stevens proposed four levels of scales in 1946. The system has been challenged by theoreticians, especially the Nominal and Ordinal levels, the Stevens system continues to be widely used.
[10]CQM was introduced by Mockus and Weiss [147] under the name Interval Quality. Hackbarth et al. [86] describe a company-wide quality improvement program based on CQM.

Notes for Chapter 12

[1]Huckleberry's comment is close to the end of Mark Twain's 1885 book, *Adventures of Huckleberry Finn.*

Bibliography

1. ACM/IEEE-CS. *Software Engineering Code of Ethics and Professional Practice* (1999). https://ethics.acm.org/code-of-ethics/software-engineering-code/.

2. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press (December 2013) http://dx.doi.org/10.1145/2534860.

3. Agile Alliance. Agile 101. (Retrieved January 21, 2019) https://www.agilealliance.org/agile101/.

4. Agrawal, Manindra, Neeraj Kayal, and Nitin Saxena. PRIMES is in *P*. *Annals of Mathematics*, Second Series 160, 2 (September 2004) 781-793.

5. Akidau, Tyler, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernáncez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015) 1792-1803.

6. Alexander, Christopher, Sara Ishikawa, Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York (1977).

7. Ambler, Scott W. UML 2.5: Do you even care? *Dr. Dobb's* (November 19, 2013). URL:http://www.drdobbs.com/architecture-and-design/uml-25-do-you-even-care/240163702.

8. Android Open Source Project. Android interfaces and architecture. https://source.android.com/devices/.

9. Apple Inc. iOS Technology Overview (September 17, 2014) https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iOSTechOverview.pdf.

10. Aranda, Jorge and Steve Easterbrook. Anchoring and adjustment in software estimation. *Proceedings 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2005) 346-355.

11. Ardis, Mark A., and Janel A. Green. Successful introduction of domain engineering into software development. *Bell Labs Technical Journal* 3, 3 (July-September 1998) 10-20.

12. Aristotle. *Politics*. See `http://classics.mit.edu/Aristotle/politics.html` for the Benjamin Jowett translation.

13. Aurum, Aybuke, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability* 12 (2002) 133-154.

14. Ayewah, Nathaniel, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software* 25, 5 (September-October 2008) 22-29.

15. Basili, Victor R,, Mikael Lindvall, Myrna Regardie, Carolyn Seaman, Jens Heidrich, Jürgen Münch, Dieter Rombach, and Adam Trendowicz. Linking software development and business strategy through measurement. *IEEE Computer* 43, 4 (April 2010) 57-65.

16. Basili, Victor R,, and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* SE-10, 6 (November 1984) 728-738.

17. Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice* (3rd ed.). Addison-Wesley (2013).

18. Bauer, Friedrich L. Software engineering. In *Information Processing 71.* North-Holland (1972) 530-538.

19. Beck, Kent. Embracing change with Extreme Programming. *IEEE Computer* (October 1999) 70-77

20. Beck, Kent, with Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Ed.* Addison-Wesley, Reading, Mass. (2005).

21. Beck, Kent, and Erich Gamma. JUnit: a cook's tour. (circa 1998). `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`.

22. Begel, Andrew, and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. *36th International Conference on Software Engineering (ICSE)* (2014) 12-23.

23. Benington, Herbert D. Production of large computer programs. *Proceedings, Symposium on Advanced Programming Methods for Digital Computers*, Office of Naval Research Symposium (June 1956). Reprinted with a Foreword by the author in *Annals of the History of Computing* 5, 4 (October 1983) 350-361.

24. Bentley, Jon. Programming Pearls: bumper-sticker computer science. *Comm. ACM* 28, 9 (September 1985) 896-901.

25. Bessey, Al, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines fo code later: using static analysis to find bugs in the real world. *Comm. ACM* 53, 2 (February 2010) 66-75.

26. Bittner, Kurt, and Ian Spence. *Use Case Modeling* Addison-Wesley Professional (2003).

27. Bland, Mike. Finding more than one worm in the apple. *Comm. ACM* 57, 7 (July 2014) 58-64.

28. Boehm, Barry W. Software engineering. *IEEE Transactions on Computers* C-25, 12 (December 1976) 1226-1241.

29. Boehm, Barry W. Verifying and validating software requirements and design specifications. *IEEE Software* (January 1984) 75-88. A 1979 version is available as technical report USC-79-501 `http://csse.usc.edu/TECHRPTS/1979/usccse79-501/usccse79-501.pdf`.

30. Boehm, Barry W. *Software Engineering Economics* (1981).

31. Boehm, Barry W. A spiral model of software development and enhancement. *Computer* 21, 5 (May 1988) 61-72,

32. Boehm, Barry W. Anchoring the software process. *IEEE Software* (July 1996) 73-82.

33. Boehm, Barry W. *Spiral Development: Experience, Principles, and Refinements*. Software Engineering Institute Report CMU/SEI-2000-SR-008 (July 2000). `http://www.sei.cmu.edu/reports/00sr008.pdf`

34. Boehm, Barry W. A view of 20th and 21st century software engineering. *Proceedings International Conference on Software Engineering (ICSE '06)* (2006) 12-29.

35. Boehm, Barry W. The Incremental Commitment Spiral Model (ICSM): principles and practices for succesful software systems. *ACM Webinar* (December 17, 2013).

36. Boehm, Barry W.; Bradford Clark, Ellis Horowitz; Chris Westland; Ray Madachy; and Richard Selby. Cost models for future life cycle processes: COCOMO 2.0, *Annals of Software Engineering* 1, 1 (1995) 57-94.

37. Boehm, Barry W., and Ricardo Valerdi. Achievements and challenges in Cocomo-based software resource estimation. *IEEE Software* (September-October 2008) 74-83.

38. Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide, 2nd Ed.* Addison-Wesley (2005).

39. Booch, Grady. The History of Software Engineering. *ACM SIGSOFT Webinar* (2018).

40. Bourque, Pierre; and Richard E. Fairley (eds) *Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 3.0.* IEEE Computer Society (2014)
`http://www.swebok.org`.

41. Bower, Andy, and Blair McGlashan. Twisting the triad. *European Smalltalk User Group (ESUG)* (2000).

42. Braden, R. (ed). *Requirements for Internet Hosts: Communication Layers* Internet Engineering Task Force RFC-1122 (October 1989).

43. Britton. Kathryn H., and David L. Parnas. *A-7E Software Module Guide.* Naval Research Laboratory Memorandum 4702 (December 1981).

44. Brooks, Frederick P., Jr. *The Mythical Man Month: Essays on Software Engineering.* (1975) Addison-Wesley, Reading, Mass. See also the *Anniversary Edition* (1995) with four added chapters.

45. Brooks, Frederick P., Jr. No silver bullet—essence and accident in software engineering. *Proceedings of the IFIP Tenth World Computing Conference.* Elsevier, Amsterdam (1986) 1069-1076. Reprinted in *IEEE Computer* (April 1987) 10-19.

46. Buxton, John N., and Brian Randell (eds), *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, Rome, Italy, October 1969. (published April 1970).
`http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`

47. Buckley, Michael, and Ram Chillarege. Discovering relationships between service and customer satisfaction. *Proceedings IEEE Intl. Conf. Software Maintenance (ICSM '95)* (1995) 192-201.

48. Cerf, Vinton G., and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications* COM-22, 5 (May 1974) 637-648.

49. Clark, David D. The design philosophy of the DARPA Internet Protocols. *Computer Communications Review* 18,4 (August 1988) 106-114.

50. Clegg, Dai; Richard Barker; and Barbara Barker. *Case Method Fast Track: A RAD Approach.* (1994) Oracle Corporation.

51. Cockburn, Alistair. *Writing Effective Use Cases*; Addison-Wesley (2001).

52. Cockburn, Alistair. Why I still use use cases. (January 9, 2008). `http://alistair.cockburn.us/Why+I+still+use+use+cases`.

53. Cockburn, Alistair, and Laurie Williams. The costs and benefits of pair programming. `http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF`.

54. Cohen, David M., Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: an approach to testing based on combinatorial designs. *IEEE Transactions on Software Engineering* 23, 7 (July 1997) 437-444.

55. Cohn, Mike. *Agile Estimation and Planning.* Prentice–Hall (2005),

56. CollabNet VersionOne. *14th Annual State of Agile Report.* (2019) `http://stateofagile.versionone.com/`.

57. Constantine, Larry L. Essential modeling: use cases for modeling user interfaces. *ACM Interactions* 2, 2 (April 1995) 34-46.

58. Conway, Melvin E. Design of a separable transition-diagram compiler. *Comm. ACM* 6, 7 (July 1963) 396-408.

59. Conway, Melvin E. How do committees invent? *Datamation* (April 1968) 28-31.

60. Cusumano, Michael A., and David B. Yoffie. *Competing on Internet Time.* The Free Press, New York (1998).

61. Dalkey, Norman, and Olaf Helmer. An experimental application of the Delphi method to the use of experts. *Management Science* 9, 3 (April 1963) 458-467.

62. Dardenne, Anne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming* 20, 1-2 (April 1993) 3-50.

63. De Bonte, Austina; and Drew Fletcher. *Scenario-Focused Engineering.* Microsoft Press, Redmond, Wash. (2013).

64. Dean, Jeffrey, and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *IComm. ACM* 51, 1 (January 2008) 107-113.

65. Dijkstra, Edsger W. Structured programming. In *Software Engineering Techniques*, J. N. Buxton and B. Randell (eds.) NATO Science Committee (April 1970) 84-88. Report on the NATO Software Engineering Conference in Rome (October 1969).

66. Doran, George T. There's a S.M.A.R.T. way to write management's goals and objectives, *Management Review* 70, 11 (1981) 35-36.

67. Drucker, Peter F. *The Essential Drucker*. Harper Business, New York (2001).

68. Ebert, Christof, and Kris Shankar. Industry trends 2017. *IEEE Software* (March-April 2017) 112-116.

69. Eick, Stephen G., Clive R. Loader, M. David Long, Lawrence G. Votta, and Scott Vander Wiel. Estimating software fault content before coding. *14th International Conference on Software Engineering (ICSE)* (May 1992) 59-65.

70. Erickson, John. A decade or more of UML: an overview of UML semantic and structural issues and UML field use. *Journal of Database Management* 19, 3 (2008) i-vii.

71. Fagan, Michael E. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1876) 258-287.

72. Fagan, Michael E. Advances in software inspections. *IEEE Transactions on Software Engineering* SE12, 7 (July 1986) 744-751

73. Ford, Gary. *1990 SEI Report on Undergraduate Software Engineering Education*. Software Engineering Institute Technical Report CMU/SEI-90-TR-003 (March 1990).

74. Fowler. Martin. GUI architectures (July 18 2006). `http://martinfowler.com/eaaDev/uiArchs.html`.

75. Fowler. Martin. Yagni. (May 26, 2015) `https://martinfowler.com/bliki/Yagni.html`.

76. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, Mass. (1995).

77. Fowler. Martin. The State of Agile Software in 2018: Transcript of a Keynote at Agile Australia. (August 25 2018). `https://martinfowler.com/articles/agile-aus-2018.html`.

78. Garvin, David A. What does "product quality" really mean? *Sloan Management Review* 26, 1 (Fall 1984) 25-43.

79. Gates, William H., III. The Internet Tidal Wave. Internal Microsoft memo. (May 26, 1995). http://www.justice.gov/atr/cases/exhibits/20.pdf .

80. Gates, William H., III. Q&A: Trustworthy Computing. *Information Week* (May 16, 2002).
http://www.informationweek.com/qanda-bill-gates-on-trustworthy-computing/d/d-id/1015083

81. Geppert, Birgit, and Frank Roessler. *Multi-Conferencing Capability* United States Patent 8,204,195 (June 19, 2012).

82. Glick, Bryan. The BBC DMI project: what went wrong? *Computer Weekly* (Feb 5, 2014)
http://www.computerweekly.com/news/2240213773/The-BBC-DMI-project-what-went-wrong .

83. Gomez-Uribe, Carlos A., and Neil Hunt The Netflix recommender system: algorithms, business value, and innovation. *ACM Transactions on Management Information Systems.* 6, 4 (Dec 2015) Article 13.

84. Greer, Derek. Interactive application and architecture patterns. (August 25 2007).
http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/ .

85. Griffin, Abbie. and John R. Hauser. The Voice of the Customer. *Marketing Science* 12, 1 (Winter 1993) 1-27.

86. Hackbarth, Randy, Audris Mockus, John D. Palframan, and Ravi Sethi. Improving software quality as customers perceive it. *IEEE Software* (July/August 2016) 40-45.
https://www.computer.org/cms/Computer.org/ComputingNow/issues/2016/08/mso2016040040.pdf

87. Hagar, Jon D., Thomas L. Wissink, D. Richard Kuhn, and Raghu N. Kacker. Introducing combinatorial testing in a large organization. *IEEE Computer* (April 2015) 64-72.

88. Hamilton, Margaret. The Engineer Who Took the Apollo to the Moon. Interviewed by Verne for a Spanish newspaper (Dec 25k 2014).
https://medium.com/@verne/margaret-hamilton-the-engineer-who-took-the-apollo-to-the-moo

89. Helmer, Olaf. *Analysis of the Future: The Delphi Method.* RAND Corporation, Report P-3558 (March 1967).

90. Henderson, Fergus. Software engineering at Google. (January 31, 2017).
https://arxiv.org/ftp/arxiv/papers/1702/1702.01715.pdf .

91. Herzberg, Frederick. *Work and the Nature of Man.* Cleveland World Publishing Co. (1966).

92. Holzmann, Gerard J. Landing a spacecraft on Mars. *IEEE Software* (March-April 2013) 17-20.

93. Holzmann, Gerard J. Mars code. *Communications of the ACM* 57, 2 (February 2014) 64-73.

94. Hormby, Tom. The story behind the Walkman. *Low End Mac* (August 13, 2013).
http://lowendmac.com/2013/the-story-behind-the-sony-walkman/ .

95. Hovemeyer, David, and William Pugh. Finding more null pointer bugs, but not too many. *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* ACM, New York (June 2007) 9-14.

96. Humphrey, Watts S. *The Watts New? Collection: Columns by SEI's Watts Humphrey.* Software Engineering Institute CMU/SEI-2009-SR-024 (November 2009).
http://resources.sei.cmu.edu/asset_files/SpecialReport/2009_003_001_15035.pdf .

97. Iansiti, Marco, and Alan D. MacCormack. Developing products on Internet time. *Harvard Business Review* (September-October 1997).

98. *IEEE Recommended Practice for Software Requirements Specifications.* IEEE Standard 830-1998 (Reaffirmed December 9, 2009).

99. IEEE Computing Society and ACM. *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.*
http://www.acm.org/education/se2014.pdf .

100. IEEE Standard 1028-2008. *IEEE Standard for Software Reviews and Audits* (August 2008).

101. itsnotabigtruck. Cause of Zune 30 leapyear problem ISOLATED!
http://www.zuneboards.com/forums/showthread.php?t=38143

102. Jacobson, Ivar. Object oriented development in an industrial environment. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (October 1987) 183-191.

103. Jacobson, Ivar. Use cases: yesterday, today, and tomorrow. *Software and Systems Modeling* (2004) 210-220.

104. Jacobson, Ivar. Ivar Jacobson on UML, MDA, and the future of methodologies. InfoQ interview. (October 24, 2006).
https://www.infoq.com/interviews/Ivar_Jacobson/ .

105. Jacobson, Ivar; Ian Spence; and Brian Kerr. Use-Case 2.0: The hub of software development. *ACM Queue* 14, 1 (January-February 2016) 94-123.

106. Jalote, Pankaj. *Software Project Management in Practice*. Addison-Wesley, Boston, Mass. (2002).

107. Jeffries, Ron. Essential XP: Card, Conversation, Confirmation. (August 30, 2001) `http://ronjeffries.com/xprog/articles/expcardconversationconfirmation/`.

108. Johnson, Philip M., and Danu Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering* 3, 1 (1998) 9-35.

109. Johnson, Stephen C. *Lint, a C Program Checker*. Computing Science Technical Report 65, Bell Laboratories (July 26, 1978). `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841`.

110. Johnson, Stephen C., and Dennis M. Ritchie. Portability of C programs and the UNIX system. *Bell System Technical Journal* 57, 6 (July-August 1978) 2021-2048.

111. Jones, Capers. Software Quality in 2013: A Survey of the State of the Art. *35th Annual Pacific NW Software Quality Conference* (October 2013). The following website has a link to his video keynote: `https://www.pnsqc.org/software-quality-in-2013-survey-of-the-state-of-the-art/`.

112. Jones, Capers. Achieving software excellence. *Crosstalk* (July-August 2014) 19-25.

113. Jørgensen, Magne. What we do and don't know about software development effort estimation. *IEEE Software* (March-April 2014) 13-16.

114. Jørgensen, Magne, and Barry Boehm. Software development effort estimation: formal models or expert judgment? *IEEE Software* (March-April 2009) 14-19.

115. Kano, Noriaki. Life cycle and creation of attractive quality. `http://huc.edu/ckimages/files/KanoLifeCycleandAQCandfigures.pdf`. According to Löfgren and Wittel [132] a paper with this title was presented at the *4th International QMOD Quality Management and Organizational Development Conference* at Linköping University (2001).

116. Kano, Noriaki, Nobuhiko Seraku, Fumio Takahashi, and Shin-ichi Tsuji. Attractive quality and must-be quality (in Japanese). *Journal of the Japanese Society for Quality Control* 14, 2 (1984) 147-156.

117. Kazman, Rick, Paul Clements, Len Bass, and Gregory Abowd. Classifying architectural elements as a foundation for mechanism matching. *Computer Software and Applications Conference (COMPSAC '97)* (August 1997) 14-17.

118. Kazman, Rick, and Amnon Eden. Defining the terms architecture, design, and implementation, *news@sei* 6, 1 (First Quarter 2003).

119. Kitchenham, Barbara, and Shari Lawrence Pfleeger. Software quality: the elusive target. *IEEE Software* (January 1996) 12-21.

120. Klein, John, and David Weiss, What is architecture? (2009) [184, p. 3-24].

121. Kreuger, Charles W., Dale Churchett, and Ross Buhrdorf. HomeAway's transition to software product line practice: engineering and business results in 60 days. *12th International Software Product Line Conference (SPLC '08)*. IEEE (September 2008) 297-306.

122. Kruchten, Philippe B. The 4+1 view model of architecture. *IEEE Software* (November 1995) 42-50.

123. Kuhn, D. Richard. Combinatorial testing: rationale and impact. Keynote at *IEEE Seventh International Conference on Software Testing, Verification, and Validation* (April 2, 2014) Presentation available at `http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-icst-14.pdf`

124. Laitenberger, Olivier, and Jean-Marc DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software* 50, 1 (2000) 5-31.

125. Lamsweerde, Axel van Goal-Oriented Requirements Engineering: A Guided Tour *IEEE* (2001)

126. Larman, Craig, and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer* 36, 6 (June 2003) 47-56.

127. LaToza, Thomas, Gina Venolia, and Rob DeLine. Maintaining mental models: a study of developer work habits. *28th International Conference on Software Engineering (ICSE '06)*. ACM, New York (2006) 492-501.

128. Ledgard, Henry F. Ten mini-languages: a study of topical issues in programming languages. *ACM Computing Surveys* 3, 3 (Sep 1971) 115-146.

129. Leslie, Ian. The scientists who make apps addictive. *1843 Magazine* (Oct-Nov 2016).
`https://www.1843magazine.com/features/the-scientists-who-make-apps-addictive`

130. Leveson, Nancy G. Medical Devices: The Therac-25. Appendix A of *Software: System Safety and Computers* by Nancy Leveson. Addison Wesley, Reading, Mass. (1995)
`http://sunnyday.mit.edu/papers/therac.pdf` .

131. Leveson, Nancy G., and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer* 26, 7 (July 1993) 18-41.
`http://courses.cs.vt.edu/professionalism/Therac_25/Therac_1.html` .

132. Löfgren, Martin, and Lars Wittel. Two decades of using Kano's theory of attractive quality: a literature review. *The Quality Management Journal* 15, 1 (2008) 59-75.

133. MacCormack, Alan D. Product-development processes that work: How Internet companies build software. *Sloan Management Review* 42, 2 (Winter 2001) 75-84.

134. Madden, William A., and Kyle Y. Rone. Design, development, integration: Space Shuttle primary flight software system. *Comm. ACM* 27, 9 (1984) 914-925.

135. Maranzano, Joseph F., Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, and David M. Weiss. Architecture Reviews: Practice and Experience. *IEEE Software* (March-April 2005) 34-43.

136. Maravić, Igor. Spotify's event delivery: the road to the cloud (Part I). (February 25, 2016). https://labs.spotify.com/2016/02/25/spotifys-event-delivery-the-road-to-the-cloud-part-i/ .

137. Mavin, Alistair, and Phillip Wilkinson. Big EARS: The return of "Easy Approach to Requirements Syntax." *2010 18th IEEE International Requirements Engineering Conference*. (2010). 277-282.

138. McConnellm Steve. Origins of 10x: how valid is the underlying research. (January 9, 2011). `http://www.construx.com/10x_Software_Development/Origins_of_10X_%E2%80%93_How_Valid_is_the_Underlying_Research_/`

139. McConnellm Steve. Measuring software productivity. ACM Learning Webinar (January 11, 2016). `http://resources.construx.com/wp-content/uploads/2016/02/Measuring-Software-Development-Productivity.pdf` .

140. McIlroy, M. Douglas. Typescript (October 11, 1964). http://doc.cat-v.org/unix/pipes/ .

141. McIlroy, M. Douglas. Coroutine prime number sieve. (May 6, 2015). http://www.cs.dartmouth.edu/ doug/sieve/ .

142. McIlroy, M. Douglas, Elliot N. Pinson, and Berkley A. Tague. Foreword: Unix time-sharing system. *Bell System Technical Journal* 57, 6 (July-August 1978) 1899-1904.

143. Miller, George A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review* 101, 2 (1955) 343-352.

144. Mockus, Audris, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11, 3 (July 2002) 309-346.

145. Mockus, Audris, Randy Hackbarth, and John D. Palframan. Risky files: an approach to focus quality improvement effort. *European Conference on Software Engineering and ACM SIGSOFT Symposium on Foundations of Software Engineering (ECSE/FSE ?13)*. ACM Press, New York, NY (2013) 691-694.

146. Mockus, Audris, and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. *International Conference on Software Engineering (ICSE)*. (2002) 503-512.

147. Mockus, Audris, and David M. Weiss. Interval quality: relating customer-perceived quality to process quality. *International Conference on Software Engineering (ICSE ?08)*. ACM Press, New York, NY (2008) 723?732.

148. Moder, Joseph J., Cecil R. Phillips, and Edward D. Davis. Project Management with CPM, PERT, and Precedence Programming, 3rd ed. Van Nostrand Reinhold, New York (1983).

149. MyersGlenford J. *The Art of Software Testing*. John Wiley (1979).

150. National Audit Office. *British Broadcasting Corporation: Digital Media Initiative* Memorandum (Jan 27, 2014)
`https://www.nao.org.uk/wp-content/uploads/2015/01/BBC-Digital-Media-Initiative.pdf`.

151. Naur, Peter, and Brian Randell (eds). *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, 7th to 11th October 1968. (Jan 1969).
`http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`.

152. Neumann,Peter G. Cause of AT&T network failure. *Risks Digest* 9, 62 (January 26, 1990). `http://catless.ncl.ac.uk/Risks/9/62#subj2`.

153. Nichols,W. T. Capital cost estimating. *Industrial and Engineering Chemistry* 43, 10 (1951) 2295-2298.

154. North, Dan. Behavior modification. *Better Software Magazine* (March 2006).
See `https://dannorth.net/introducing-bdd/`.

155. Northrop, Linda M. SEI's software product line tenets. *IEEE Software* (July-August 2002) 32-40.

156. Parnas, David L. On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15, 12 (December 1972) 1053-1058.

157. Parnas, David L. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (March 1976) 1-9.

158. Parnas, David L. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5, 2 (March 1979) 128-138.

159. Parnas, David L. Software engineering: multi-person development of multi-version programs. In *Dependable and Historic Computing*, C. B. Jones and J. L. Lloyd (eds.), *Lecture Notes in Computer Science* 6875 (2011) 413-427.

160. Parnas, David L., Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Trans. Software Engineering* SE-11, 3 (March 1985) 259-266.

161. Petre, Marian. UML in practice. *International Conference on Software Engineering (ICSE '13)*. (2013) 722-731.

162. Porter, Adam, Harvey P. Siy, Carol A. Toman, and Lawrence G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering* 23, 6 (June 1997) 329-346.

163. Porter, Adam, and Lawrence G. Votta, Jr. What makes inspections work? *IEEE Software* 14, 6 (November-December 1997)

164. Potel, Mike. MVP: Model-View-Presenter (1996). `http://www.wildcrest.com/Potel/Portfolio/mvp.pdf`.

165. Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOKGuide)*. Project Management Institute (2008)

166. Quote Investigator. t's Difficult to Make Predictions, Especially About the Future. (October 20, 2013). `https://quoteinvestigator.com/2013/10/20/no-predict/`.

167. Rabinowitz, Dorelle. `http://www.aiga.org/inhouse-initiative/intuit-quickbooks-ipad-case-study-app-design/`.

168. Reenskaug, Trygve. The original MVC reports (February 12, 2007). `http://folk.uio.no/trygver/2007/MVC_Originals.pdf`.

169. Richards, Carl (interviewed by Charles Rotblut). Creating and following a financial plan. *AAII Journal* 37, 8 (August 2015) 29-32.

170. Rigby, Peter C., Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer review on open- source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology* 23, 4 (August 2014) 35:1-35:33.

171. Rogers, Carl R., and F. J. Roethlisberger. Barriers and gateways to communication. *Harvard Business Review* (July-August 1952). Reprinted, *Harvard Business Review* (November-December 1991). https://hbr.org/1991/11/barriers-and-gateways-to-communication

172. Royce, Winston W. Managing the development of large software systems. *Proceedings IEEE WESCON* (August 1970).

173. Ruberto, John. Design for Delight applied to software process improvement. *Pacific Northwest Software Quality Conference* (October 2011). http://www.pnsqc.org/design-for-delight-applied-to-software-process-improvement/.

174. Sackman, H., W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM* 11, 1 (January 1968) 3-11.

175. Sanders, Elizabeth B.-N. Converging perspectives: product development research for the 1990s. *Design Management Journal* 3, 4 (Fall 1992) 49-54.

176. Sanders, Elizabeth B.-N. From user-centered to participatory design approaches. In *Design and the Social Sciences*, J. Frascara (ed.) Taylor & Francis Books (2002).

177. Savor, Terry; Mitchell Douglas; Michael Gentili; Laurie Williams; Kent Beck; and Michael Stumm. Continuous deployment at Facebook and OANDA. *IEEE/ACM International Conference on Software Engineering (ICSE)*. (2016) 21-30.

178. Schneier, Bruce. Attack trees. *Dr. Dobbs' Journal* (1999). https://www.schneier.com/academic/archi

179. Shewhart, Walter A. *Economic Control of Quality of Manufactured Product*. D. Van Nostrand, New York (1931). Reprinted by American Society for Quality Control (1980).

180. Shul, Forrest, Victor R, Basili, Barry W, Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What have we learned about fighting defects. *Proceedings Eighth IEEE Symposium on Software Metrics (METRICS '02)* (2002) 249-258.

181. Siy, Harvey P., and Lawrence G. Votta. Does the modern code inspection have value? *IEEE International Conference on Software Maintenance* (2001) 281-289

182. Smith, Greg, and Ahmed Sidky. *Becoming Agile: ... in an imperfect world*, Manning Publications (2009).

183. Software Engineering Institute. *What is Your Definition of Software Architecture?* Carnegie Mellon University, 3854 01.22.17 (2017) https://resources.sei.cmu.edu/asset_files/FactSheet/2010_010_001_513810.pdf.

184. Spinellis, Diomidis, and Georgios Gousios (eds.). *Beautiful Architecture.* O'Reilly Media, Sebastopol, Calif. (2009).

185. Stevens, W. P., G. J. Meyers, and L. L. Constantine. Structured design. *IBM Systems J.* **13**, 2 (June 1974) 115-138.

186. Sutherland, Jeff; and Ken Schwaber. *The Scrum Guide.* (Novermber 2017)
`http://www.scrumguides.org/` .

187. Taylor, Suzanne, and Kathy Schroeder. *Inside Intuit.* Harvard Business School Press, Boston, Mass. (2003).

188. TeXShop.]`https://pages.uoregon.edu/koch/texshop/` .

189. Thomson, William (Lord Kelvin). *Popular Lectures and Addresses, Volume I, 2nd Edition* MacMillan (1891).

190. Tversky, Amos, and Daniel Kahneman. Judgement under uncertainty: heuristics and biases. *Science* 185 (September 27, 1974) 1124-1131.

191. Wikipedia. 5 Whys. `https://en.wikipedia.org/wiki/5_Whys` .

192. US Department of Defense. *Military Standard: Technical Reviews and Audits for Systems, Equipments, and Computer Software* MIL-STD-1521B (June 4, 1985).
`http://www.dtic.mil/dtic/tr/fulltext/u2/a285777.pdf`

193. US House of Representatives. Ways and means of effecting economies in the national space program. Hearing before the Committee on Science and Astronautics, 87th Congress, Second Session, *No.* 17 (July 26, 1962) 75-193.

194. US House of Representatives. PPACA implementation failures: answers from HHS? Hearing before the Committee on Energy and Commerce, 113th Congress, *Serial No.* 113-87 (October 24, 2013).

195. US House of Representatives. PPACA implementation failures: didn't know or didn't disclose? Hearing before the Committee on Energy and Commerce, 113th Congress, *Serial No.* 113-90 (October 30, 2013).

196. US Naval Observatory. Introduction to Calendars.
`http://aa.usno.navy.mil/faq/docs/calendars.php` .

197. USA Today. Microsoft banks on anti-Clippy sentiment. (February 6, 2002).
`http://usatoday30.usatoday.com/tech/news/2001-05-03-clippy-campaign.htm` .

198. van Lamsweerde, Axel. Requirements engineering in the year 00: A research perspective. *International Conference on Software Engineering (ICSE).* (2000) 5-19.

199. van Lamsweerde, Axel, Robert Darimont, and Philippe Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. *Second IEEE International Symposium on Reliability Engineering.* (1995) 194-203

200. Vitruvius Polilo, Marcus. *de Architectura.* (circa 15 BC). See `http://penelope.uchicago.edu/Thayer/E/Roman/Texts/Vitruvius/1*.html` for the Joseph Gwilt English translation (1826), maintained by Bill Thayer.

201. Votta, Lawrence G., Jr. Does every inspection need a meeting? *1st ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '93).* Distributed as *Software Engineering Notes* 18, 5 (December 1993) 107-114.

202. Walston, C. E., and C. P. Felix. A method of programming measurement and estimation. *IBM Systems Journal* 16, 1 (March 1977) 54-73.

203. Weaver, Patrick. The origins of modern project management. Originally presented at the *Fourth Annual PMI College of Scheduling Conference* (2007). `http://www.mosaicprojects.com.au/PDF_Papers/P050_Origins_of_Modern_PM.pdf`

204. Weiss, David M., and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process* Addison-Wesley, Reading Mass. (1999).

205. Weyuker, Elaine J. On testing non-testable programs. *The Computer Journal* 25, 4 (1982) 465-470.

206. Whittaker, James A. What is software testing? And why is it so hard? *IEEE Software* (January-February 2000) 70-79.

207. Wikipedia. A map is not the territory. `https://en.wikipedia.org/wiki/Map%E2%80%93territory_relation`.

208. Wikipedia. Modified condition/decision coverage. `https://en.wikipedia.org/wiki/Modified_condition/decision_coverage`.

209. Wikipedia. Office Assistant. `https://en.wikipedia.org/wiki/Office_Assistant`.

210. Wikipedia. Project Management Triangle. `https://en.wikipedia.org/wiki/Project_management_triangle`

211. Wikipedia. Zune 30. `https://en.wikipedia.org/wiki/Zune_30`.

212. Wirfs-Brock, Rebecca. Designing scenarios: making the case for a use case framework. *The Smalltalk Report* 3, 3 (November-December 1993). `http://wirfs-brock.com/PDFs/Designing Scenarios.pdf`.

213. Wray, Stuart. How pair programming really works. *IEEE Software* (January-February 2010) 51-55. See also: Responses to "How pair programming really works." *IEEE Software* (March-April 2010) 8-9.

214. Zheng, Min, Hui Xue, Yolong Zhang, Tao Wei, and John C. S. Lui. Enpublic apps: security threats using iOS enterprise and developer certificates. *10th ACM Symposium on Information, Computer and Communications Security (Asia CCS '15).* ACM, New York (April 2015) 463-474.

215. Zave, Pamela. Classification of research efforts in requirements engineering. *ACM Computing Surveys* 29, 4 (1997) 315-321.

216. Zhu, Hong, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys* 29, 4 (December 1997) 366-427.