COSC 340: Software Engineering

Design and Architecture

Audris Mockus

(adapted from slides by Ravi Sethi, University of Arizona)

Concepts from Building Architecture

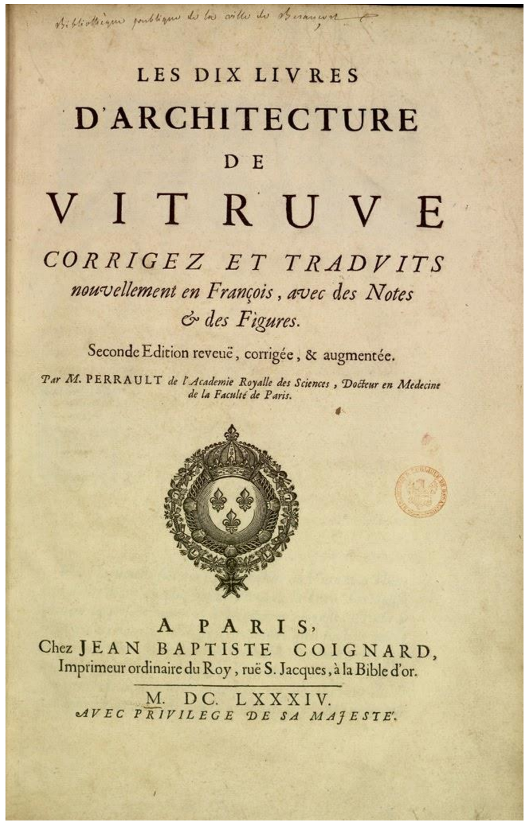
- **Draw on centuries old traditions of building architecture**
- **Similarities between building and software architecture**

– Principles:	attributes of good architecture
– Structure:	components and their relationships
– Views:	tailored to intended purpose and audience
– Patterns:	core of a solution to a recurring problem

- **Differences**
  - Software is both static and dynamic
  - Software is intangible
  - **Some** software is easy to change

De Architectura: Ten Books on Architecture

- **Treatise on architecture by Vitruvius, Roman architect, 1st century BC**
  - Town planning, architecture, civil engineering
  - Building materials
  - Temples
  - Civil buildings
  - Domestic buildings
  - Pavements and decorative plasterwork
  - Water supplies and aqueducts
  - Sciences: geometry, measurement, astronomy
  - Machines: water mills, drainage, hoisting, pneumatics



Bibliotheque publique de la ville de Besançon

LES DIX LIVRES
D'ARCHITECTURE
DE
VITRUVE
CORRIGEZ ET TRADVITS nouvellement en François, avec des Notes & des Figures.

Seconde Edition reveuë, corrigée, & augmentée.

Par M. PERRAULT *de l'Academie Royalle des Sciences, Docteur en Medecine de la Faculté de Paris.*

A PARIS,
Chez JEAN BAPTISTE COIGNARD,
Imprimeur ordinaire du Roy, ruë S. Jacques, à la Bible d'or.

M. DC. LXXXIV.
AVEC PRIVILEGE DE SA MAJESTE'.

Principles from Classical Architecture

by Vitruvius, Roman Architect, 1st Century BC

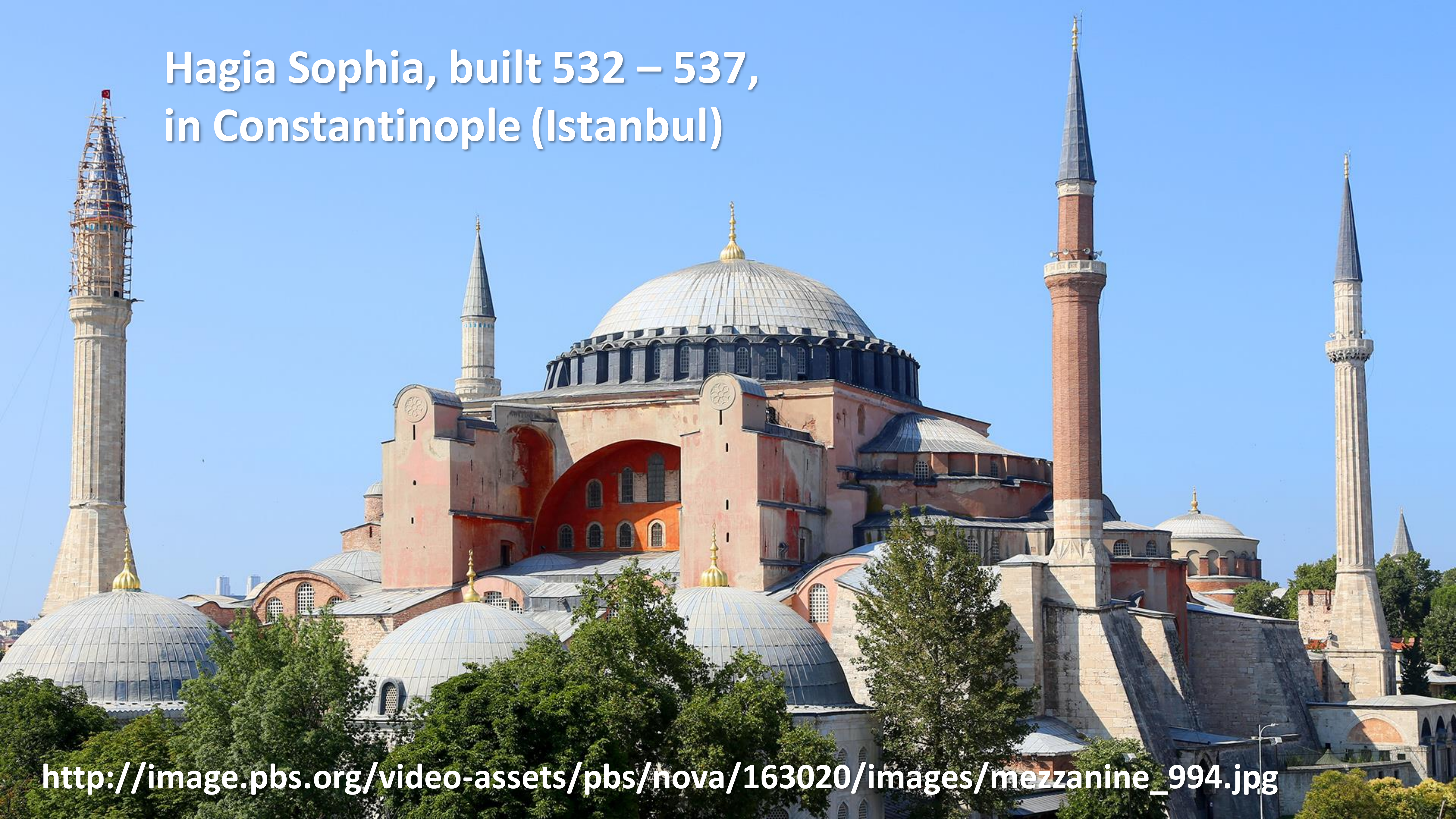
- **Utility**
  - Does the building conveniently serve its intended purpose?
- **Strength**
  - Will the building stand? Are the foundations solid and have the materials been wisely selected?
- **Beauty**
  - Is the appearance of the building pleasing and in good taste? Are the elements of the building in due proportion

Principles from Classical Architecture

Software Equivalents for Vitruvius' Principles

- **Utility**
  - Does the system meet its requirements?
- **Strength**
  - Is the system robust? Will it scale and perform? Is the technology appropriate?
- **Beauty**
  - Is the implementation of the system elegant? Is it easy to understand and modify?

Hagia Sophia, built 532 – 537, in Constantinople (Istanbul)



http://image.pbs.org/video-assets/pbs/nova/163020/images/mezzanine_994.jpg

Principles from Classical Architecture

Example: The Hagia Sophia

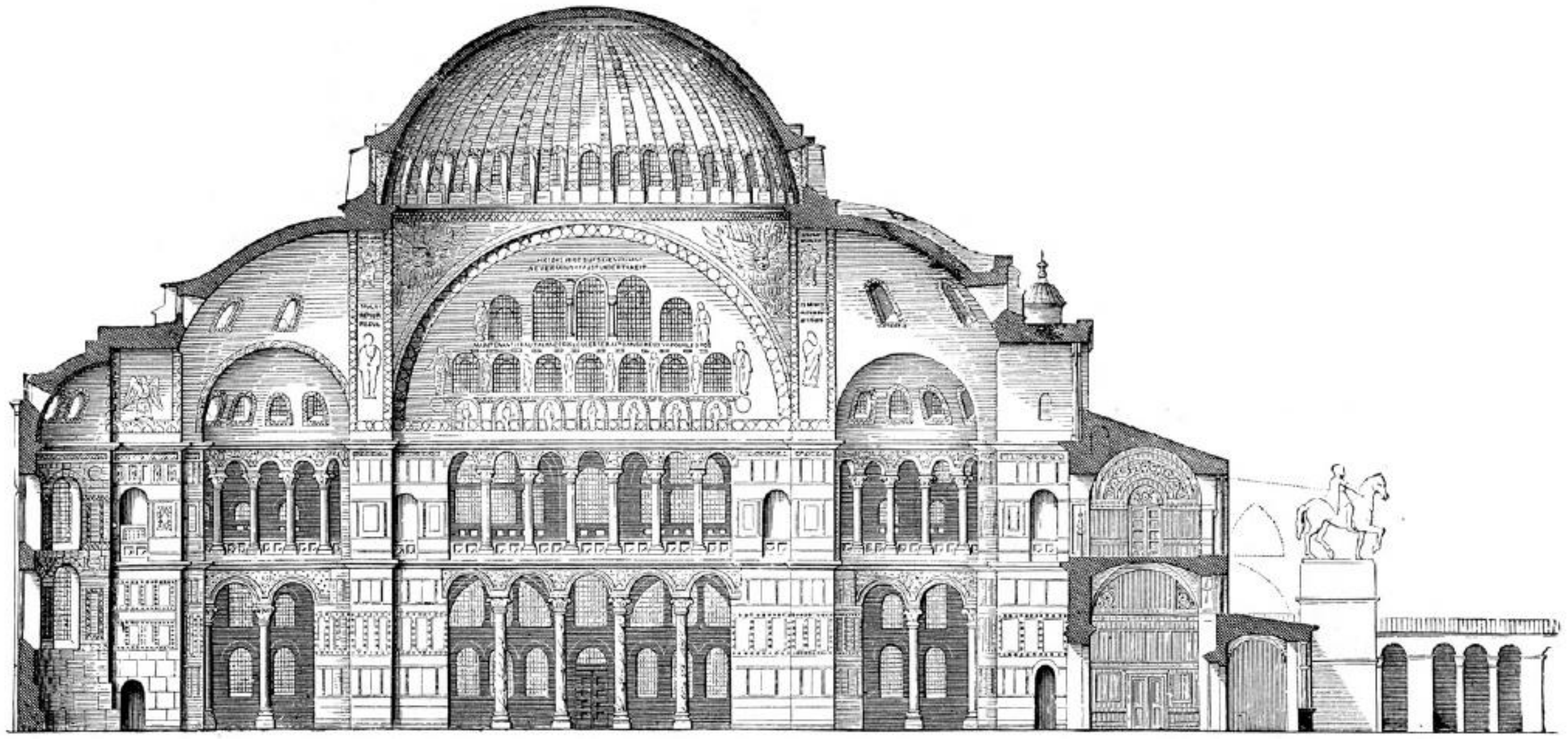
- **Utility**
  - Fulfills Emperor Justinian I’s wish for a majestic church, grander and more imposing than all its predecessors
- **Strength**
  - Stands tall, almost 1500 years after it was built
  - The main dome was re-architected, but that was in 562 AD
- **Beauty**
  - Intrinsic to its architecture and proportions
  - The main dome soars 182 feet from the floor
  - As a museum, it attracts millions of visitors every year

Architectural Views

- **Utility, strength, and beauty are different perspectives on the same architecture**
- **An *architectural view* focuses on some aspect of a given architecture**
  - Views are tailored to the intended purpose and the intended audience
  - Typically a view outlines how the architecture solves a problem
- **No one view captures everything about an architecture**

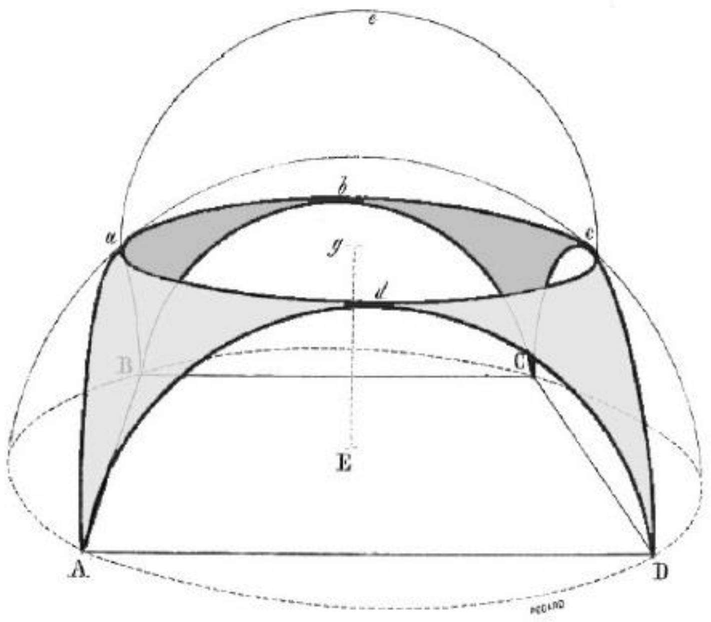
Hagia Sophia: Structural View

Note the round main dome and half-domes

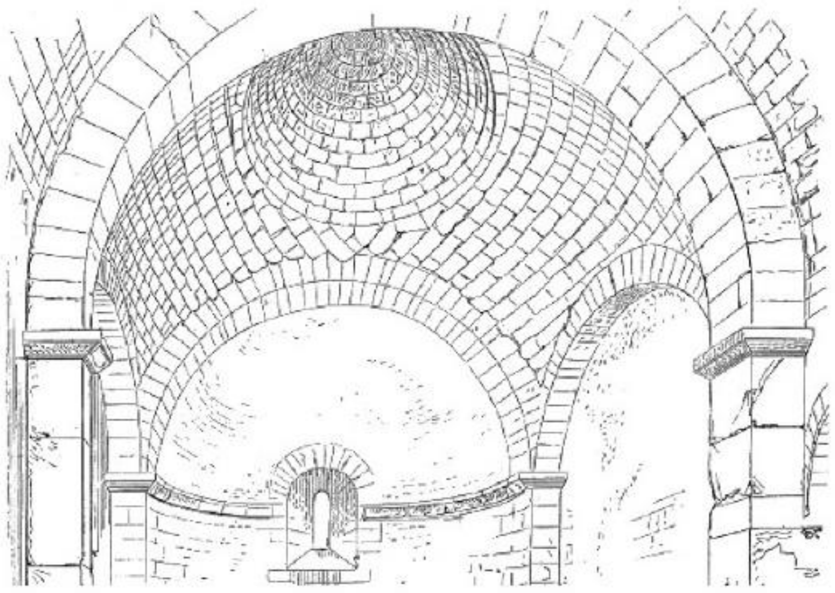


Pendentive: Round Dome on a Square Base

- A pendentive distributes the load of the dome onto the base

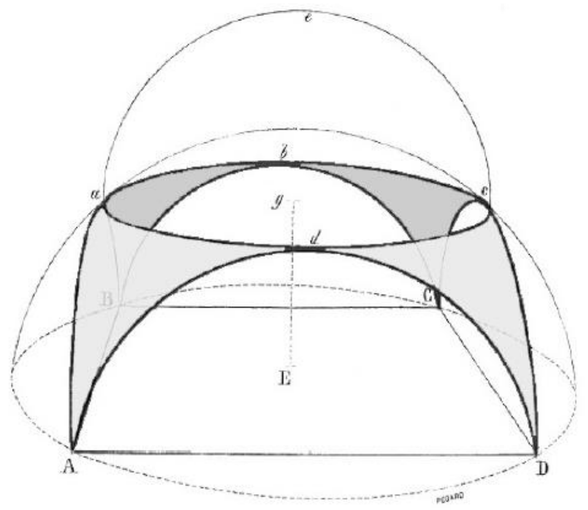


(a)


(b)

Hagia Sophia Dome Re-architected

- The original dome collapsed during the earthquake of 558
- The rebuilt dome is 30 feet higher to better distribute its weight to the supporting walls
- The re-architected dome is still standing, ~1500 years later



Architectural Patterns

- **First studied by architect and urban planner Christopher Alexander**
- **A *pattern* is a problem that occurs over and over again, together with "the core of the solution to that problem"**
- **Applications to Software**
  - Inspired object-oriented design patterns
  - Software architecture patterns
  - Pattern Languages of Programming conferences
  - Extreme Programming was influenced by Alexander’s work, especially the belief that the occupiers of a building should design it
  - …

Alexander's Patterns

- **Context**
  - Each pattern has both a larger and a smaller context
  - e.g., larger: roof completes a room, a room is part of a building, …
- **Problem**
  - Some fundamental aspect of a design
  - e.g., the design of roofs for a cluster of buildings
- **"Core" of a Solution**
  - Guidance for designing a specific structure
  - e.g., high ceilings for public rooms lower for smaller gatherings very low in rooms or alcoves for one or two people

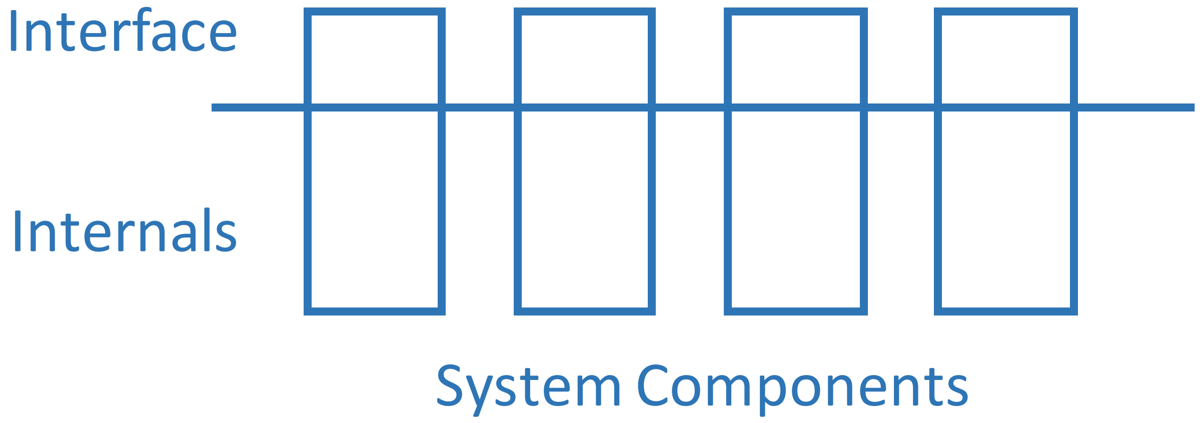
Software Architecture

- SEI has collected over 200 definitions of software architecture
  - "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').
  - "But the amount of detail alone is insufficient to characterize the differences, because architecture and design documents often contain detail that is not explicit in the implementation (e.g., design constraints, standards, performance goals)."

Distinction between Design and Architecture

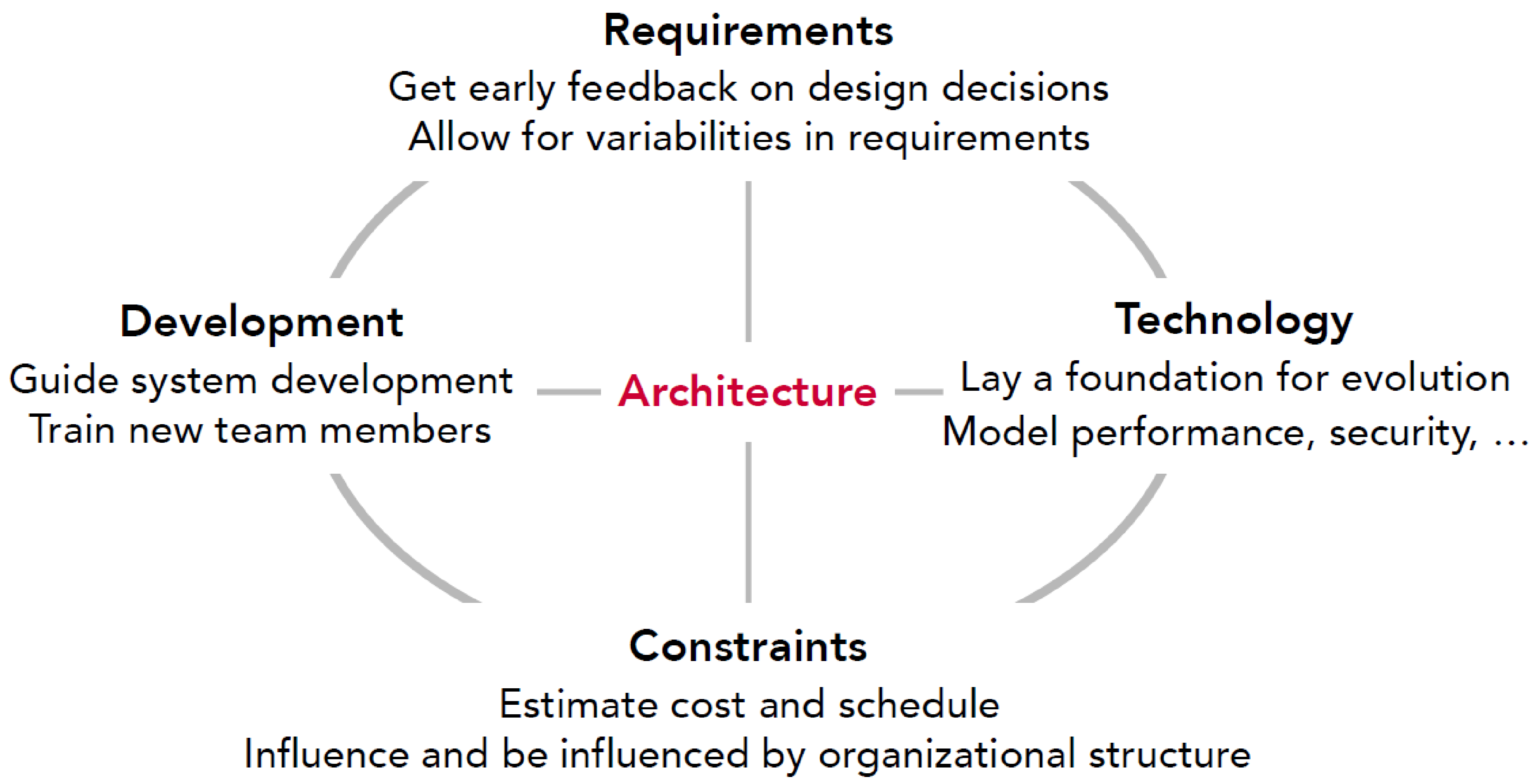
- **Architecture is part of the design of a system**
  - And is thus a subset of design

Architecture deals with relationships among components, externally visible properties



Design includes the internal structure of components

Role and Benefits of Software Architecture


Requirements
Get early feedback on design decisions
Allow for variabilities in requirements
Development
Guide system development
Train new team members
Architecture
Technology
Lay a foundation for evolution
Model performance, security, …
Constraints
Estimate cost and schedule
Influence and be influenced by organizational structure

Example: Conway's Law

- **"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." – Melvin Conway**
- **Sociological Observation**
  - Two software modules A and B …
  - cannot interface correctly with each other …
  - unless the designer of A communicates with the designer of B.
- **And so:**
  - the interface structure of the system *necessarily* reflects …
  - the social structure of the organization that produced it

Example: Identifying Potential Security Risks

- **Architecture**
  - Apps use features provided by iOS
  - Some of the API's are for Apple's own use (called private APIs)
  - Enterprises may use private APIs for their own use
- **Qihoo distributed apps that used private APIs**
  - Misuse of private APIs can pose a security risk
  - Hence, apps from Qihoo were banned by Apple

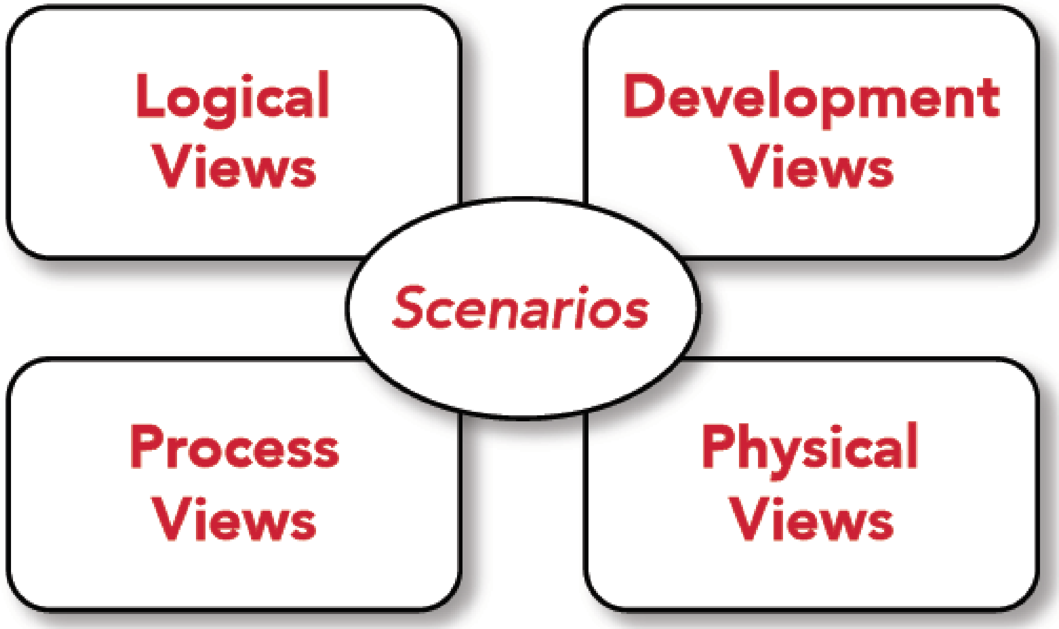
Example: Architecture and Work Assignment

- Microsoft used a modular architecture to accelerate development of its browser in 1996
- Allowed team to develop sub-components in parallel and allowed the system to be delivered sooner
- "If someone asked what the most successful aspect of [Internet Explorer 3.0] was, I would say it was the job we did in ‘componentizing’ the product.“
  - Development team member, IE 3.0. (Source: MacCormack 2001).

4+1 View Model

Focus:
User
Functionality

Focus:
Static
Modules

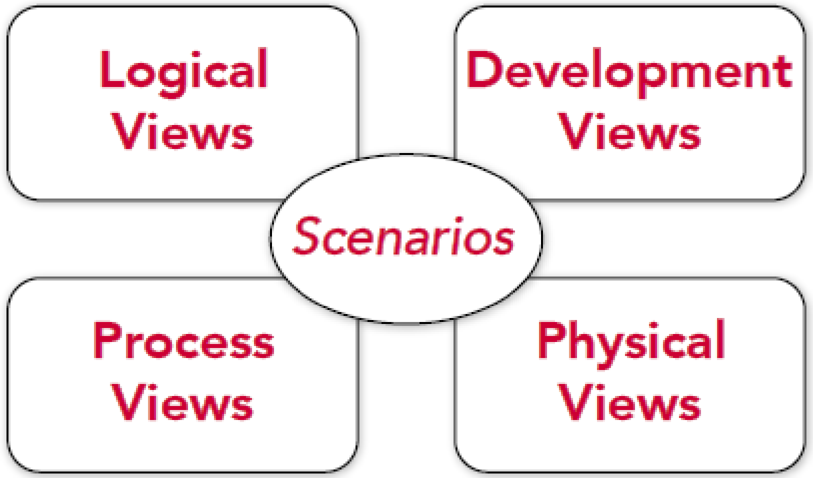


Focus:
Dynamic
Concurrency

Focus:
Physical
Configuration

4+1 View Model

- Logical Views
  - Focus on structures that support requirements and end-user functionality
- Development Views
  - Focus on modules in static source
- Process Views
  - Focus on dynamic or runtime processes
- Physical views
  - Focus on the configuration and physical distribution of the system e.g., allocation of processes to servers



Information Hiding and Modules

- Goals of modularization: make the system easier to:
  - Understand, integrate and build, maintain (modify), test, verify, develop in collaboration with others
- Information hiding principle
  - Hide independently-changeable information, such as design decisions, in independently-changeable modules
  - Aim for well-defined interfaces that are stable over time that hide module implementations
- Modules secrets
  - Design decisions hidden inside a module

Information Hiding Example:

- Software for counting items for store inventory
- Design decisions
  - How are counts represented?
  - Where are they stored?
  - How is top selling item identified?
- Decisions can be hidden in modules
- Interface to the module can answer questions about the counts
  - What is the count for a particular item?
  - What item has the highest count?
- Allows you to change implementation without changing the interface

Coupling and Cohesion

- Aim for loose coupling and high cohesion
- **Coupling** is the degree to which modules are inter-related
  - *Loosely coupled* if interaction is only through interfaces
  - *Tightly coupled* if the implementation of one module depends on the implementation of another

Coupling	**Modules**
Message	Pass messages through their interface
Subclass	Inherit methods and data from a superclass
Global	Two modules share the same global data
Content	One relies on implementation of the other

Coupling and Cohesion

- Aim for loose coupling and high cohesion
- **Cohesion** is the degree to which the elements of a module belong together
  - *High cohesion* if module has one secret and all elements relate to that secret
  - *Low cohesion* if it has elements that are unrelated

Cohesion	**Group elements based on**
Functional	One secret (e.g. parsing)
Sequential	Process steps (a la pipes in Unix)
Informational	Data that is manipulated
Temporal	Order in which events occur
Coincidental	Elements have little to do with each other

Microsoft Case Study

- Concepts of information hiding, coupling / cohesion, object oriented design date back to the 1970's
- Software developers still face problems
- 2005 survey by Venolia, DeLine, and LaToza asked software architects, developers, and testers at Microsoft what problems they face in developing software
- 7 of top 8 problems relate to modular structure of system

The following is "a serious problem for me"

• Understanding the rationale behind a piece of code	66%
• Having to switch tasks because of ... teammates or manager	62%
• Being aware of changes to code elsewhere that impact my code	61%
• Finding all the places code has been duplicated	59%
• Understanding code that someone else wrote	56%
• Understanding the impact of changes I make on code elsewhere	55%
• Understanding the history of a piece of code	51%
• Understanding who "owns" a piece of code	50%

Module Descriptions and Hierarchy

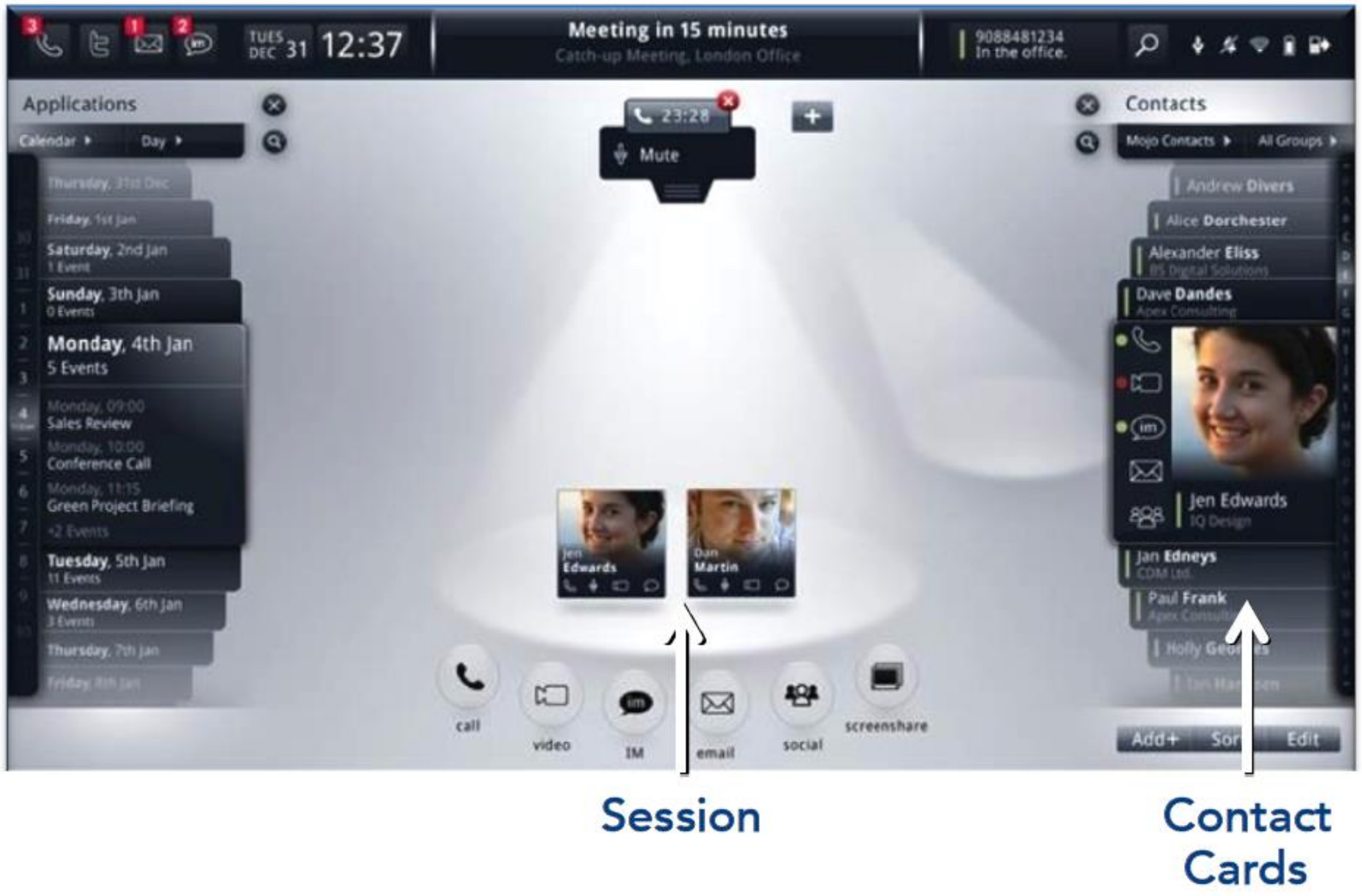
- A realistic system can have hundreds of modules
  - Finding relevant modules is difficult beyond a dozen or so
- Solution
  - Group related modules into a tree-structured hierarchy
  - Provide descriptions of the modules written in plain English
- In a module hierarchy, the secret of a child module is a subsecret of its parent module

Secret	Typical Change
How to monitor a sensor	New more reliable sensor, higher resolution sensor
How to control a device	Faster “larger” version of device
Platform characteristics	Faster processor, multi-processor, larger memory
How to control a display	Reorganization of screen real-estate, look and feel
How to exchange data	Protocol change
Database physical structure	Fields added, field access needed, field sizes change
Algorithm	Change in time-space tradeoff, more accurate algorithm invented
Representation of system entities such as jobs and users	Change in performance requirements, more system entities

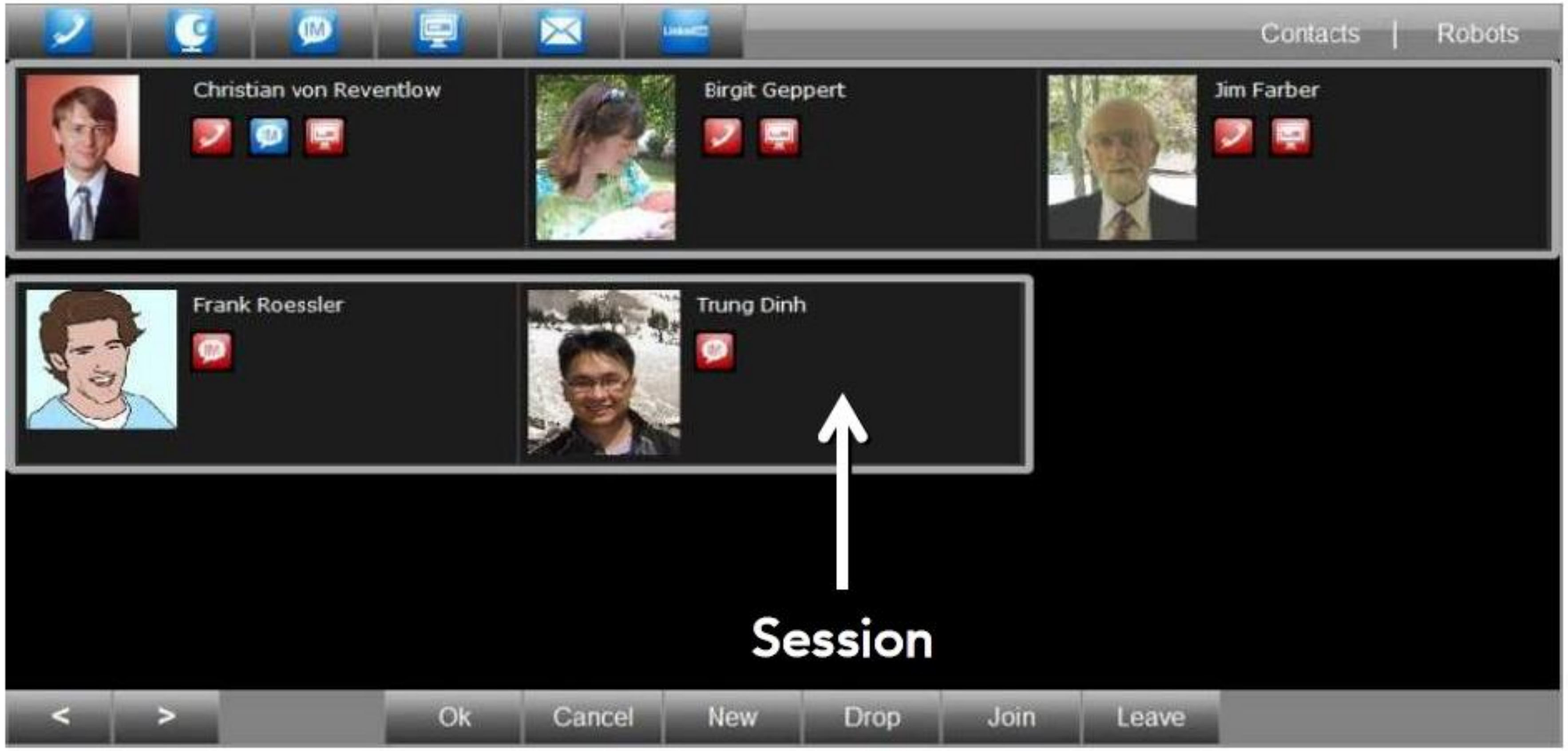
Module Descriptions

- Module Interface Specification
  - Defines services provided and services needed
  - Defines syntax and semantics for accessing services
  - Defines data types, program effects, ...
  - Defines test cases
  - Records design decisions and implementation notes
- Module Guide
  - Textual description of the module hierarchy, with each module described by its secret

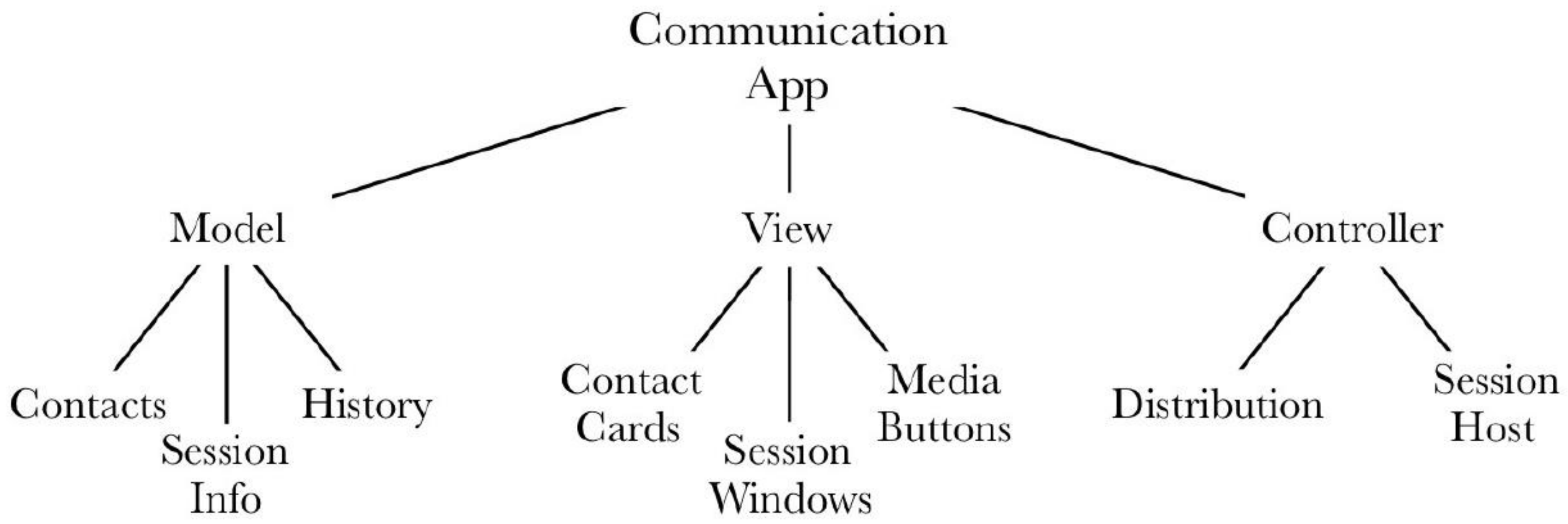
Example: Visual Communication App



Another View of the Same App



A Module Hierarchy



- Note: modules can use other modules across subtrees in the hierarchy
- e.g. contact cards can use contacts, without knowing how contacts are stored

Module Guide: Description in Plain English

- The purpose of a module guide:
  - Provide an overview of the system
  - Bring out the context and assumptions behind the design approach
  - Describe the responsibilities and behavior of the modules
- The guide supplements, not replaces, a specification of module interfaces

Template for a Module Guide

- Module Name
  - Textual description
    - The responsibility of the module
    - Overview and context for the service and the secret of the module
  - Service Provided
    - Service provided to the other modules through the module interface
  - Secret
    - Service provided to the other modules through the module interface
    - Any secondary design decisions that are needed for the implementation
  - Error and Exception Handling
    - List of possible errors and exceptions

Modules Summary

- Information hiding modules
  - Hide design decisions (secrets)
  - They can be viewed as black boxes with abstract interfaces
- Module hierarchy
  - Organizes information hiding modules into a tree structure
  - Described in the module guide
- Examples: Using modules services
  - Abstract data types: use data without knowing its representation
  - GUI creation environments: construct user interfaces without knowing how to display
  - Protocols: send and receive data – but hide channel details
  - Methods: invoke methods without knowing their implementation

Family and Product Lines

- "We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members" – *David L. Parnas*
- Software product line
  - Programs specifically designed and implemented as a family
- Software Product Lines Conference Hall of Fame
  - Honors organizations for commercially successful product lines
  - Link

Examples of Product Families

- iPods
- Airbus Airplanes
- Linux
- IBM 360
- Portable C Compiler
- Web Browsers

Product Lines

- A family of products designed to take advantage of its
  - Commonalities: common aspects of the family members, and
  - Variabilities: predicted range of differences between members
- A product line may be decomposed into sub-families
  - Each sub-family contributes a member to members of the product line
  - Sub-families may themselves be product-lines

Product Line Engineering

Underlying Assumptions

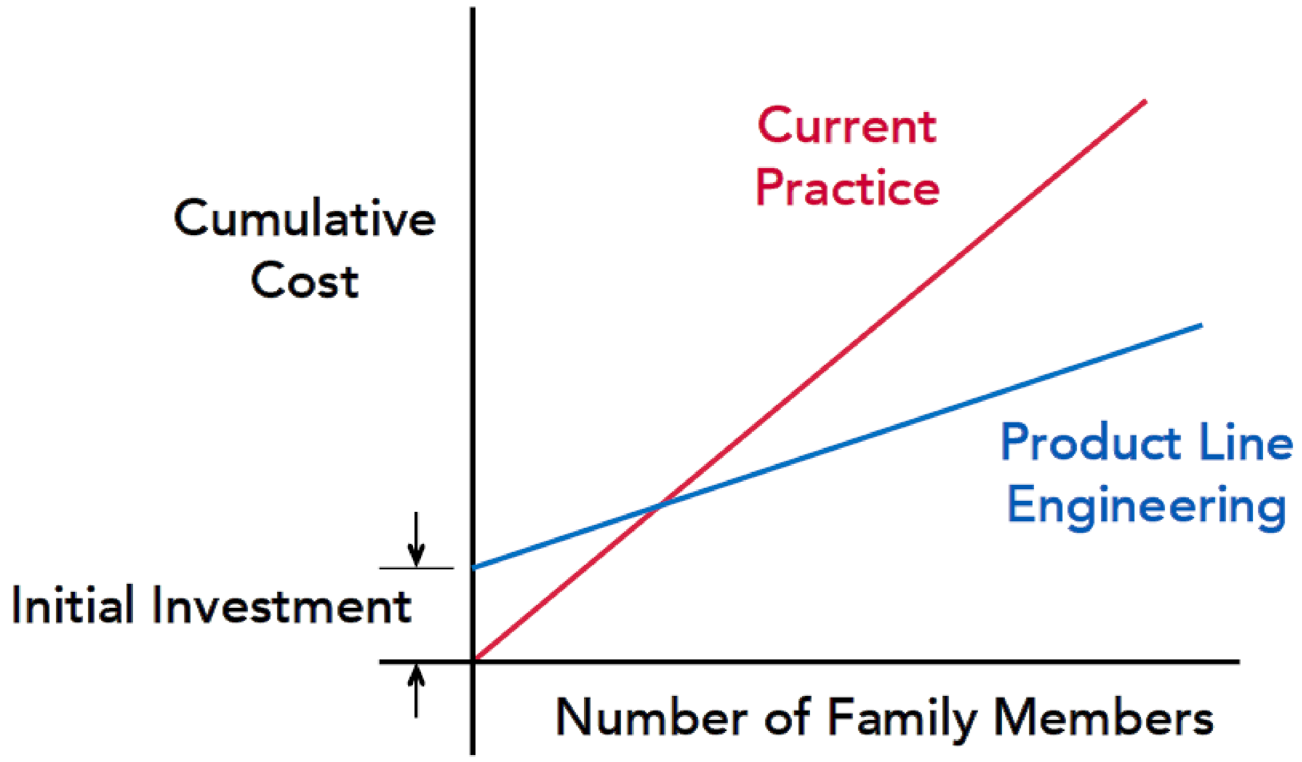
- Redevelopment Hypothesis
  - Most software development is mostly redevelopment
- Oracle Hypothesis
  - It is possible to plan for changes that are likely to be needed
- Organizational Hypothesis
  - It is possible to organize both software and the organization that develops it so as to take advantage of predicted changes

Product Line Engineering

Initial Investment

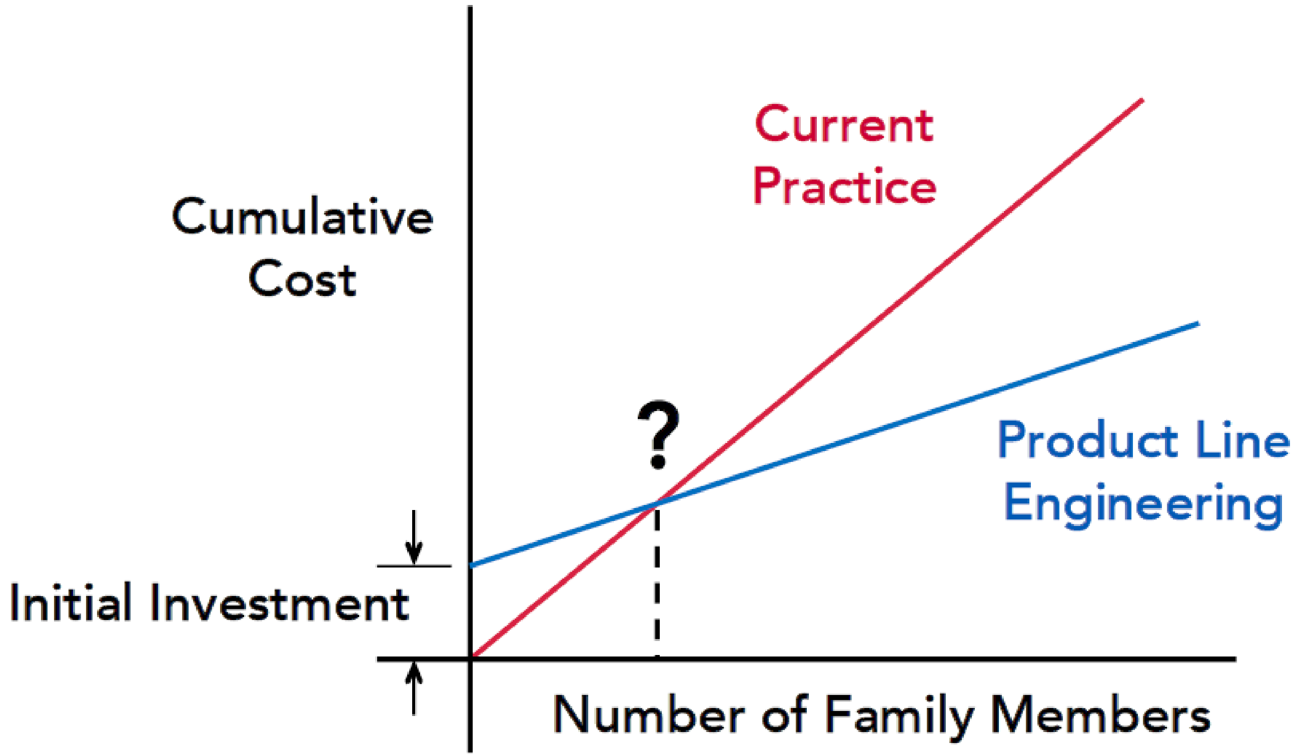
- Requires a high initial investment for
  - Identifying commonalities and variabilities
  - Creating a business case that encompasses multiple products
  - Developing a modular architecture that hides variabilities
  - Building test plans that span products
  - Conducting additional training for developers and managers
- Management support is essential
  - Projects that lack management support often fail to deliver improvements promised by product-line engineering

Economics of Product Line Engineering



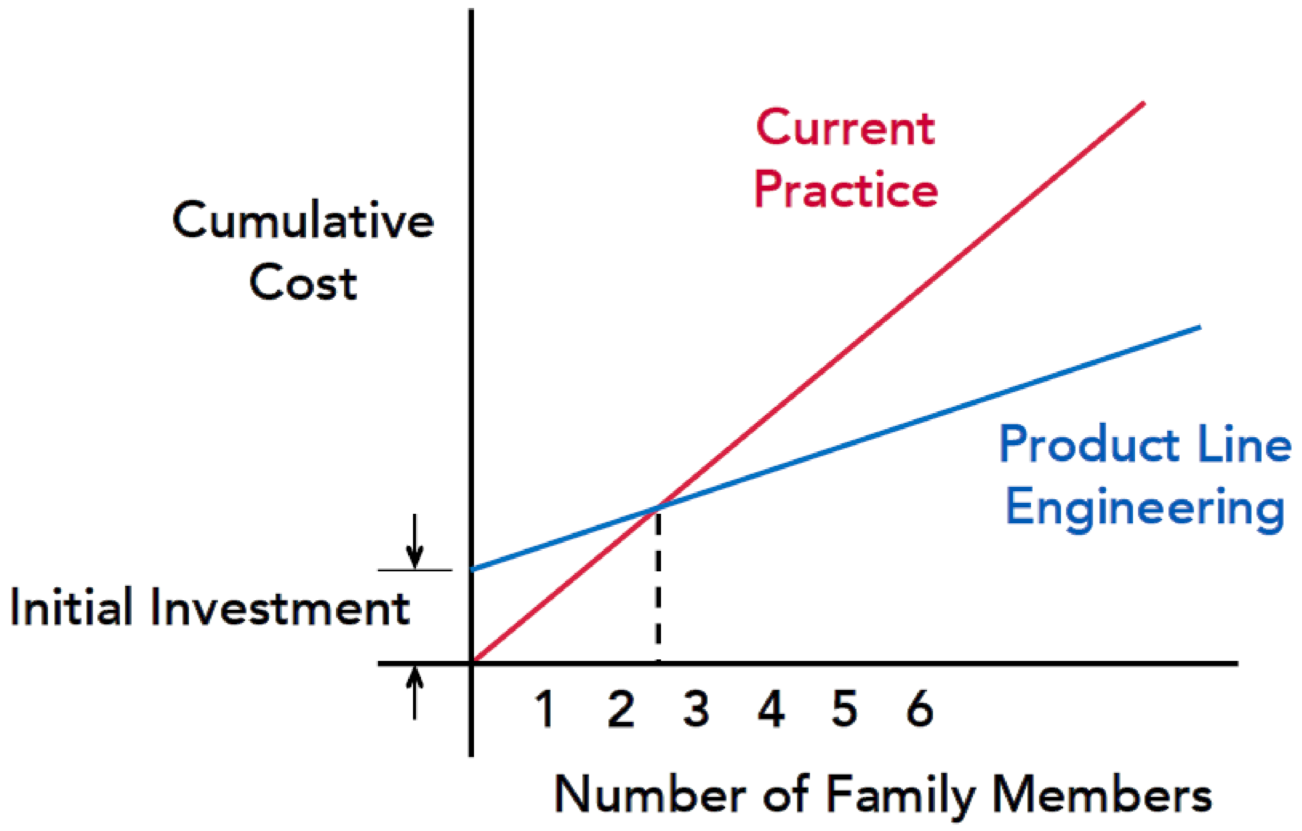
- Initial investment pays off as members of the product family are developed

Economics of Product Line Engineering



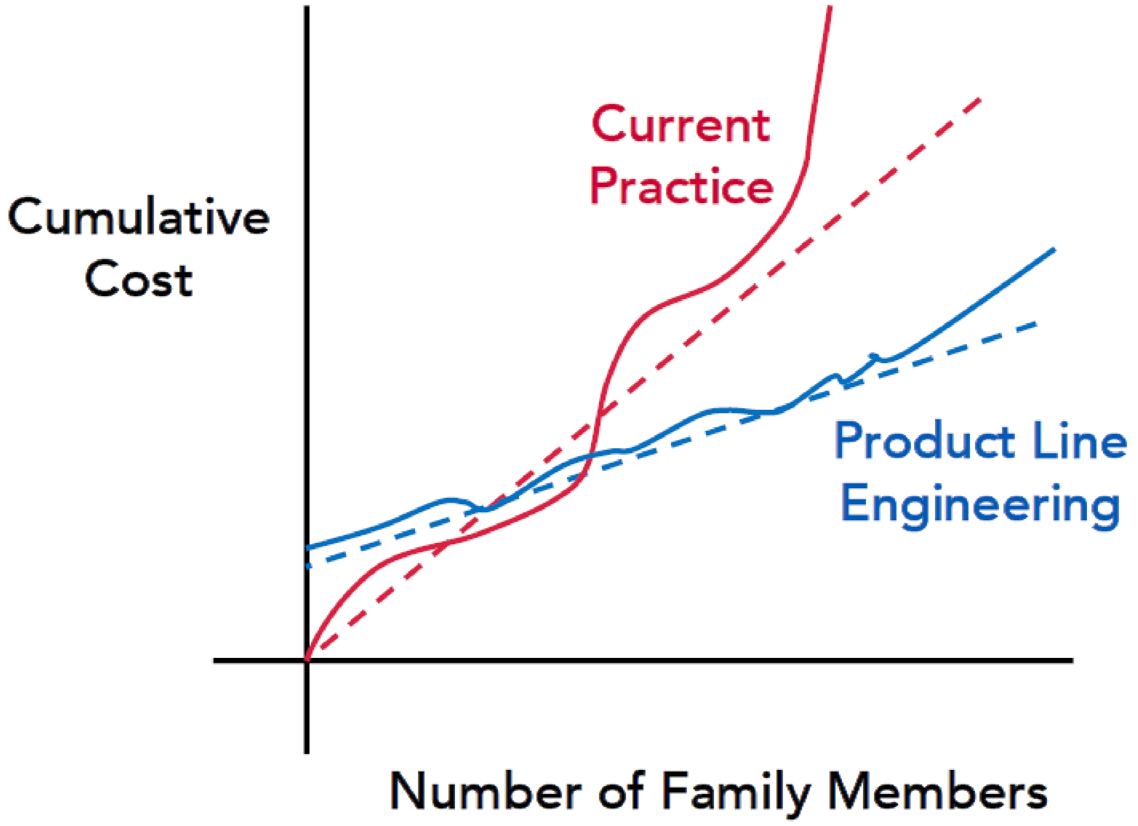
- Crossover point of when investment for product families starts to pay off

Economics of Product Line Engineering



- Bell Labs experience with several projects: crossover between 2 and 3

Economics of Product Line Engineering



- A more realistic picture

Unix Portability

- In 1977, Unix ported from the PDP-11 to the Interdata-8/32
- The portability of Unix led to Open Systems from multiple vendors in the 1980s
  - Open System refers to some combination of interoperability, portability, and open software standards

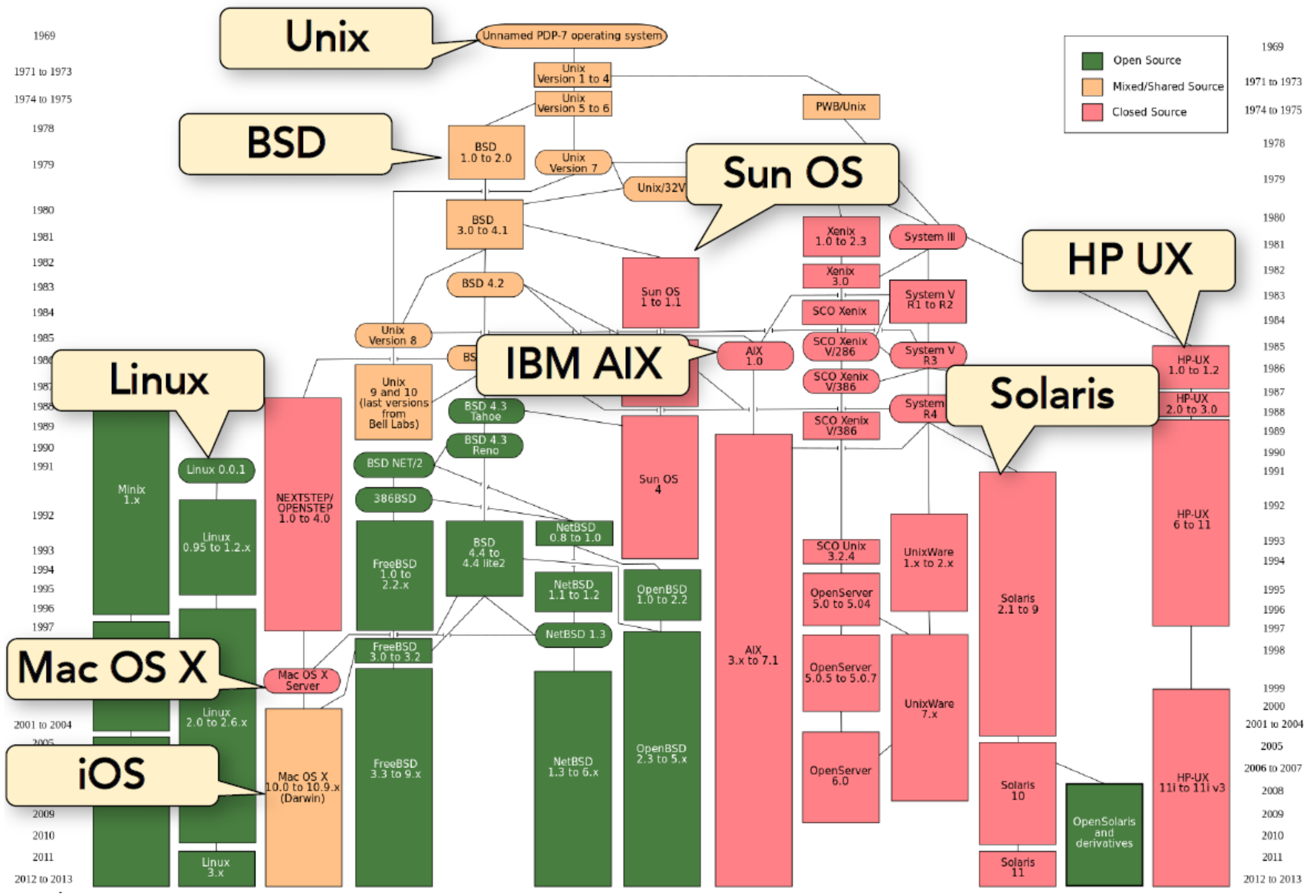
Why Study Unix Portability?

The portability of Unix has enormous significance for computing

- Illustrates information hiding
  - Unix was written in a high-level language (C) to hide the machine
  - The Portable C Compiler hid language dependencies in the front end and machine dependencies in the back end
- Case Study: illustrates design decisions
  - Dealing with inherent machine dependencies
  - Performance trade-offs
- Well written account

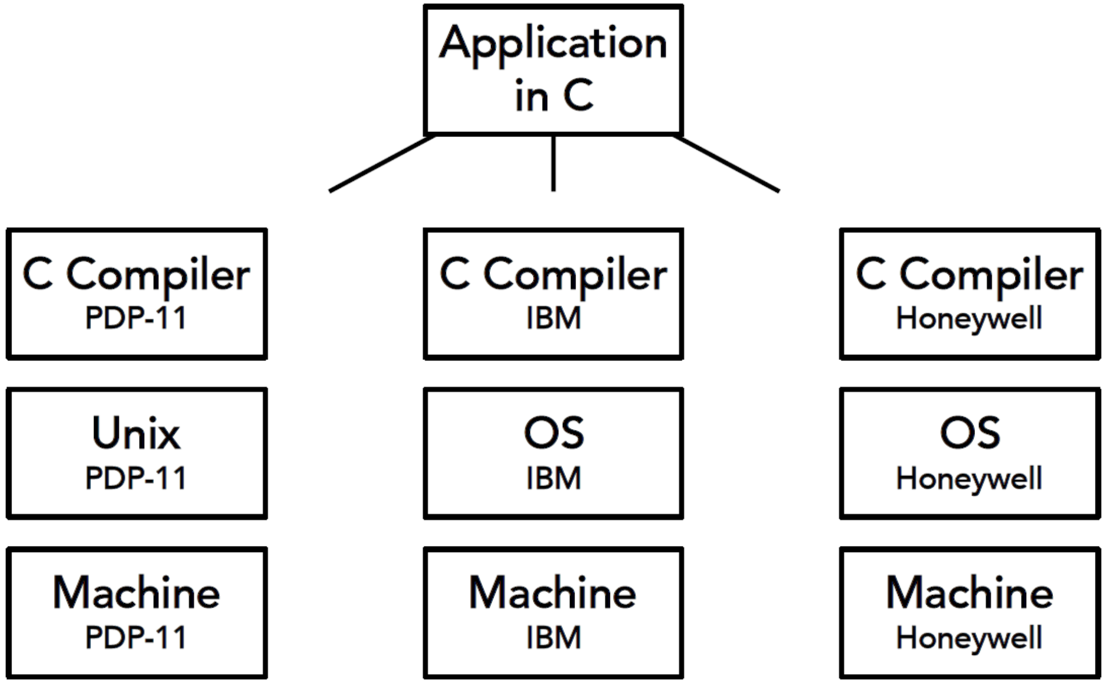
The Unix Family

“The real growth of Unix began only after portability had been achieved.”



Portability of C Programs

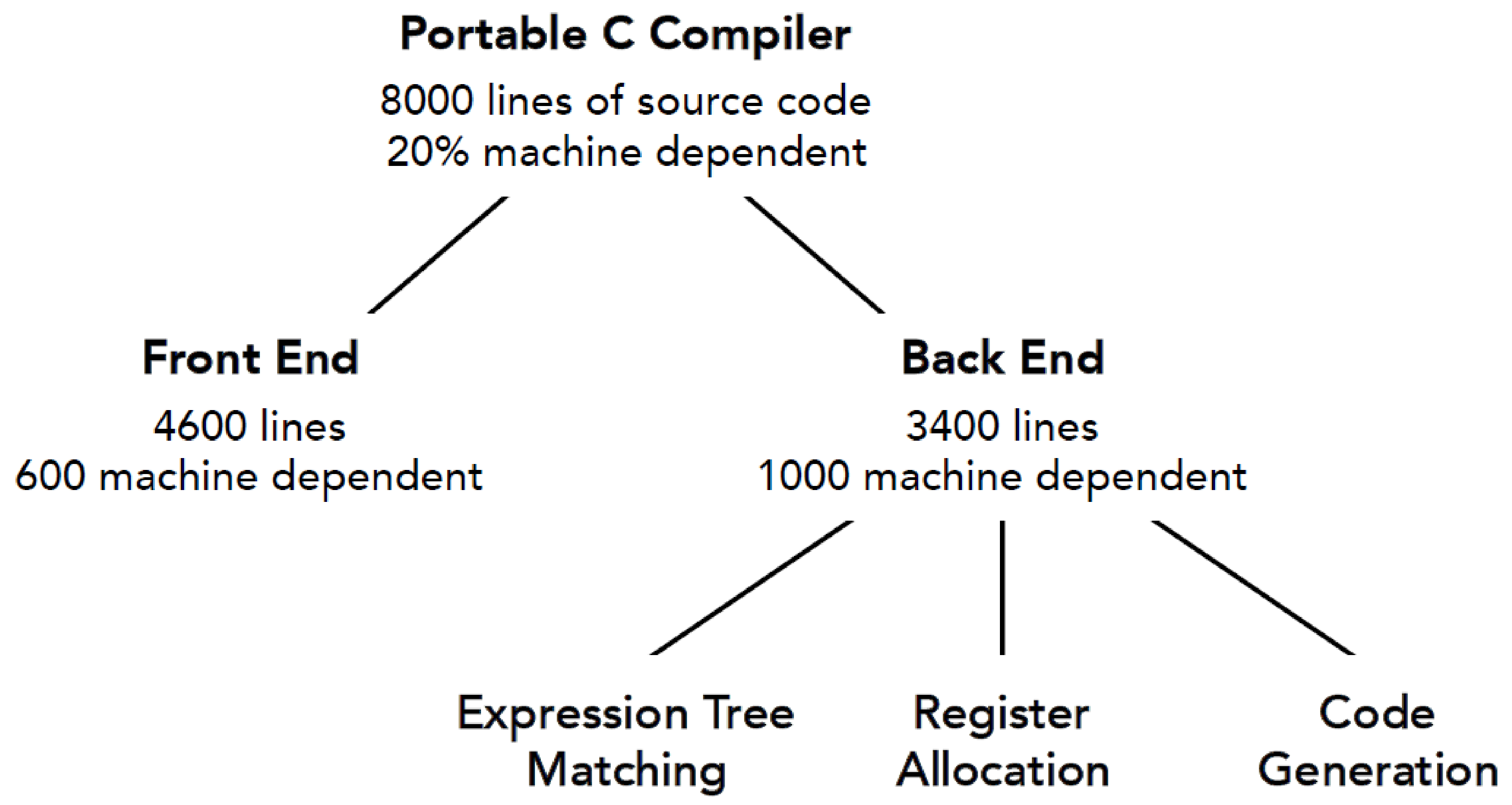
- Although portability was not a goal for C, Unix applications written in C ported to other machines with relative ease
- "the operating system interface caused far more trouble for portability than the actual hardware or language differences them-selves"



Unix Portability Project: Goals

- Write a portable C Compiler
  - "To write a compiler for C that could be changed without grave difficulty to generate code for a variety of machines."
- Refine C for portability
  - "To refine and extend the C language to make most C programs portable to a wide variety of machines, mechanically identifying non-portable constructions where possible."
- Rewrite and port Unix
  - "To revise or recode a substantial portion of UNIX in portable C, detecting and isolating machine dependencies, and demonstrate its portability by moving it to another machine."

Portable C Compiler: Module Hierarchy

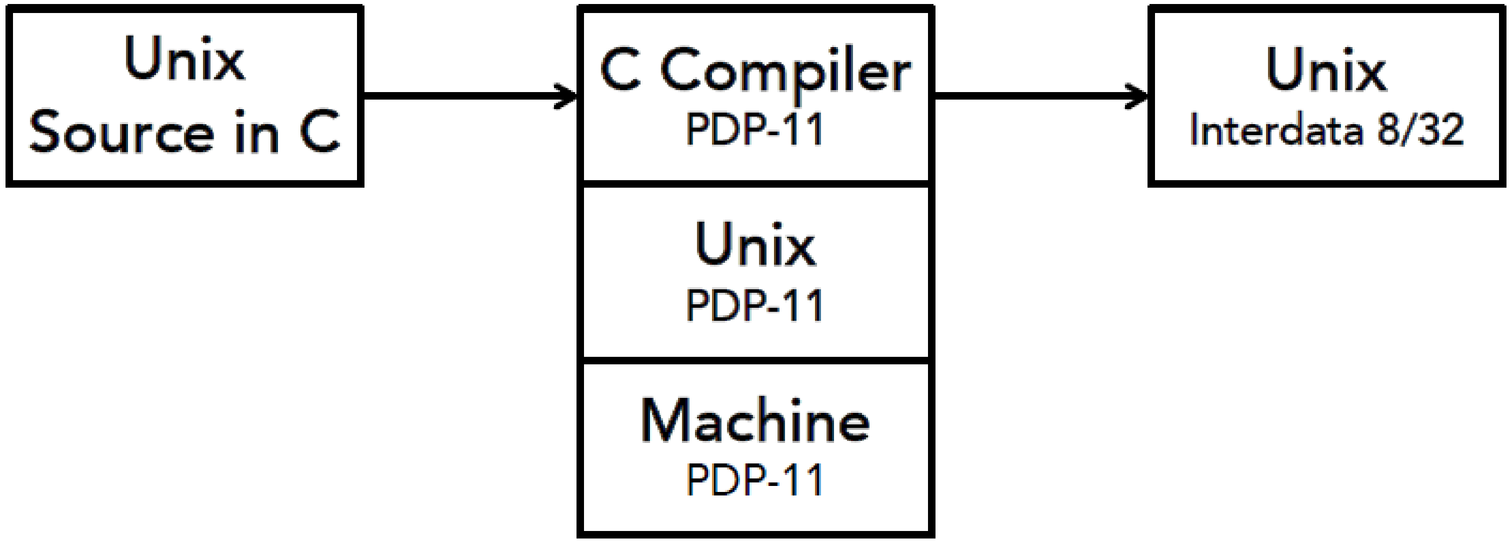


Portable C Compiler: Information Hiding

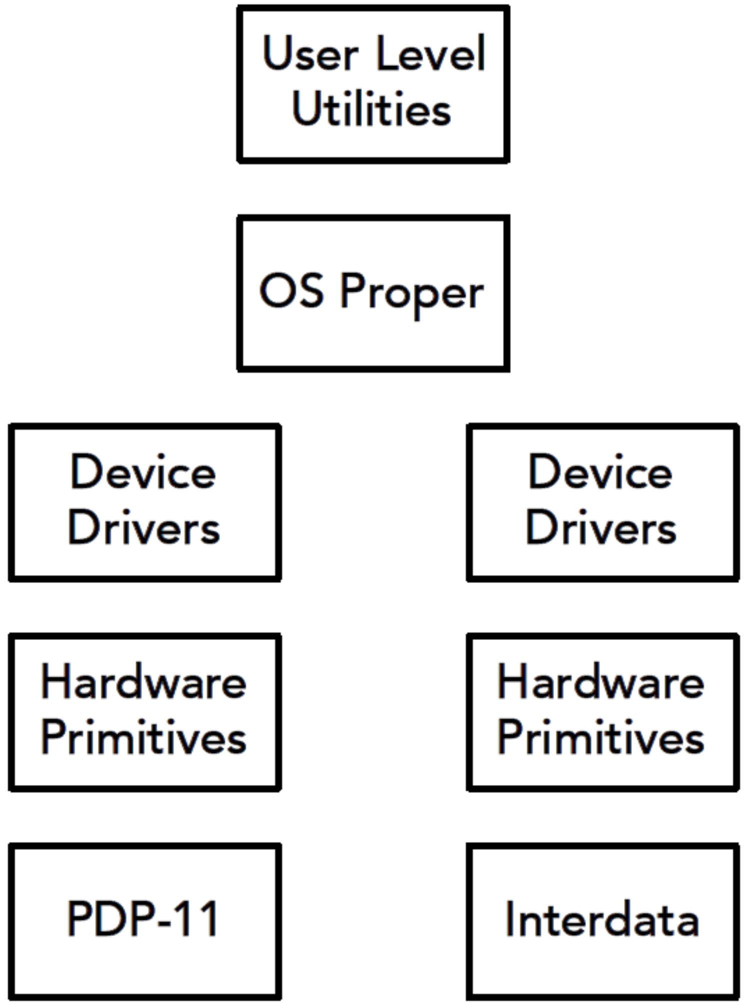
- Front end hides the source language
  - Generates an intermediate representation consisting mostly of expression trees and stylized code for subroutine entry/exit
  - 13% machine dependent lines; e.g., for subroutine entry/exit
- Back end hides the target machine
  - The back end source is surprisingly machine independent (70+%)
  - Even in the machine dependent routines, only a third to a half vary across machines.
- Assessment
  - Within months the compiler was running on a multitude of machines
  - A Fortran 77 compiler was created by reusing back ends
  - In the machine independent portions, a bug could be fixed in all versions almost mechanically

Unix Portability Project

- "Transportation of an operating system and its software between nontrivially different machines [was] rare, but not unprecedented"



Unix Portability Project



- ~ 50,000 lines in C
  - 20,000 identical between machines
- ~ 7,000 lines in C on the Interdata
  - 350 differed, PDP-11 to Interdata
- ~ 1,100 lines in C, machine specific
  - Interrupts, I/O, error handling
- ~ 800 lines of assembly (Interdata)
  - Most of the bugs appeared here

Unix Portability Project: Experience

- High portability achieved
  - The operating system was 95% the same on the two machines, outside of hardware primitives and device drivers
  - Inherently machine-specific software – compiler, assembler, loader, debugger was 75-80% unchanged
- Limitations to portability
  - Machine model: tradeoff between using powerful machine features and maintaining machine independence
  - With scale, algorithms need to be revisited; e.g., sorting ten items is different from sorting millions

Self-Hosting Programming Languages

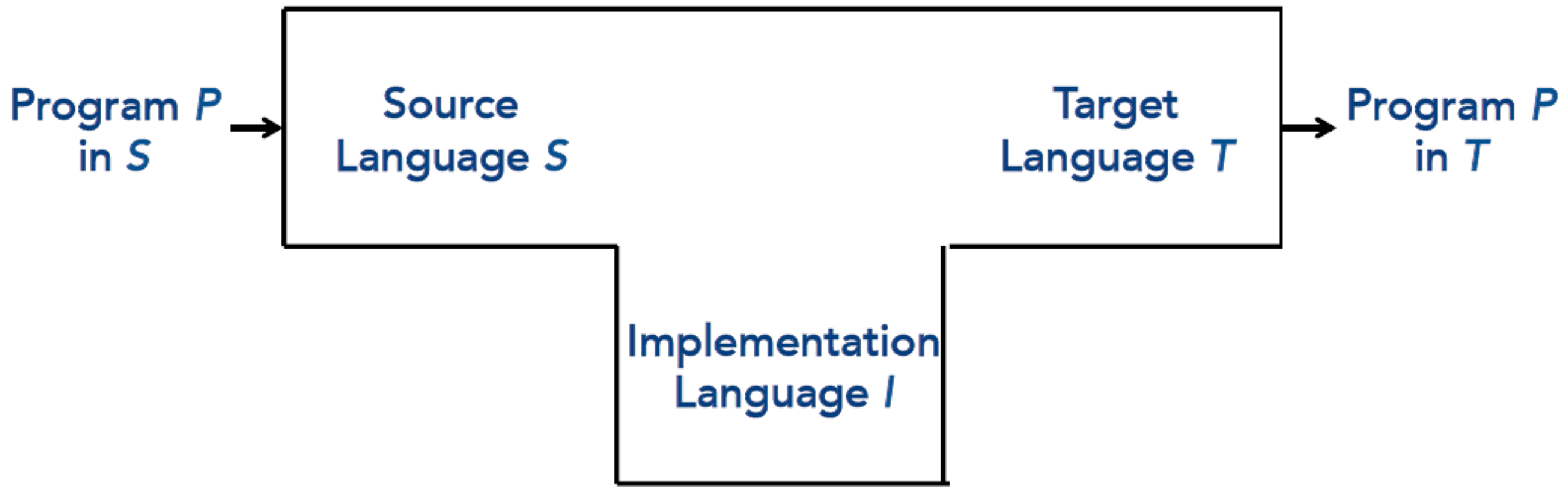
- Self-hosting principle: a programming language should allow enough expression that it can be implemented in its own language
  - e.g., a Pascal compiler written in Pascal is self-hosting, but a Pascal compiler written in C is not.
- Allows the language developer to use features of the language for which they are responsible
  - Creates a virtuous cycle for language implementers
  - Need expressive language features to develop performance-critical parts of the language implementation
  - Often leads to innovative ways to efficiently implement language features

Self-Hosting Java

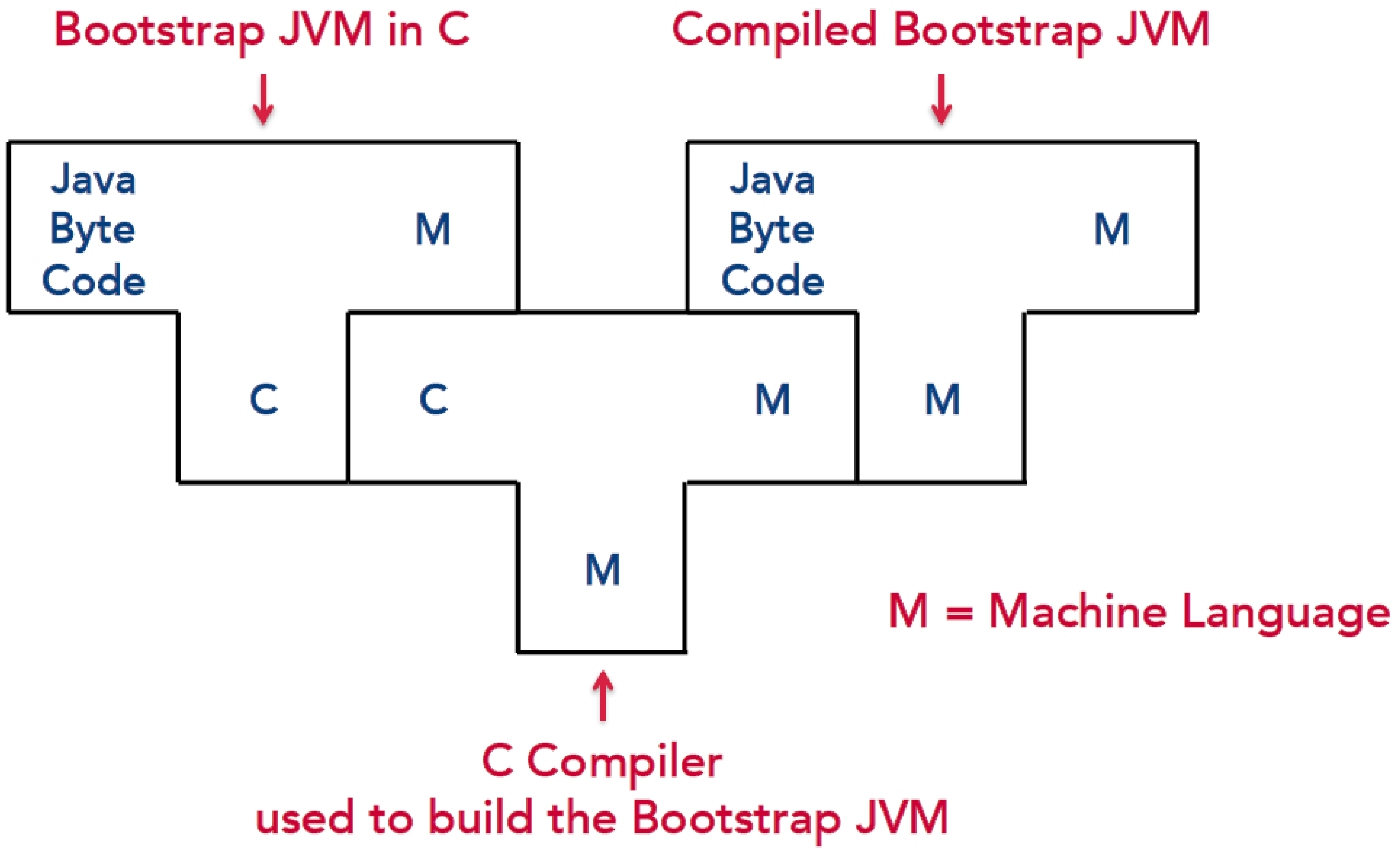
- Allows the runtime implementation to take advantage of HLL features
  - Automatic memory management / memory safety features
  - Standard threading / concurrency libraries
  - Exceptions and exception handling routines
  - etc. …
- Simplifies interface between runtime and application code
- JIT-compilation of runtime methods

Bootstrapping

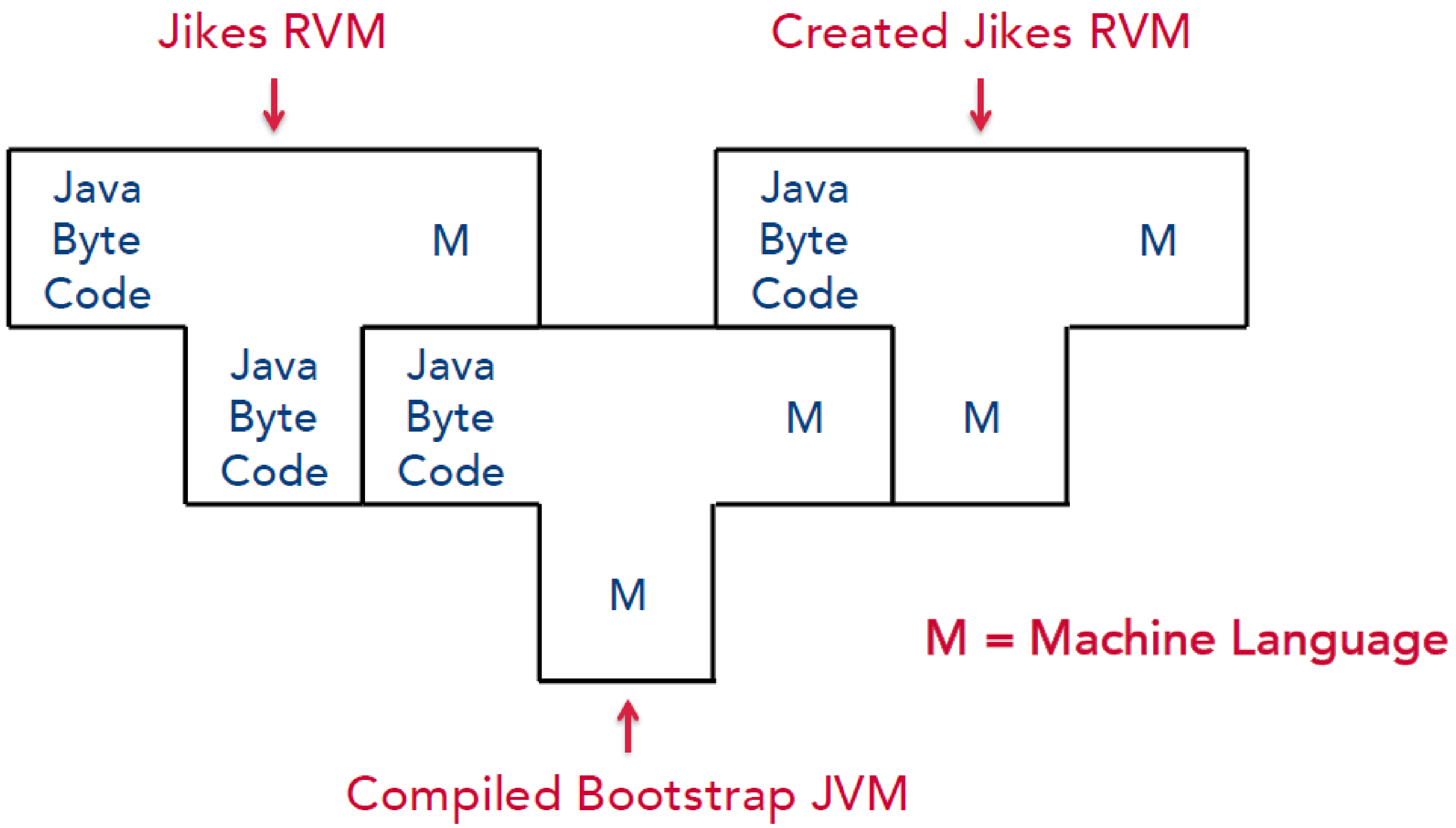
- A T-Diagram depicts a compiler from source language *S* to target language *T*, written in implementation language I



Creating the Bootstrap JVM



Creating the Self-Hosting Runtime System

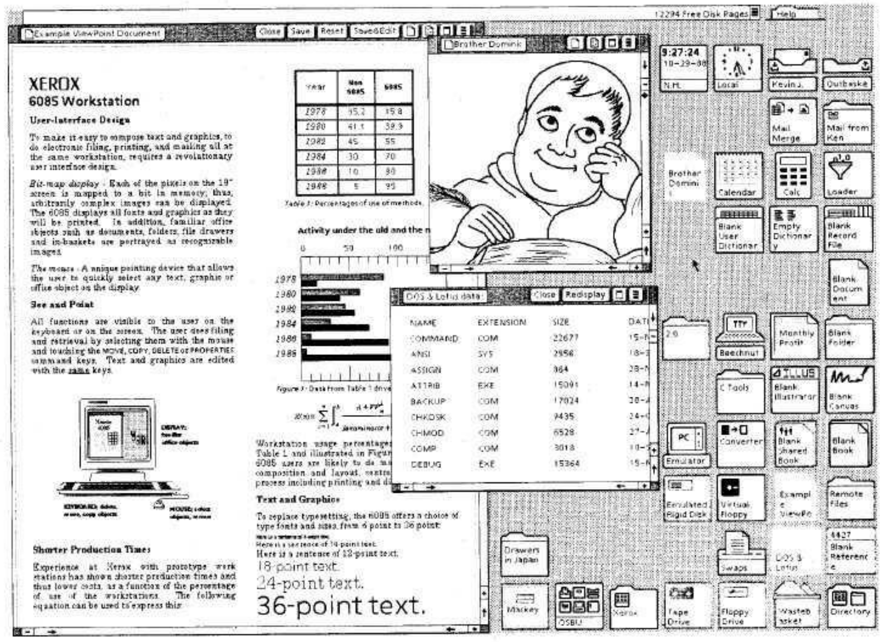


Architectural Patterns

- An **architectural pattern** is
  - a package of design decisions that is found repeatedly in practice, has known properties that permit reuse, and describes a *class* of architectures
    - from *Software Architecture in Practice* by Bass, Clements, and Kazman (2013)
- Patterns are abstracted from software found in practice
  - What is and is not a pattern depends on your point of view
  - There is no complete catalog
- Experienced architects typically adapt patterns
  - Either consciously or unconsciously work with patterns
  - Or, simply mimic successful solutions

Model View Controller

- Originally developed for Smalltalk at Xerox PARC by Trygve Reenskaug in 1979, the pattern has been widely adopted for web applications
- The Smalltalk group produced the first WIMP (Windows, Icon, Mouse, Pointer) user interface



XEROX
6085 Workstation

User-Interface Design

To make it easy to compose text and graphics, to do electronic filing, printing, and mailing all at the same workstation, requires a revolutionary user interface design.

Bit-map display - Each of the pixels on the 19" screen is mapped to a bit in memory; thus, arbitrarily complex images can be displayed. The 6085 displays all fonts and graphics as they will be printed. In addition, familiar office objects such as documents, folders, file drawers and in-baskets are portrayed as recognizable images.

The mouse - A unique pointing device that allows the user to quickly select any text, graphic or office object on the display.

See and Point

All functions are visible to the user on the keyboard or on the screen. The user does filing and retrieval by selecting them with the mouse and touching the MOVE, COPY, DELETE or PROPERTIES command keys. Text and graphics are edited with the same keys.

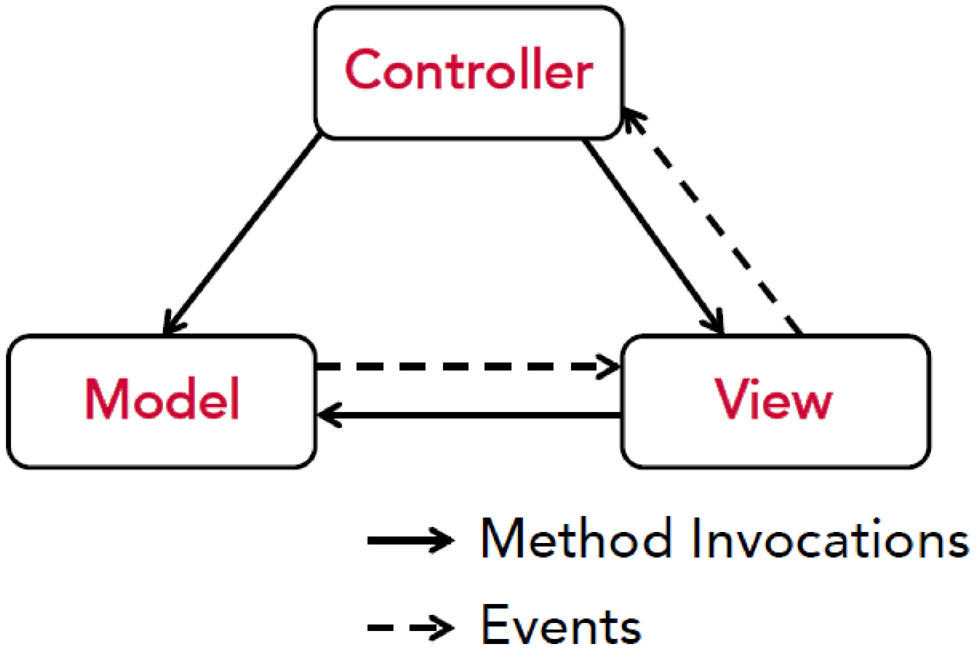
NAME	EXTENSION	SIZE
COMMAND	COM	22677
ANSI	SYS	2956
ASSIGN	COM	864
ATTRIB	EXE	15091
BACKUP	COM	17024
CHKDSK	COM	9435
CHMOD	COM	6528
COMP	COM	3018
DEBUG	EXE	15364

Text and Graphics

18-point text.
24-point text.
36-point text.

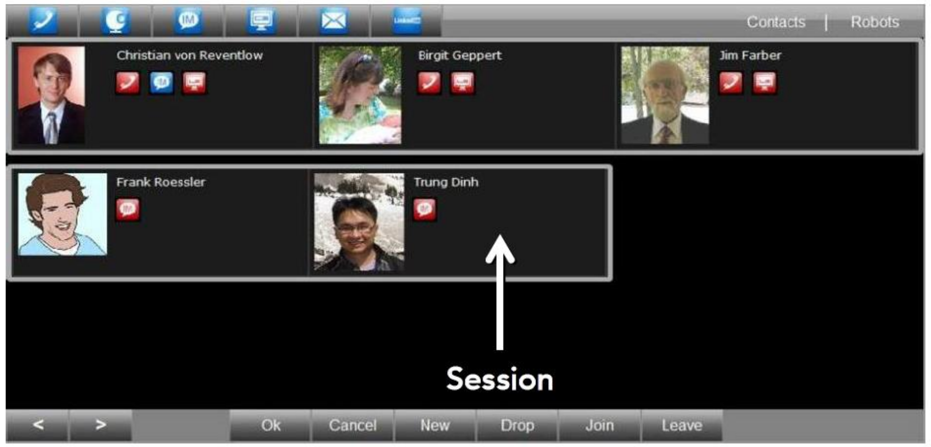
Model View Controller Pattern

- Core of a solution
  - Model supports application behavior
    hides state of the application
  - View manages presentation to the user
    hides the display device
  - Controller interprets user actions
    hides mapping from events to state changes
- Traditional Examples
  - Graphical User Interfaces



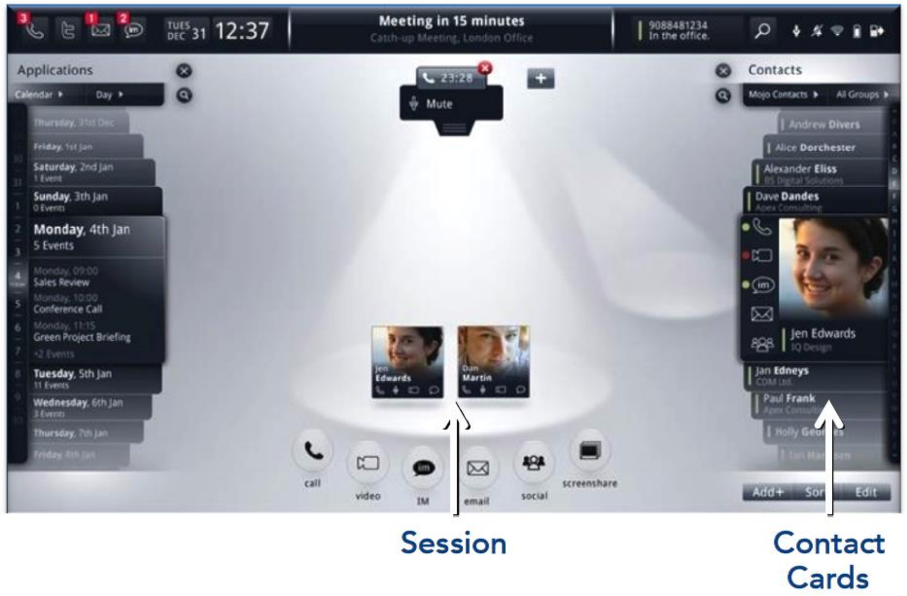
Application of Model View Controller

Labs and production versions of the Avaya Flare Experience

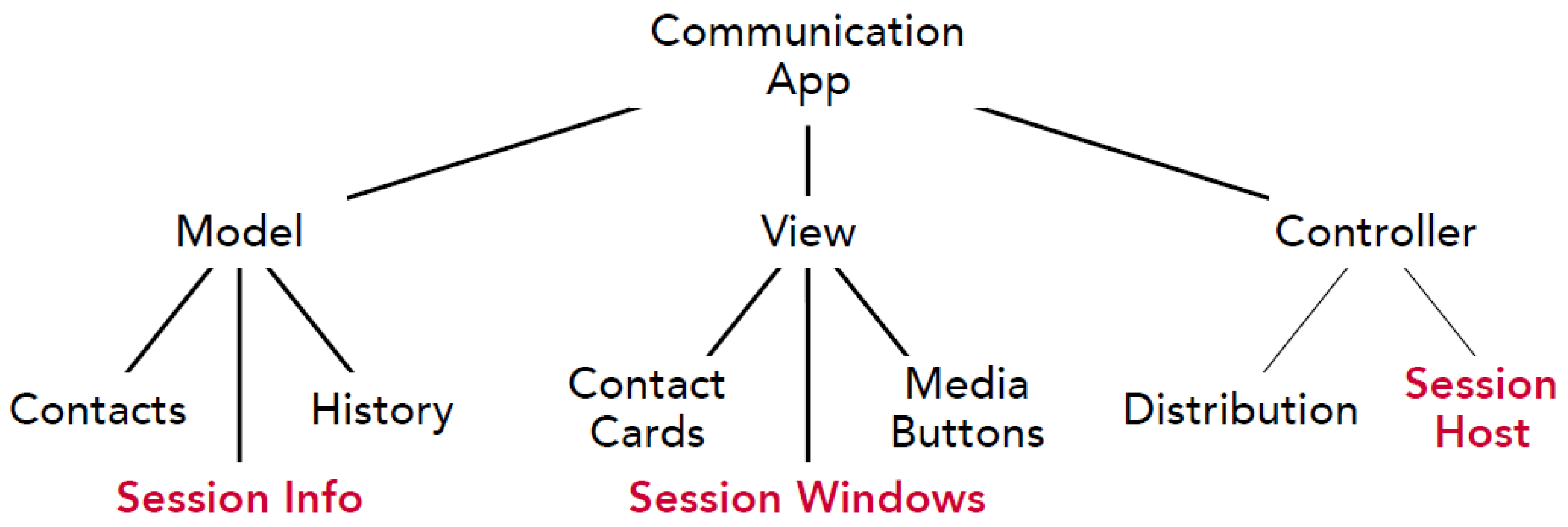


- Lab Version
  - Cards show participants in a communication session
  - Box encloses participant cards

- Production Version
  - Cards show participants
  - Set up a communication session by dragging cards into the spotlight

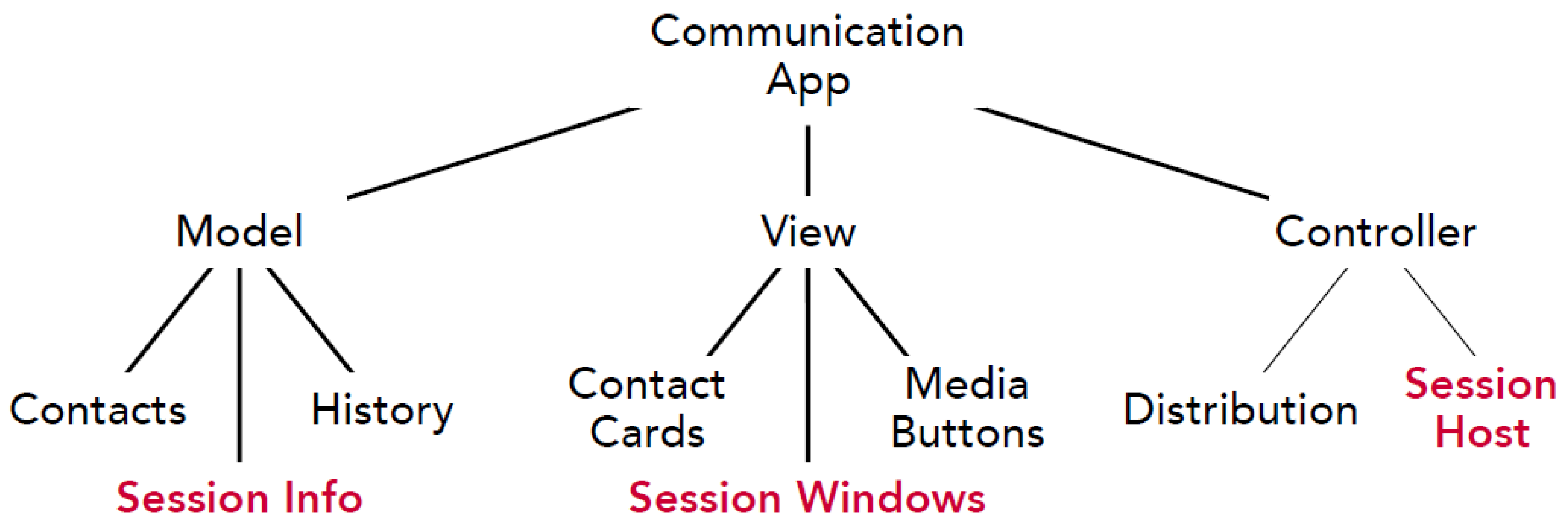


Application of Model View Controller



- Is this a good design? Should the session related modules be grouped together?

Application of Model View Controller



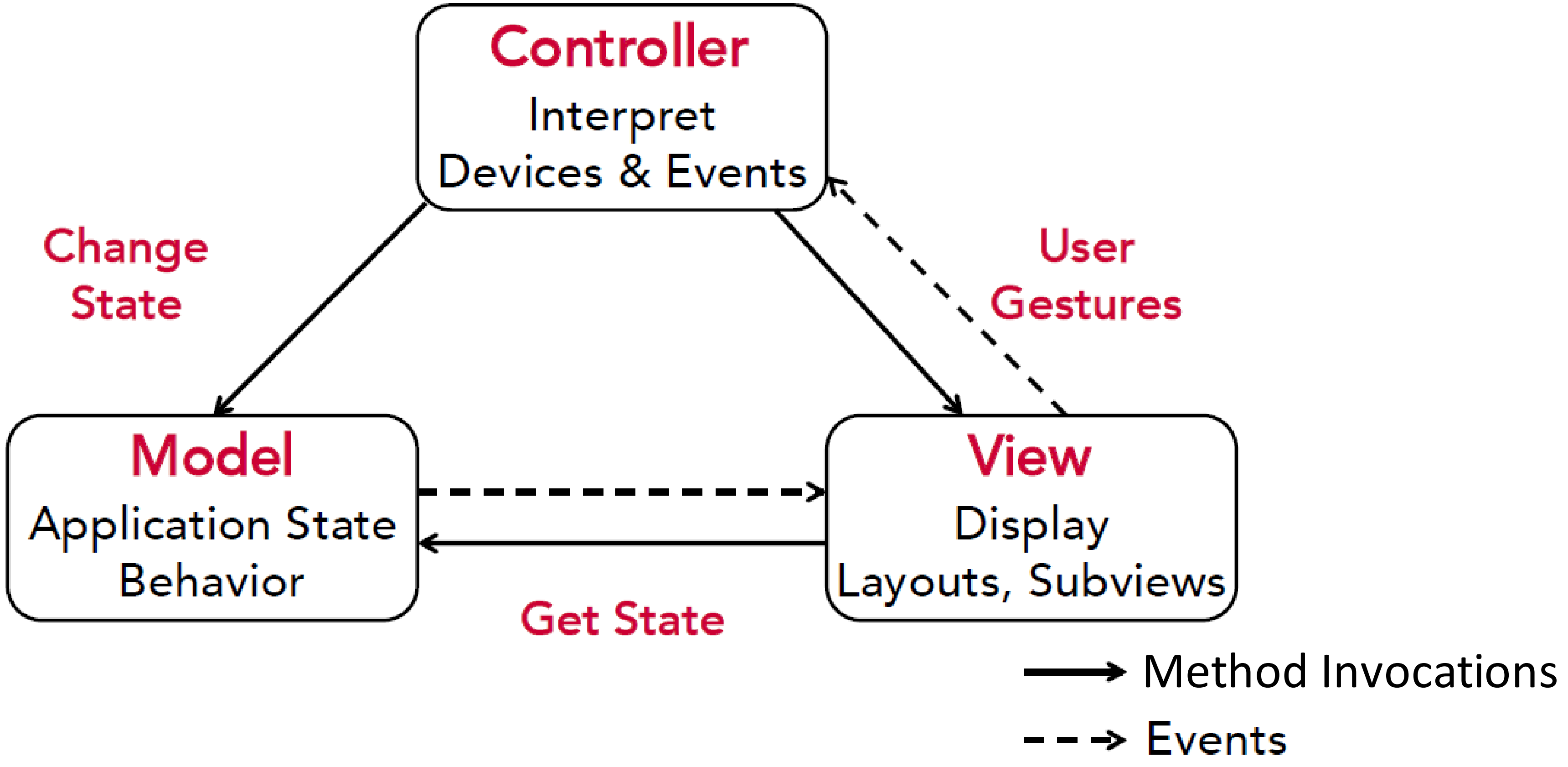
How to track current sessions, participants, the modes they are using

How to present sessions: spotlight or box

Who to add, who to drop

- No! Different design decisions

Model View Controller



Model-View-Presenter

- Taligent's (IBM) take on MVC for Java and C++
- The approach
  - Breaks the problem into two questions
    - "How do I manage my data?"
    - "How does the user interact with my data?"
  - Refines each question to further decompose the problem

Model-View-Presenter: Data Management

Refinement of "How do I manage my data?"

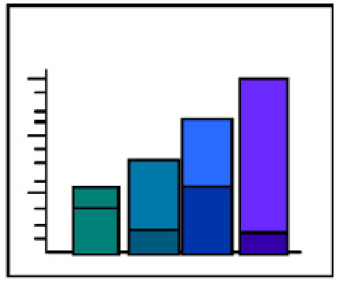
Model

lArray:

37, 23, 51, 18
7, 41, 47, 104

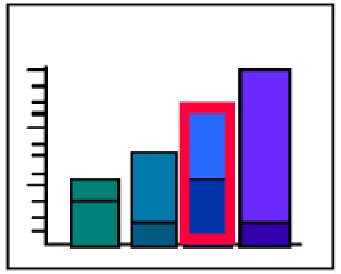
"What is my data?"

View



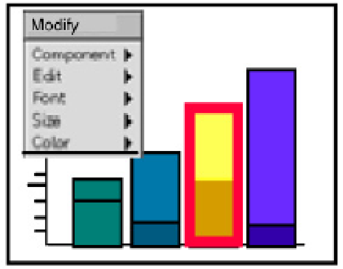
"How do I display my data?"

Selections



"How do I specify my data?"

Commands



"How do I change my data?"

Model-View-Presenter: User Interface

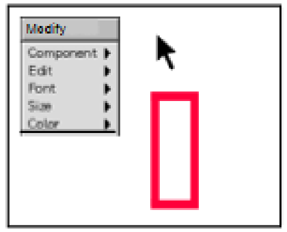
Refinement of "How does the user interact with my data?"

- How do I display my data?
  - Role of the View
- How do events map into changes in my data?
  - Relate user gestures to changes in the data (Interactor component)
- How do I put it all together?
  - Generalizes the Controller; now called the Presenter
  - The Presenter interprets events and gestures initiated by the user and provides logic to map them onto the appropriate command

Model-View-Presenter: User Interface

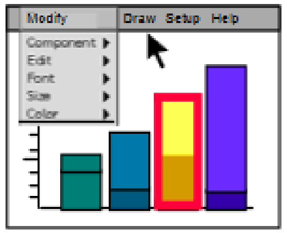
Refinement of "How does the user interact with my data?"

Interactors



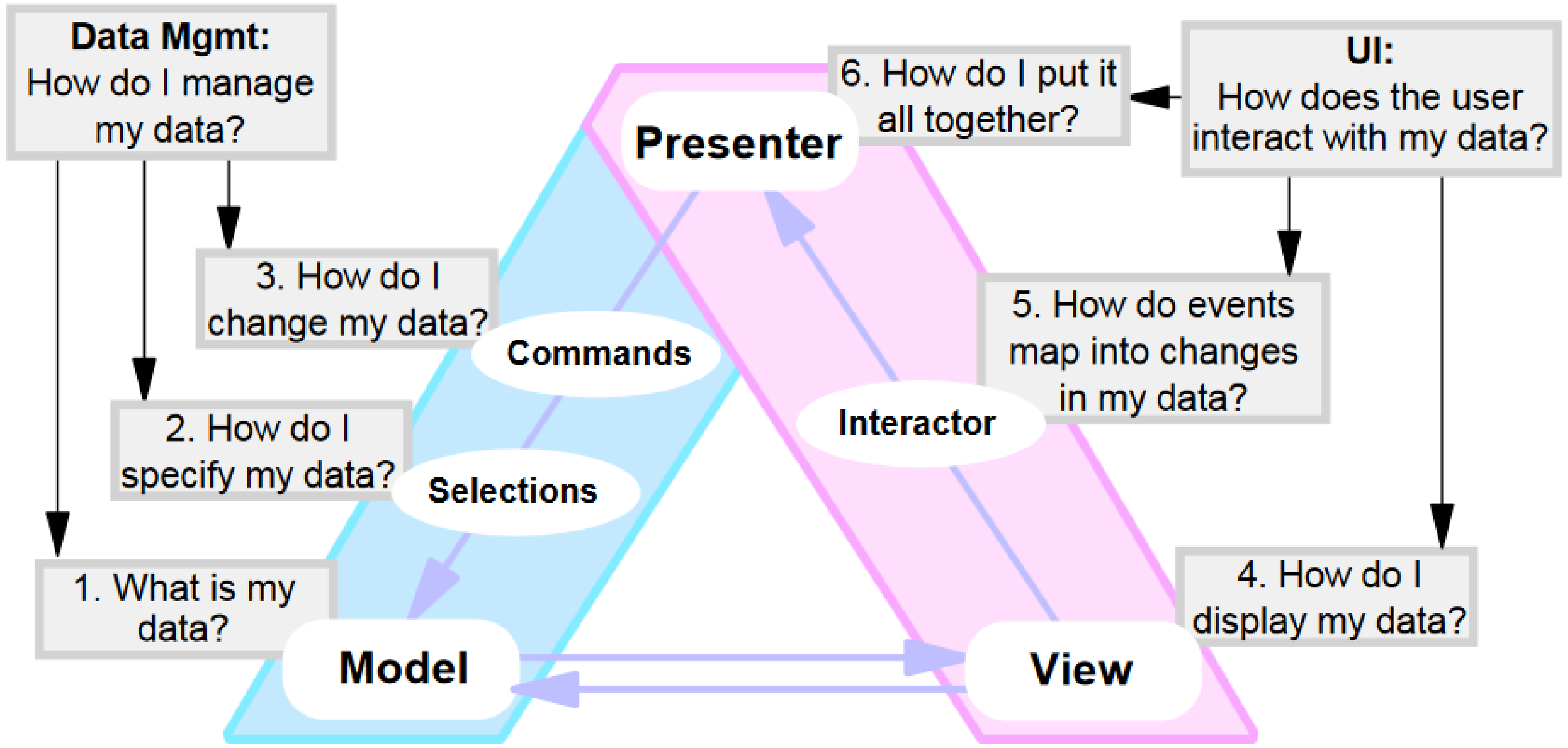
"How do events map onto changes to my data?"

Presenter



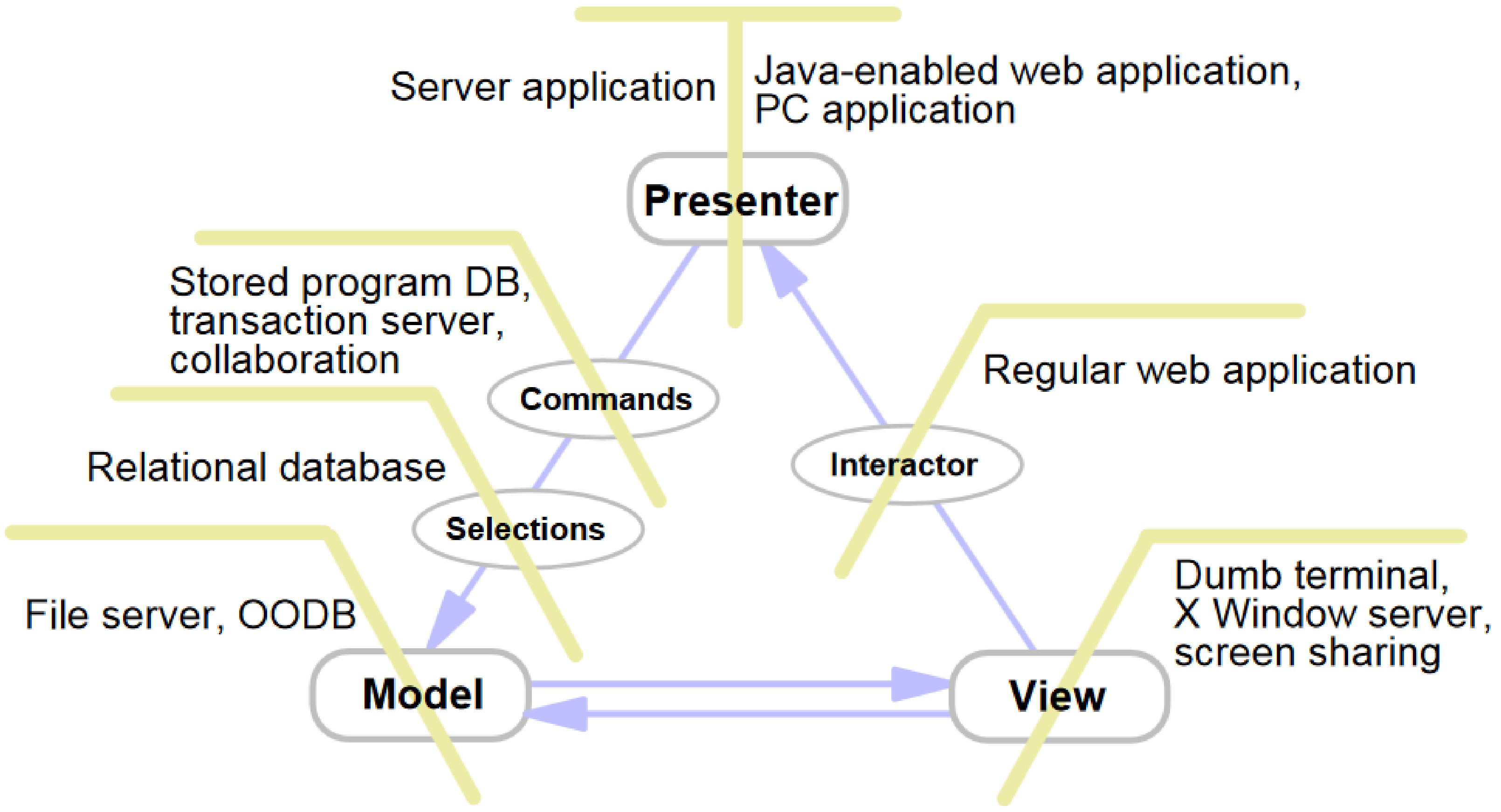
"How do I put it all together?"

Model-View-Presenter: Summary


Data Mgmt:
How do I manage my data?
6. How do I put it all together?
UI:
How does the user interact with my data?
Presenter
3. How do I change my data?
Commands
5. How do events map into changes in my data?
Interactor
2. How do I specify my data?
Selections
1. What is my data?
Model
View
4. How do I display my data?

Model-View-Presenter: Distributed Applications

"How do I partition my application between client and server?"



Server application
Java-enabled web application, PC application
Presenter
Stored program DB, transaction server, collaboration
Commands
Relational database
Selections
File server, OODB
Regular web application
Interactor
Dumb terminal, X Window server, screen sharing
Model
View

Layer Pattern

- Core of a solution
  - Group modules into sets (layers)
  - Modules interact through their interfaces
  - Modules in each layer use modules in the same layer or the layer immediately below (strict layering)
- Uses Relation
  - If A uses B, B must be present & satisfy its spec for A to satisfy its spec
  - But lower layers have no dependency on upper layers
- Traditional Example
  - TCP/IP and the IP stack
  - Virtual machines

Principles of Layered Architecture

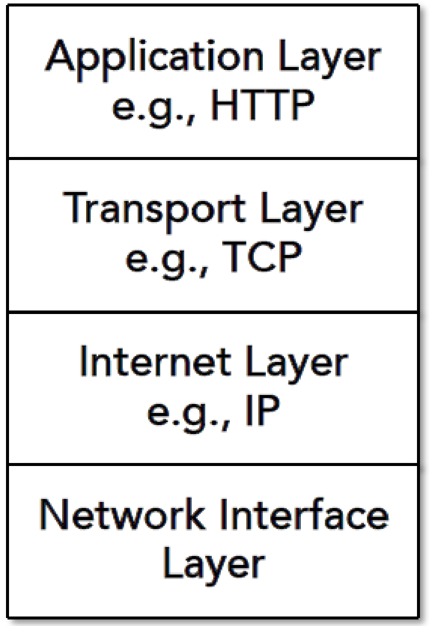
- Abstraction
  - Abstract away details of the lower layers
- Encapsulation
  - Do not expose implementation details at layer boundaries
- Clearly defined functional layers
  - Separation between functionality in each layer is clear
- High cohesion
  - Well-defined responsibility boundaries for each layer
- Reusable
  - Lower layers have no dependency on upper layers and are reusable
- Loose coupling
  - Communication between layers is always through well-defined interfaces

Layering: TCP/IP

- The layering of TCP and IP was not in the original proposal
- "These changes in the Internet design arose through the repeated pattern of implementation and testing that occurred before the standards were set."
  - – Clark (1988)

Internet Protocol Stack: TCP/IP

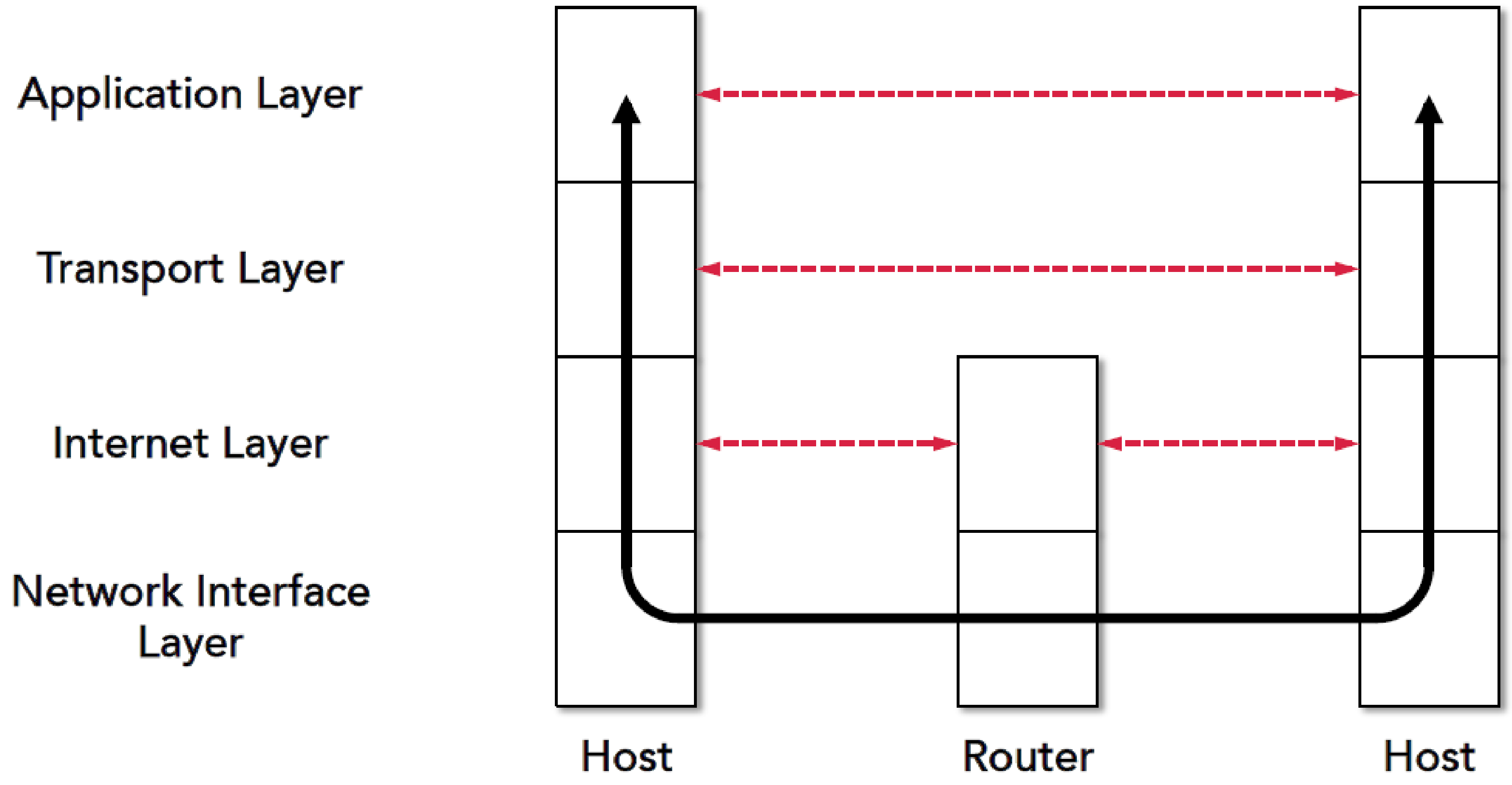
- Key Architectural Decision
  - The intelligence is in the Application Layer
  - The routing and transport of packets through the Internet are in the layers below



- Application programs pass data (packets) to the Transport layer for delivery
- TCP provides reliable app-to-app communication. It passes packets through the Internet Layer
- IP provides best effort packet delivery. It uses a routing algorithm to select the next hop

TCP/IP Protocol Stack

- Each layer experiences peer-to-peer service (red dashed lines)
- The actual flow of packets is through the layers (black solid lines)



Why TCP and IP Were Layered

They were originally a single protocol

- Initial concept of TCP (1973)
  - General enough that it could provide any needed type of service
- But …
  - The cross-Internet debugger, XNET, needed access to whatever got through in times of stress or failure – better something (best effort) than nothing (insisting on every byte being delivered in order).
  - For real-time delivery of digitized speech for teleconferencing the need was to minimize delay. Some dropped packets could be smoothed over. The biggest source of delay was the reliable delivery mechanism.
- So, TCP and IP were separated
  - TCP supports reliable delivery, but packets might be ***delayed***
  - IP supports speedy delivery, but packets might be ***dropped***

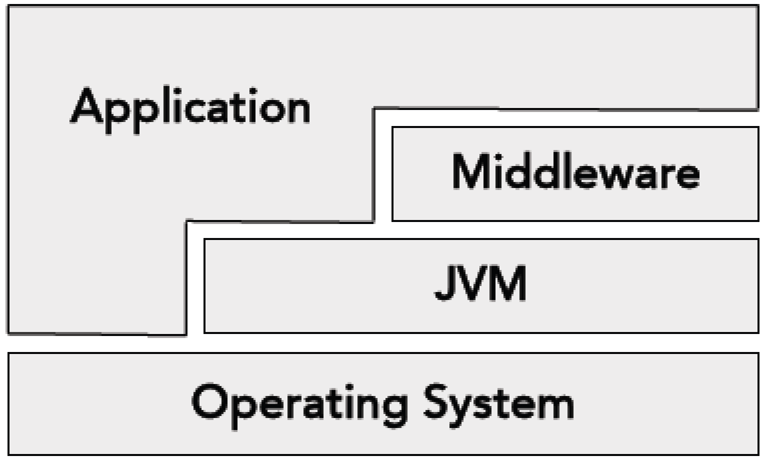
Layered Architectures: Assessment

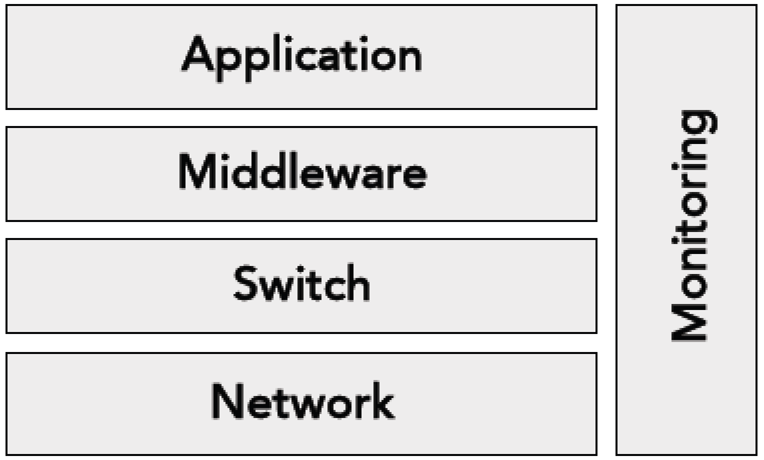
Strict layering has proven to be enormously flexible for Internet protocols

- Benefits:
  - Support designs with increasing levels of abstraction (thereby partitioning a complex problem)
  - Support enhancement without changing the other layers (e.g. new application layer protocols above TCP/IP)
- Tradeoff:
  - Performance – adding headers going down the Internet protocol stack at the source and then stripping them going up the stack at the destination adds overhead
  - Strict layering favors clarity for user over ease for designer. e.g. the designer of a new protocol may need to separate it into layers to fit the IP suite

Variants of Layering

- Be explicit with the uses structure when drawing layer diagrams, as in the explanations below these diagrams.





Layer Bridging, where a module at one layer can use modules in any layer below it

Sidecar, where the vertical Monitoring modules can access any layer

Dataflow: Pipe and Filter

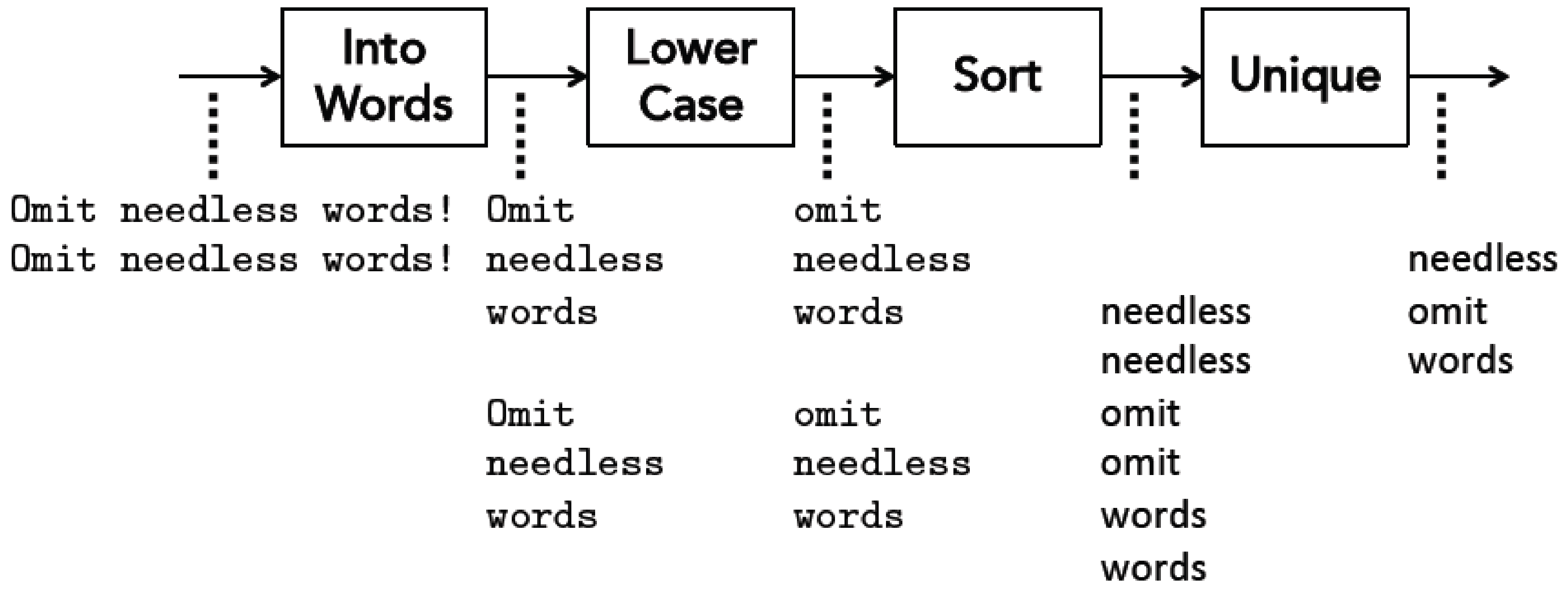
- “We should have some ways of coupling programs like a garden hose – screw in another segment when it becomes necessary to massage data in another way.”
  - *– Doug McIlroy, October 11, 1964*

Pipe-and-Filter Pattern

- Core of a solution
  - Assemble modules into a pipeline, where the output of one module becomes the input to the next
  - Modules are independent and unaware of each other
  - Not limited to linear pipelines
  - Elements of a pipeline are co-routines; they can execute in parallel, constrained only by availability of input
- Traditional Examples
  - Unix pipelines (which are linear)
  - Exception: tee command copies input to two output streams

Unix Pipeline

> tr -C a-zA-Z '\n' | tr A-Z a-z | sort | uniq



Google Dataflow

Introduced June 2014

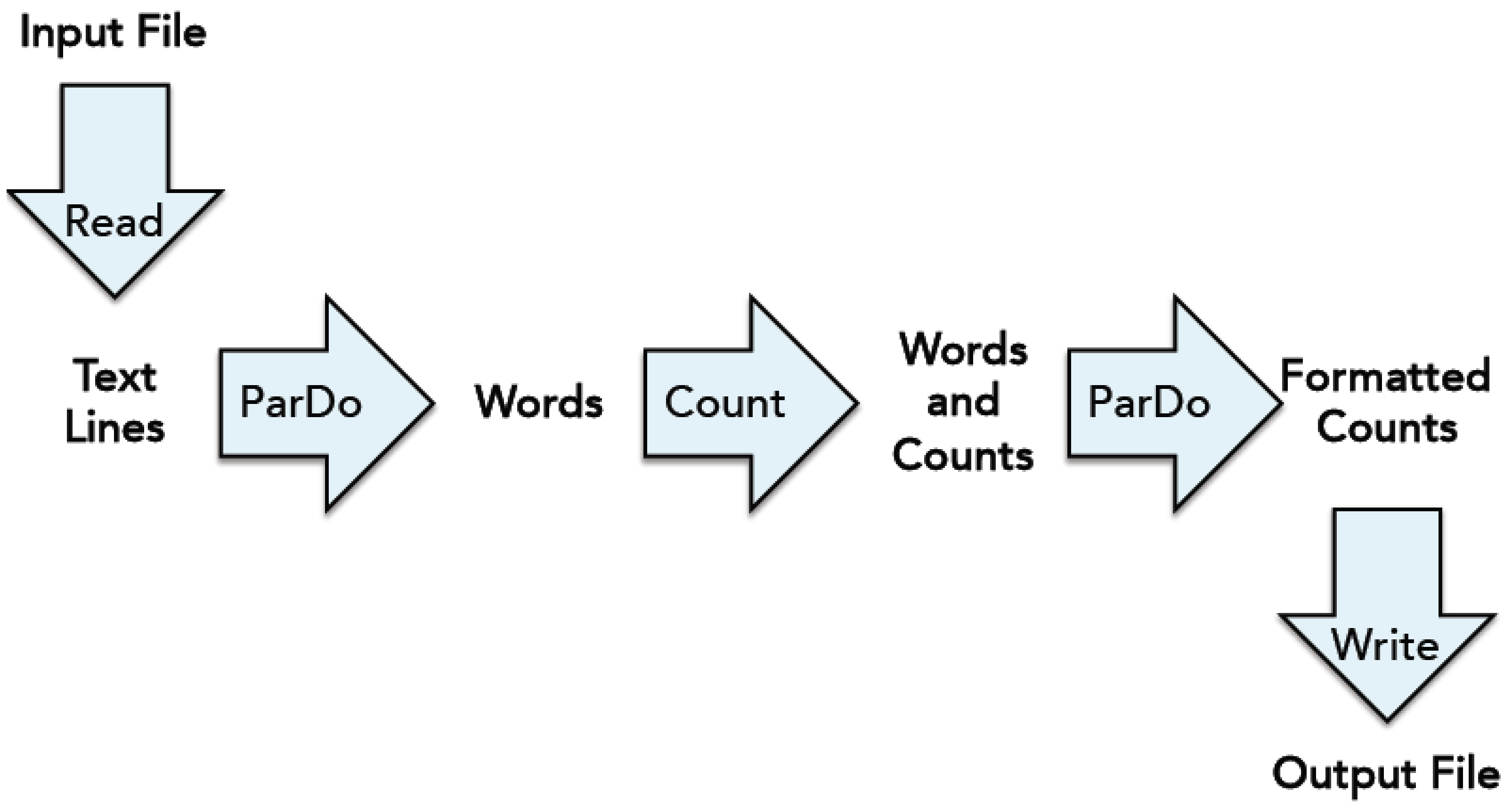
- Ready for unbounded, unordered, global-scale datasets
  - e.g. Web logs, mobile usage statistics, and sensor networks
- Propose a fundamental shift
  - Stop trying to groom unbounded datasets into finite pools that eventually become complete
  - Instead, assume that we will never know if/when we have seen all of our data
- Dataflow replaces Map-Reduce

Google Dataflow
Sample Application

- Streaming video provider wants to display video ads
  - Bill advertisers for the amount of advertising watched
  - Platform supports online and offline views for content and ads
- Requirements
  - How often and for how long are videos being watched, with which content/ads, and by which demographic groups?
  - Want all of this information as quickly as possible, to adjust budgets and bids, change targeting, tweak campaigns, and plan future directions in as close to real time as possible

Google Dataflow: Word-Count Pipeline

Tutorial example from the website



Google Dataflow

Dealing with unbounded streams of data

- Text lines to words can be done on the fly

```
Omit needless words!     Omit
                         needless
                         Words
```

- How do you count words in an unbounded stream
  - There is no end to the stream

Google Dataflow

Dealing with unbounded streams of data

- Text lines to words can be done on the fly

```
Omit needless words!        Omit
                            needless
                            Words
```

- How do you count words in an unbounded stream?
  - There is no end to the stream!
- Settle for windowing
  - Fixed windows
  - Sliding windows
  - Session windows

Design and Architecture Summary

- What is Software Architecture?
  - – Literally hundreds of definitions, e.g. see SEI's website
- An architecture is
  - – a set of structures (each structure answers one or more questions)
    - To structure is to partition the whole into parts
    - and specify the relations among the parts
  - – that satisfy the requirements
    - functional requirements; end-user features, …
    - other engineering requirements: 'ilities
    - non-functional requirements: legal, regulatory, environmental
- A good architecture is elegant
  - – The system is easy to modify – if you can modify it, you can understand it

Design Concepts

Fred Brooks, *The Design of Design*

- Great designs have conceptual integrity
  - unity, economy, integrity
  - You can develop a mental model about the design that lets you make predictions about how it will operate; if they predictions come true, we often take delight in the use of the design
- Gives a design team something to talk about
  - Unity of concept is essential and it can only be achieved though lots of conversation; what’s in scope, what’s out? what concepts does this design make use of? how do they relate?
- Example, "everything's a file" in Unix systems

Kinds of Design

- Routine
  - Standard models exist and are routinely (re)used
  - Short suspension bridges
  - Client-server
- Adaptive
  - Standard models can be adapted to fit the need (Patterns)
  - Tandem bicycle
  - In-memory database
- Original
  - Take a deep breath!
  - First skyscraper
  - First PC

Conclusion

- Design is a process of discovery and satisficing*
  - Investigate design decisions, but be prepared to backtrack and seek alternatives
  - Leave a trail for others to follow (issues and resolutions)
  - Satisfice: the space is too big for complete exploration
- Use standard models when feasible
  - Adapt as necessary
  - The rewards of innovation come with risk

* satisficing: to act in such a way as to satisfy the minimum requirements for achieving a particular result

Backup

What is Architecture?

- "The art or science of building; esp. the art or practice of designing and building edifices for human use, taking both aesthetic and practical factors into account."
  - – The Shorter Oxford English Dictionary, Fifth Edition, 2002
  - – Merriam Webster Online Dictionary
- "In wider use, the term 'architecture' always means 'unchanging deep structure.'"
  - – Stewart Brand, *How Buildings Learn*