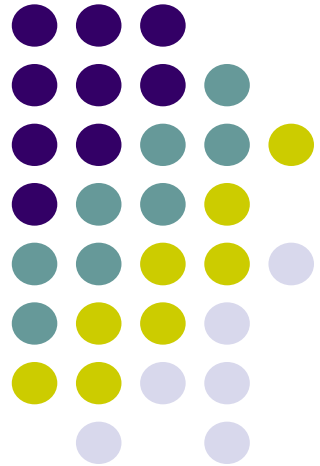
A short introduction to Software Architecture



David Garlan
Carnegie Mellon University



© Garlan & Schmerl

Examples of Software Architecture

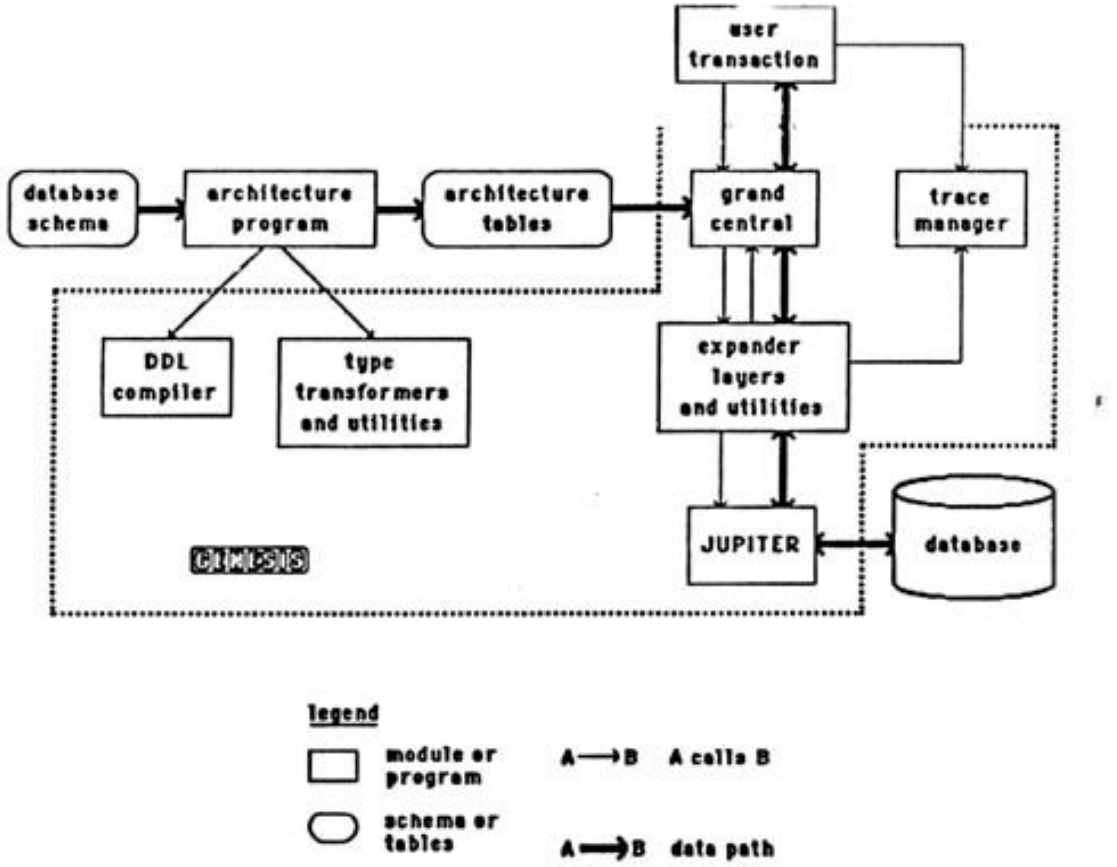


Figure 3.1 The Configuration of the GENESIS Prototype

Genesis: A Reconfiguration Database Management System, D. S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, Department of Computer Sciences, University of Texas at Austin,

© Garlan & Schmerl

More Examples

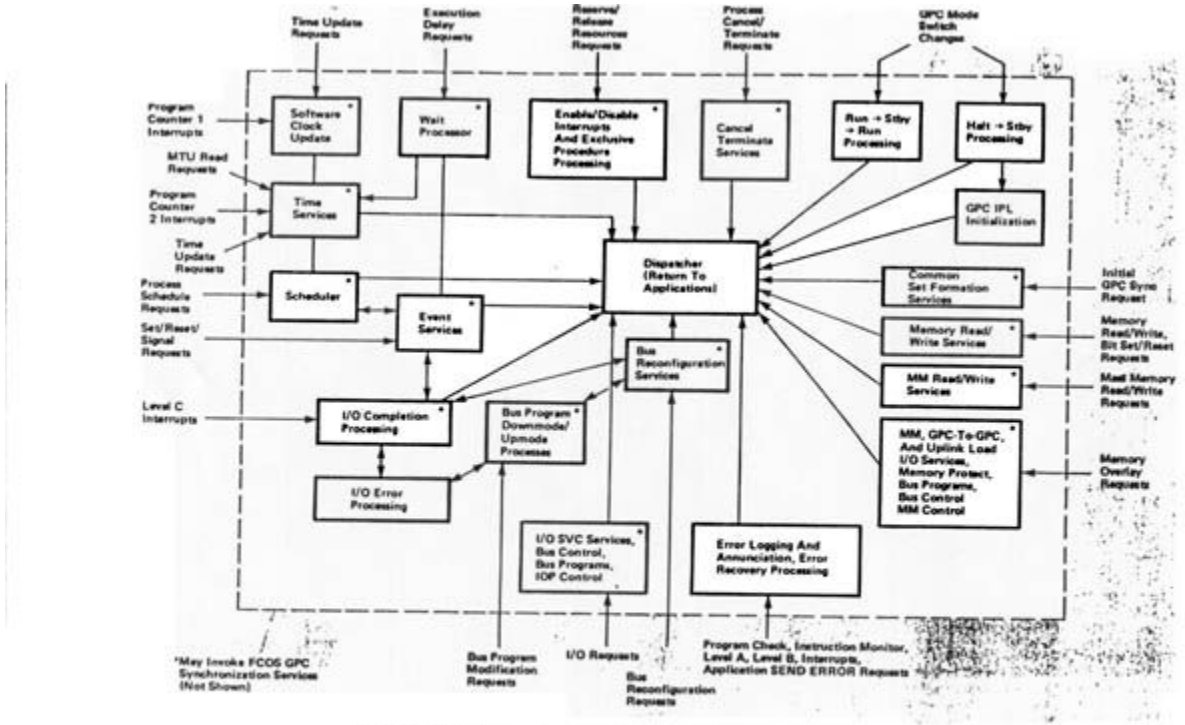


FIGURE 7. Flight Computer Operating System (The FCOS dispatcher coordinates and controls all work performed by the on-board computers.)

Communications of the ACM, "Architecture of the Space Shuttle Primary Avionics Software Systems," Gene D. Carlow, September 1984, Vol. 27, No. 9, P. 933

© Garlan & Schmerl

More Examples

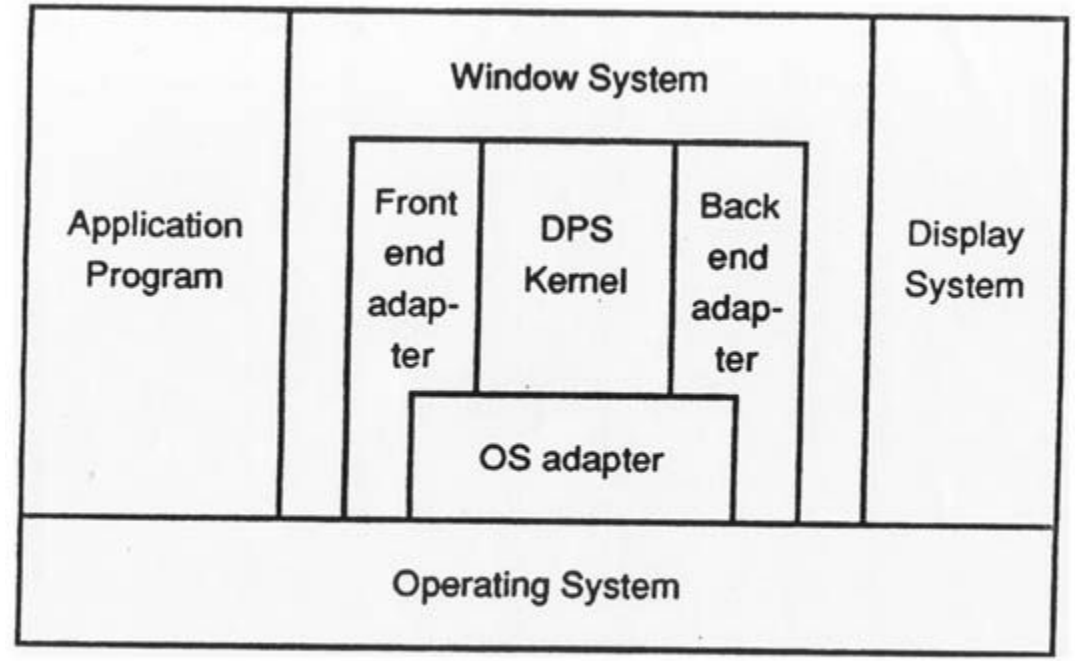


Figure 2. Display PostScript interpreter components.

An Overview of the DISPLAY POSTSCRIPT™ System, Adobe Systems Incorporated, March 16, 1988, P. 10

© Garlan & Schmerl

More Examples

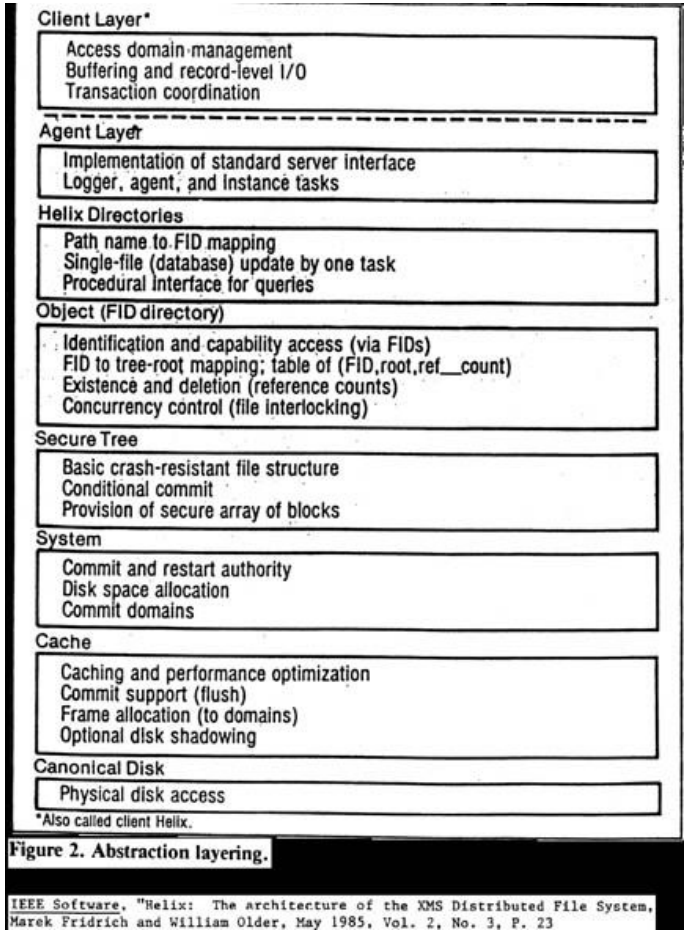
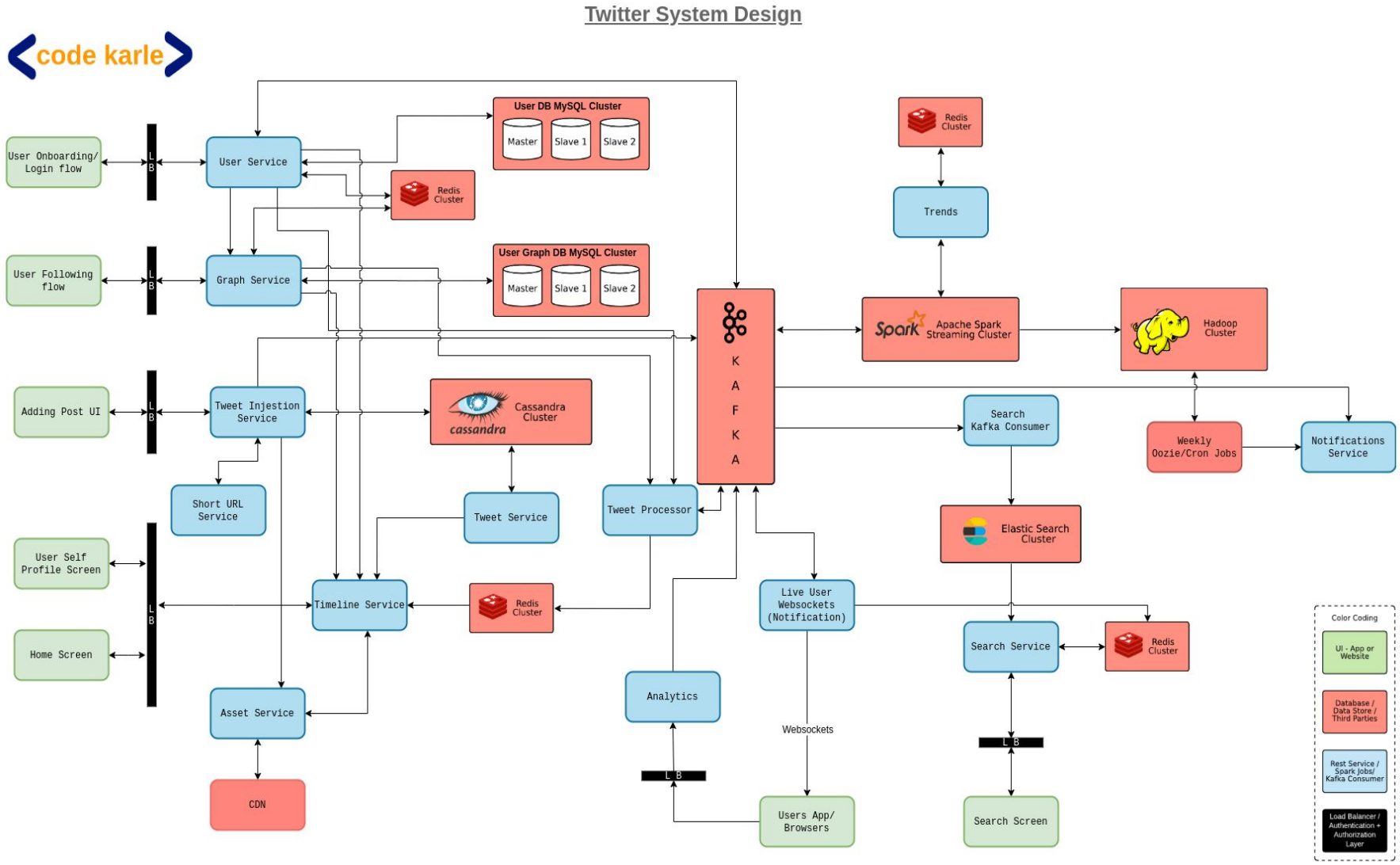


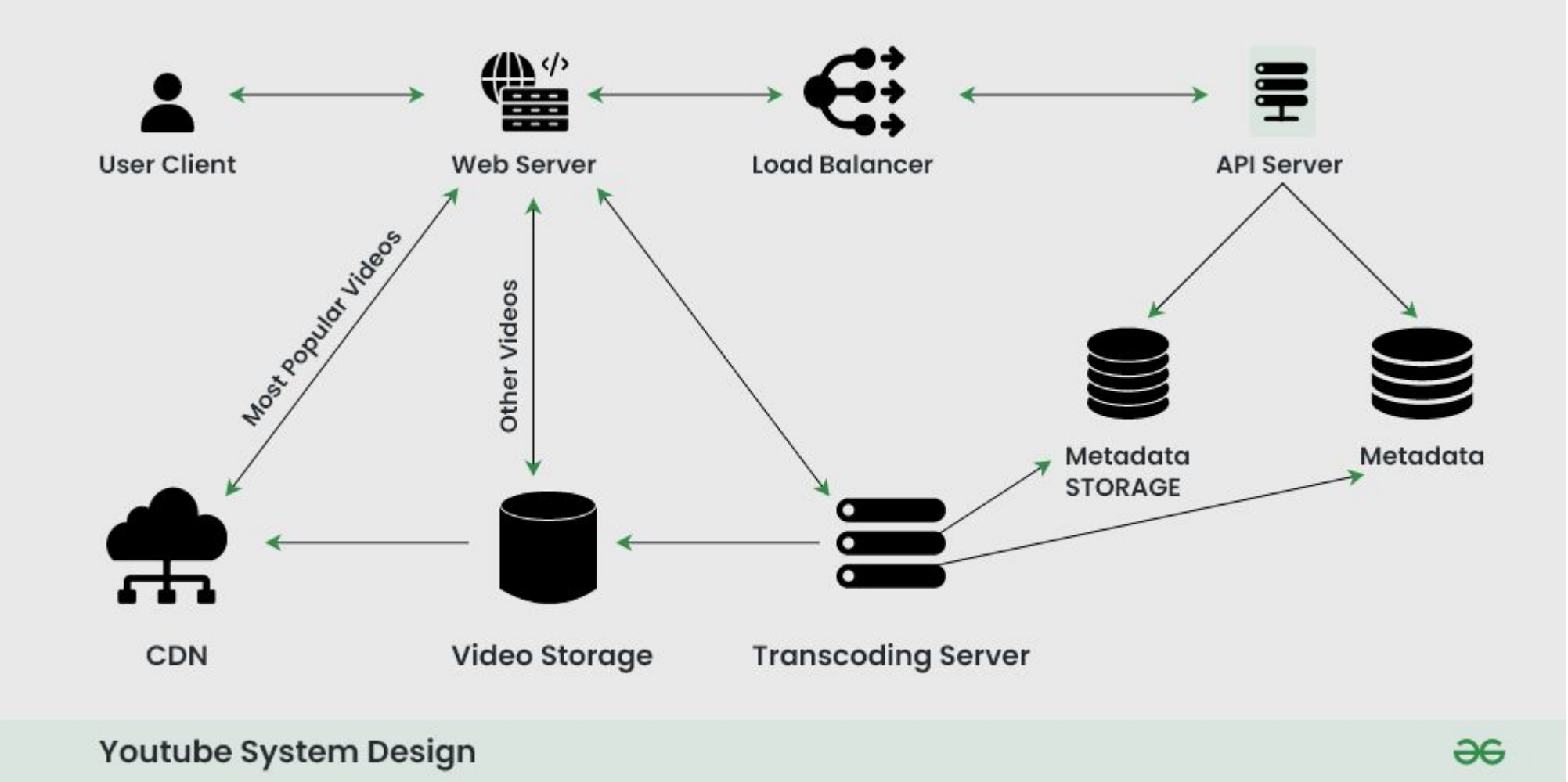
Figure 2. Abstraction layering.

IEEE Software, "Helix: The architecture of the XMS Distributed File System, Marek Fridrich and William Older, May 1985, Vol. 2, No. 3, P. 23

More Examples



More Examples



More Examples

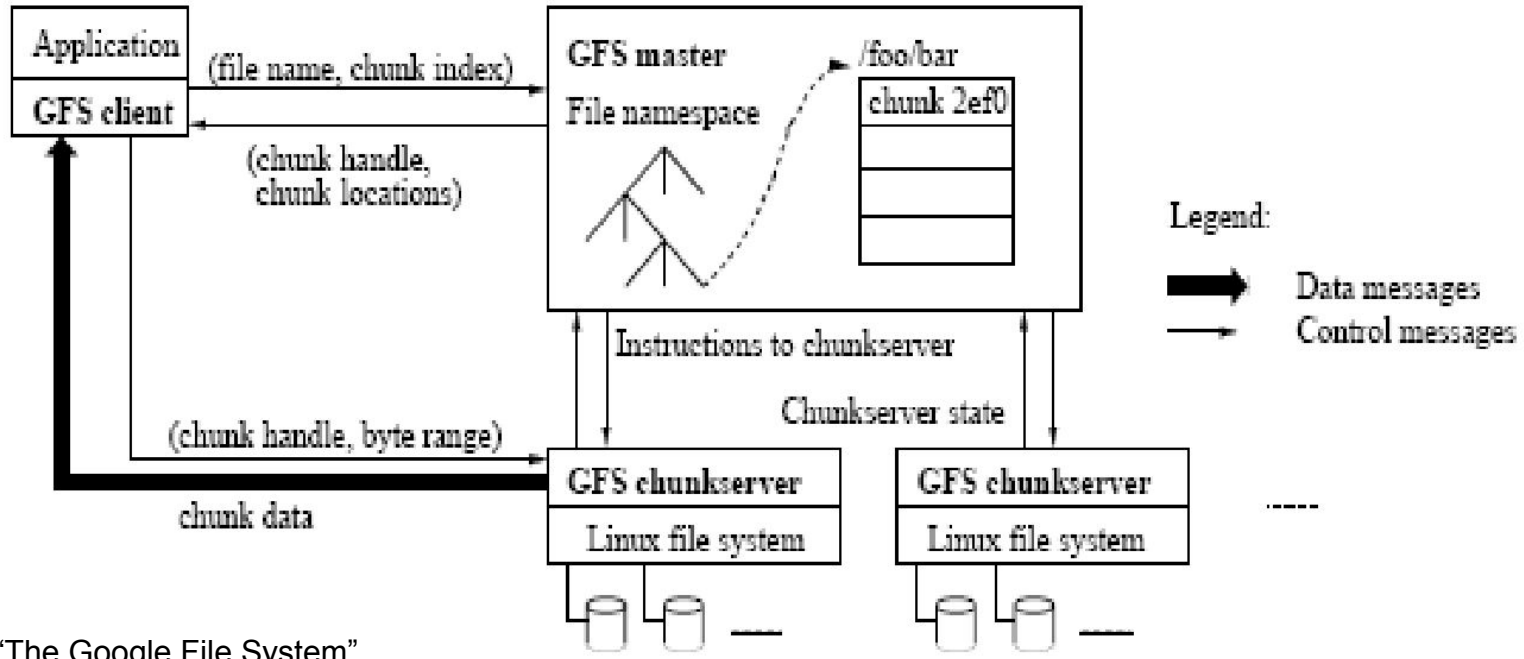
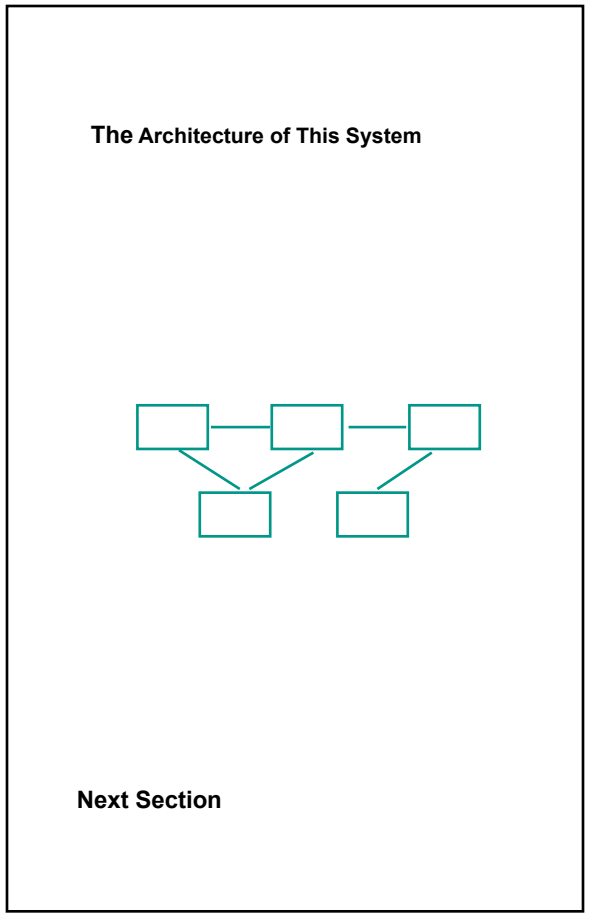


Figure 1: GFS Architecture

Source: "The Google File System" Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. SOSP 2003.

© Garlan & Schmerl

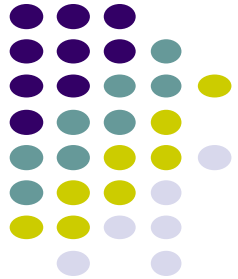
Descriptions of Software Architecture



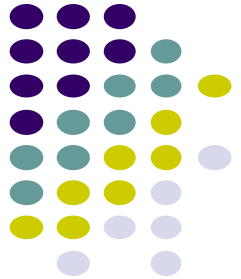
- Descriptions of software systems often include a description of “the architecture of this system”.
- Usually this consists of a box-and-line diagram, as in the previous examples, together with some descriptive prose.

© Garlan & Schmerl

Such Descriptions are Useful



- They provide a high-level, “birds-eye” view of the system.
  - abstracting implementation/coding details
  - focusing on the essential system structures and their relationships
- … and thereby (hopefully) support understanding
  - what are the system’s principal components
  - how information flows through the system
  - where is data stored persistently
  - what parts of a system might need to be modified to add some new feature



But …!!!

Such diagrams usually leave many questions unanswered:

- Is this a “good” architecture?
  - What do we even mean by “good”?
- What significant design decisions were made?
  - Did those decisions involve tradeoffs between desirable properties?
- Is this architecture similar to others used in industry?
  - … that I might be familiar with? that use common frameworks? that solve design challenges in a similar way?
- What changes are going to be difficult/easy to make?
  - Downstream flexibility and ease of enhancement?
- How do I know if I am interpreting the diagram correctly?
  - What do those lines/arrows actually mean?

Requirements, Reviews

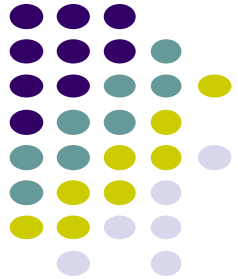
Trade-off Analysis

Styles, Patterns, Platforms,Tactics

Architecture Evolution

Modeling, Documentation, Presentation

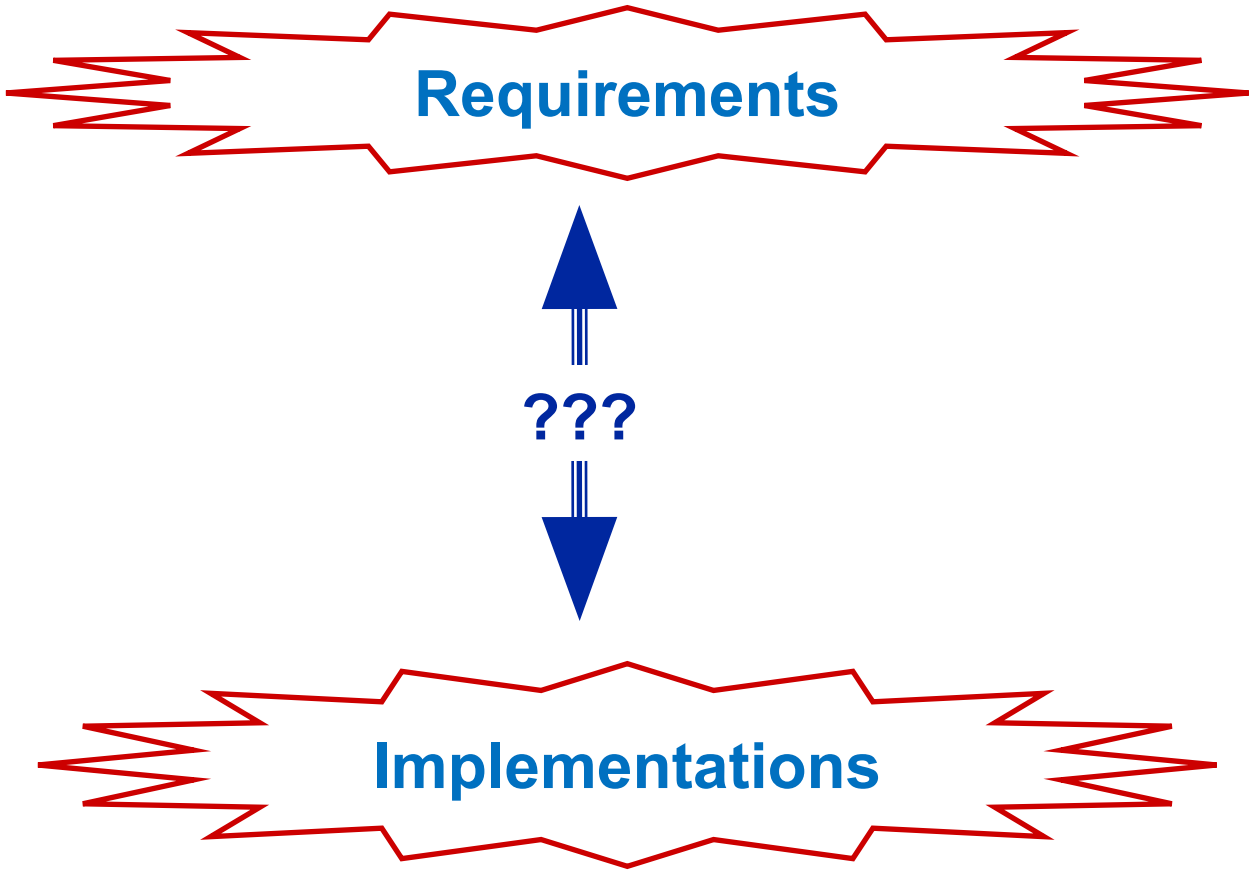
The Challenge



- Learn how to practice Software Architecture design as an *engineering discipline*
  - from ad-hoc decisions/representations to solid principles/techniques
- Think *"architecturally"*
  - design at an architectural level of abstraction
  - determine whether the system will satisfy critical *systemic* requirements
  - recognize when we must make tradeoffs and have a rational basis for making them
  - reuse common architecture patterns, styles, and expertise
  - reduce costs through product-lines, platforms, frameworks

© Garlan & Schmerl

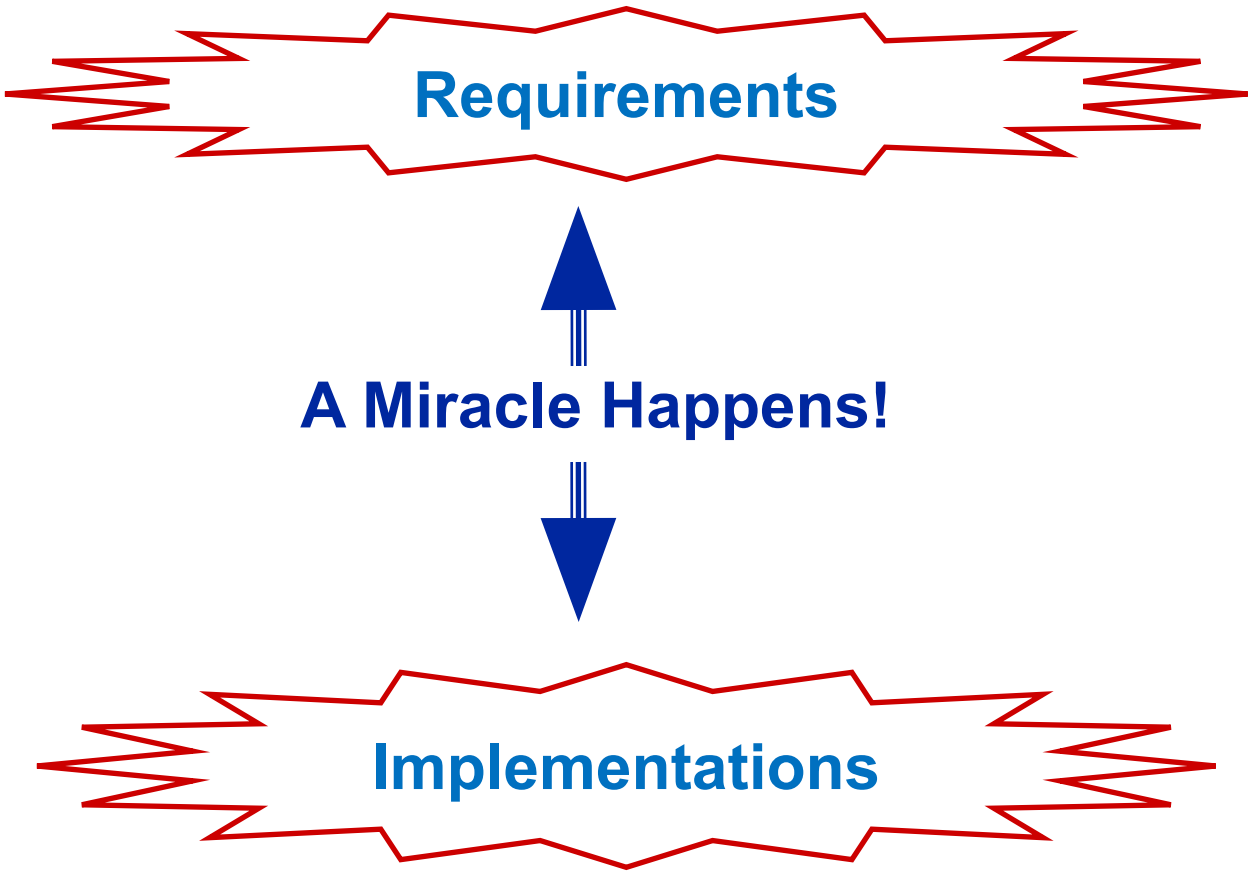
The Big Problem



How to bridge the gap between requirements and solutions?

© Garlan & Schmerl

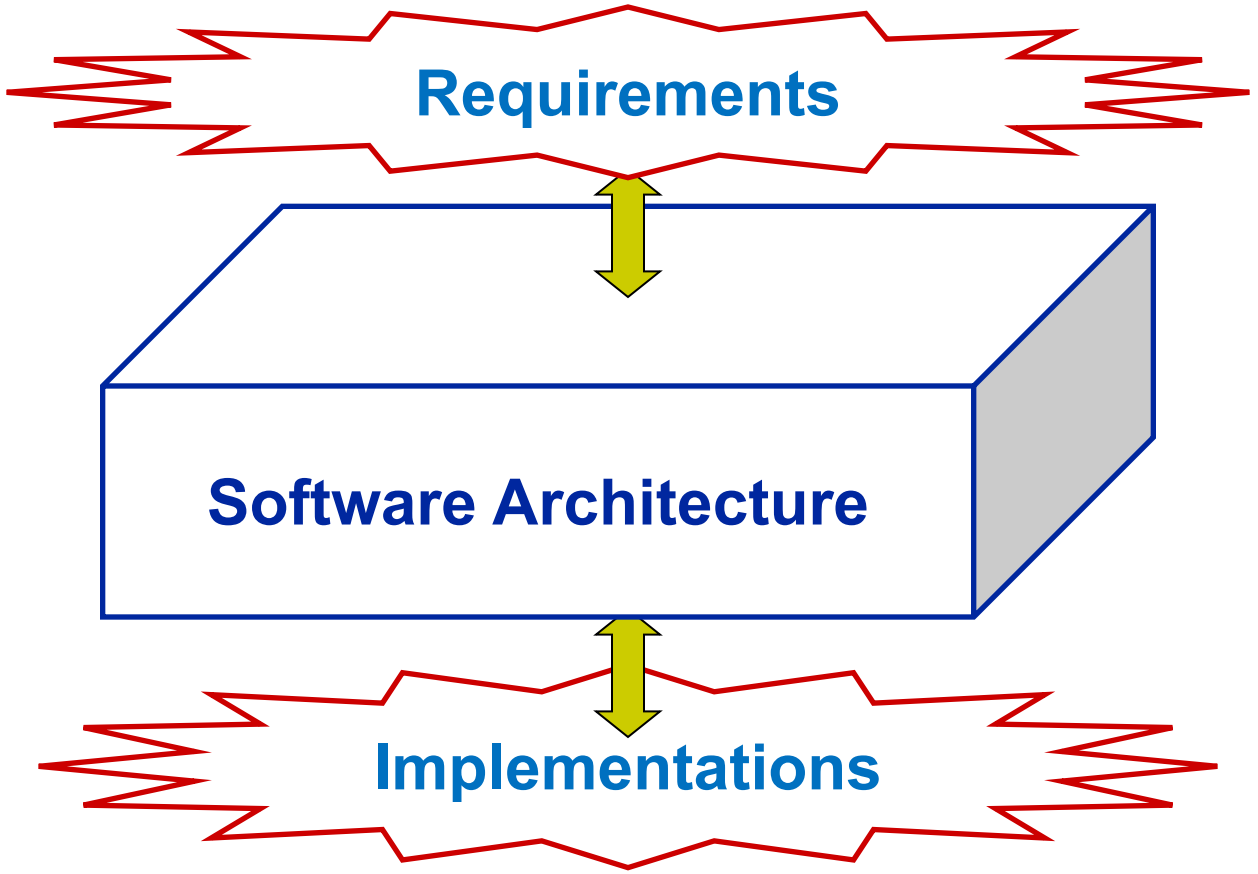
One Possible Answer



- We’re smart; let’s just code it!
- Lots of testing to gain confidence.
- Cross our fingers that we made good decisions and don’t have a lot of rework.

© Garlan & Schmerl

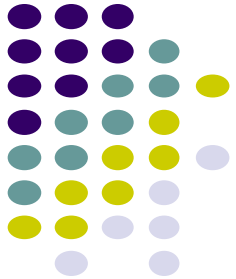
The Role of Software Architecture



- Expose critical system-level requirements.
- Understand key tradeoffs and make principled design decisions.
- Reuse well-founded design principles and techniques.
- Understand how implementation choices may constrain our design options.

© Garlan & Schmerl

What is Software Architecture?



- There are many definitions in the literature
  - The CMU Software Engineering Institute web site on software architecture lists a large number of them.*
- The definition we will use is this:**

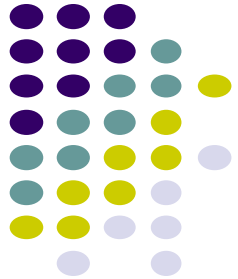
> The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them and properties of both.

* https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513807

** Documenting Software Architecture: Views and Beyond, 2nd Ed. Clements et al. 2011.

© Garlan & Schmerl

Issues Addressed by Software Architecture



- Gross decomposition of a system into parts
  - often using rich abstractions for *component interaction* (or system “glue”)
  - often using common design *patterns/styles*
- "Emergent” system properties
  - performance, throughput, latencies
  - reliability, security, fault tolerance, evolvability
- Rationale
  - justifying architectural decisions and tradeoffs
- Envelope of allowed change
  - “load-bearing walls” and constraints on implementation
  - choices left up to the developer

© Garlan & Schmerl

Example of Google File System

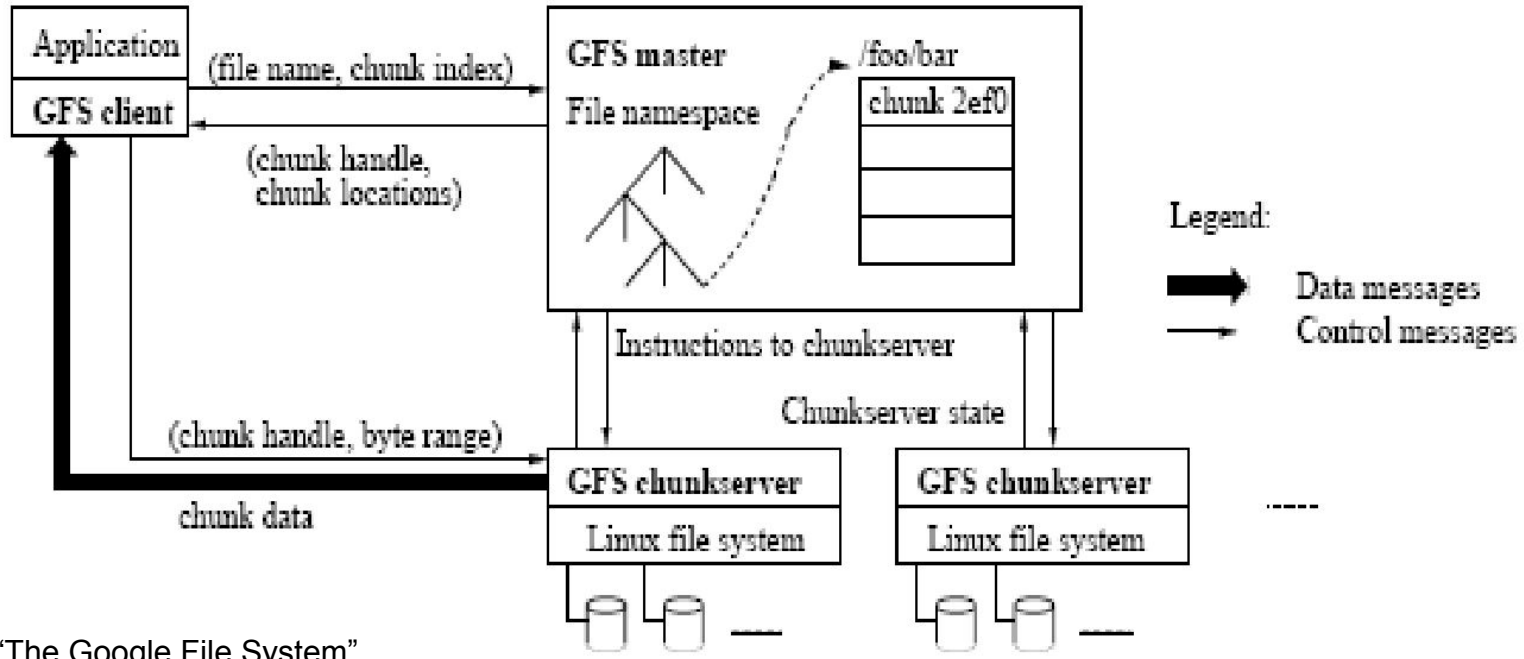
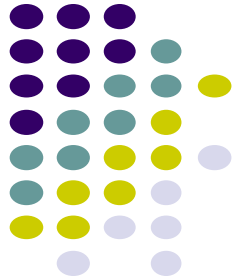


Figure 1: GFS Architecture

Source: "The Google File System" Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. SOSP 2003.

© Garlan & Schmerl

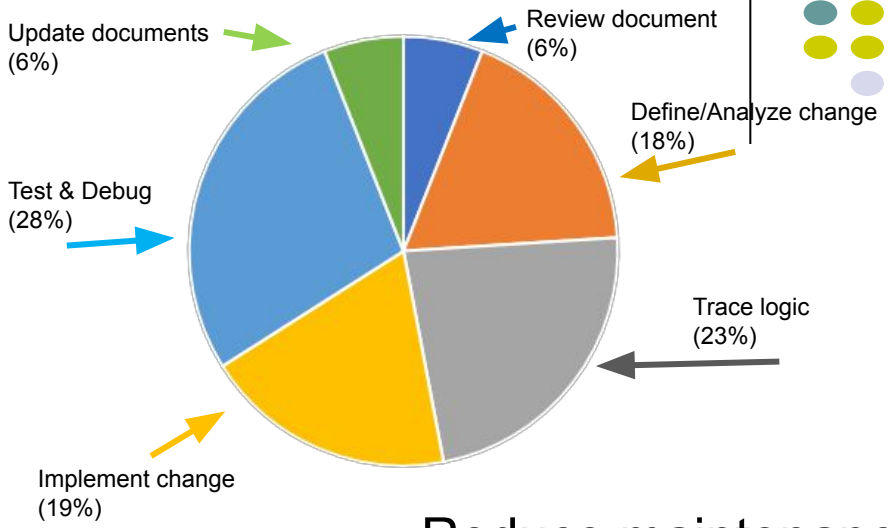
Why Software Engineers Care



- Improve the quality of the system.
  - Clarify requirements.
  - Make principled engineering decisions.
  - Early analysis of design flaws.
- Reduce development and maintenance costs.
  - Reuse of designs.
  - Avoid design errors.
  - Improve understandability.
  - Impact downstream system evolution.

© Garlan & Schmerl

Anticipated Benefits

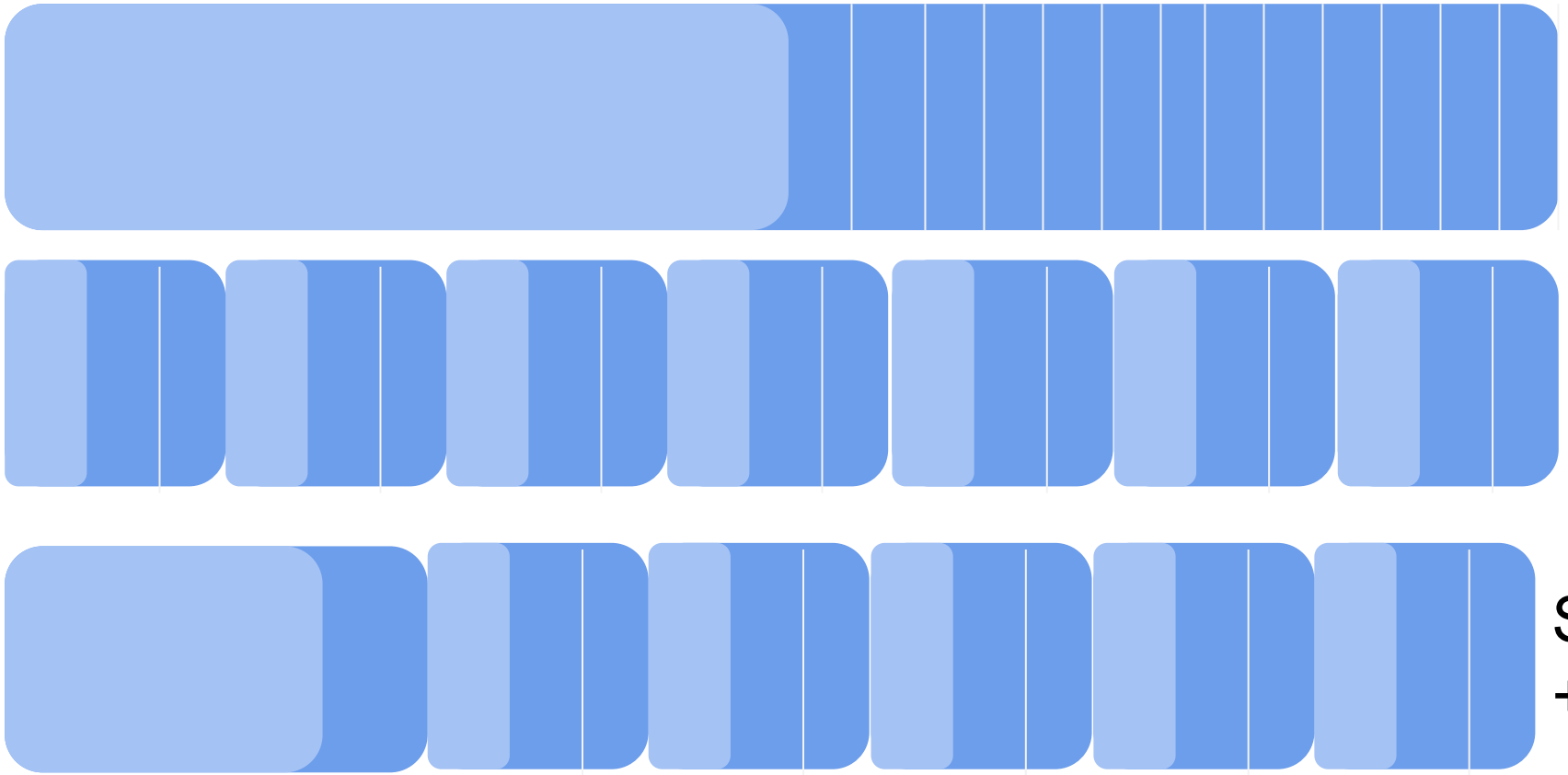


Reduce maintenance and enhancement costs, directly and indirectly

Requirements → Architecture → Design → Coding/Integration → Test/Accept → Maintenance

- Clarify intentions
- Make decisions and implications explicit
- Support system-level analysis

When Do We Do Architecting?



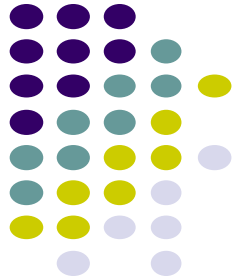
Big Upfront Design

Evolutionary Design

Some Upfront Design
+ Evolutionary Design

Simon Brown, “The lost art of software design” Devoxx Belgium 2022

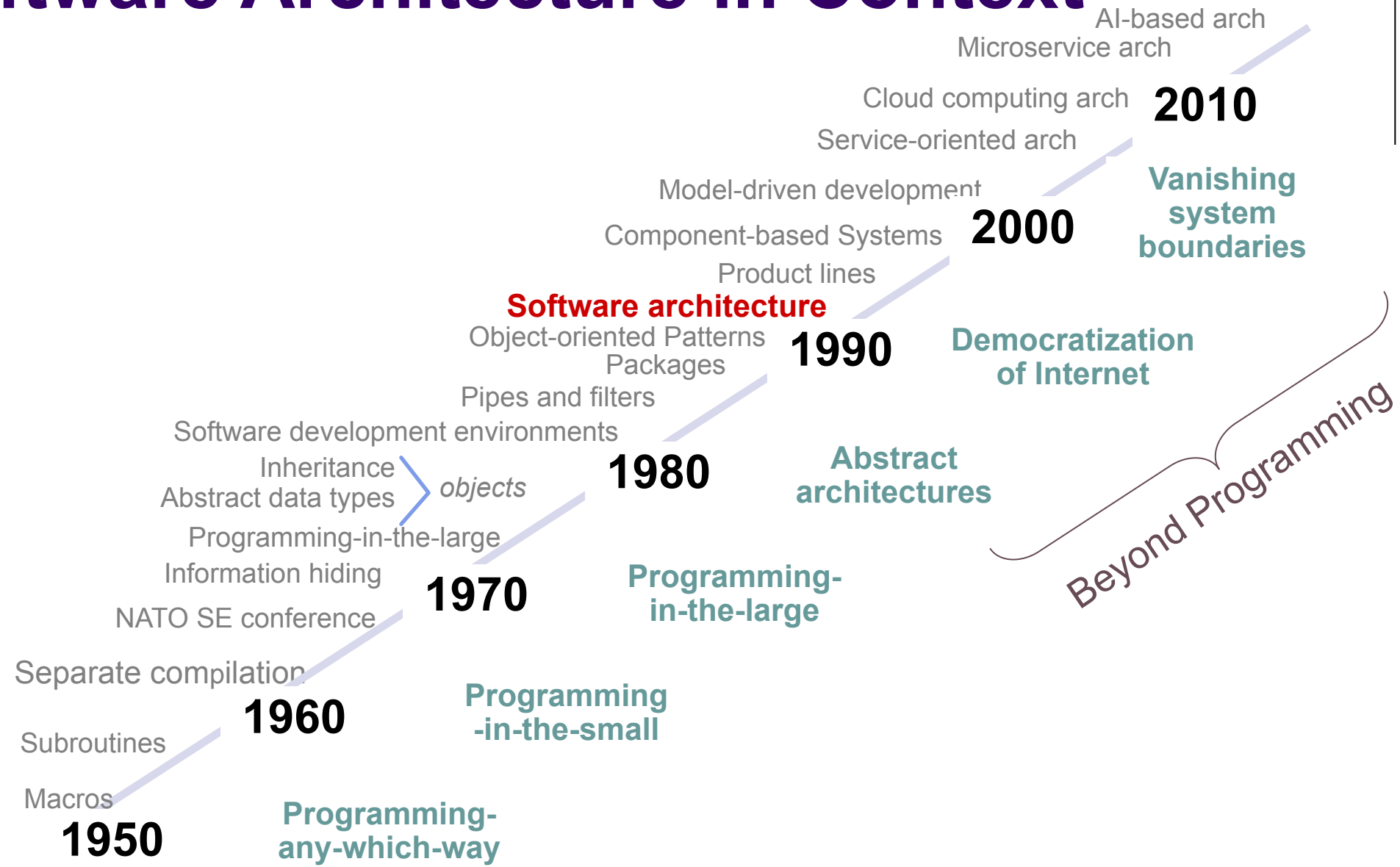
Architecture vs Programming



Architecture requires a different way of thinking.

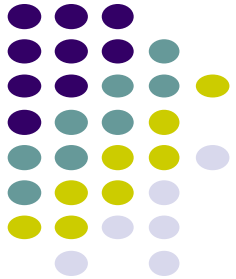
Architecture	Programming
Interactions among parts	Implementation of parts
Structural properties	Computational properties
Declarative	Operational
Mostly static	Mostly dynamic
System-level properties	Algorithmic properties
Outside module boundary	Inside module boundary

Software Architecture in Context



© Garlan & Schmerl

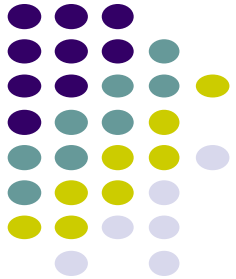
Evolution of the Field of Software Architecture – 1980's



- Informal use of *box and line diagrams*
- Ad hoc application of architectural expertise
- Diverse, uncodified use of architectural patterns and styles
- No identified "architect" on most projects

© Garlan & Schmerl

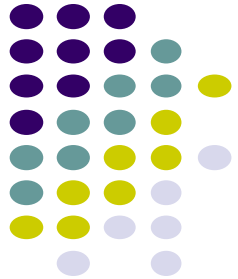
1990's



- Recognition of the value of *architects* and *architectural expertise* in software development organizations.
- *Processes* requiring architectural design reviews & explicit architectural documentation.
- Use of *product lines*, commercial architectural *standards*, component *integration frameworks.*
- *Codification* of vocabulary, notations & tools for architectural design.
- *Books/courses* on software architecture.

© Garlan & Schmerl

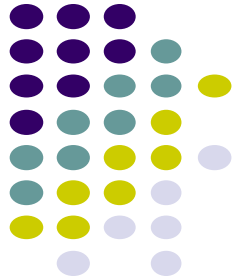
2000's



- Incorporation of architectural notions into mainstream *design languages* (e.g., UML-2), and *tools* (e.g., Rational Software Architect).
- *Methods* based on architectural design and refinement (e.g., Model-Driven Architecture).
- Some architecture *analysis tools.*
- Architectural *standards* for Enterprise Systems (e.g., RM-ODP, TOGAF).
- Skepticism about benefits of architecture in an agile world.

© Garlan & Schmerl

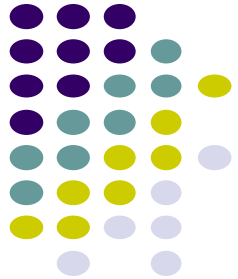
2010s



- Internet-enabled architectures
  - Web services, service-oriented architectures, clouds
- Platforms, frameworks, and ecosystems
  - .NET, iOS, Android, ROS
- Cyber-physical systems
  - Smart environments, energy-aware buildings, collision avoidance systems, robotics
- Integration of architecture with sw development processes
  - ACDM, Agile+Architecture,Technical Debt

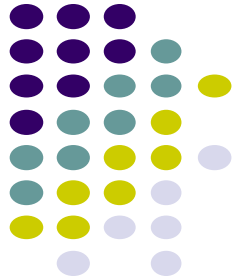
© Garlan & Schmerl

2020's



- Micro-services and cloud-native applications
  - Automated mechanisms for resource management (e.g., Kubernetes)
- DevOps, Continuous Integration (CI) and Continuous Delivery/Deployment (CD)
  - Designing for continuous incremental development and deployment of new features
- AI-based systems (systems with AI in them)
  - Self-driving cars, recommendation systems, chatbots
- Architectures for AI (systems for creating AI components)
  - ML pipelines, reinforcement learning

What are the conceptual tools to help with architectural design?



Three important concepts:

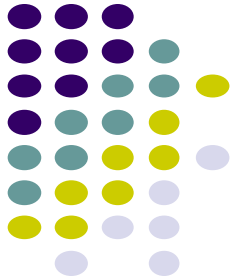
1. Views structure *what you can talk and reason about*
2. Styles constrain *how you organize systems*
3. Tactics refine *how qualities are achieved*

We need views to *reason*, styles to *organize*, and tactics to *achieve qualities*.

Recall our definition

The software architecture of a computing system is the set of **structures** needed to reason about the system, which comprise software elements, relations among them and properties of both.

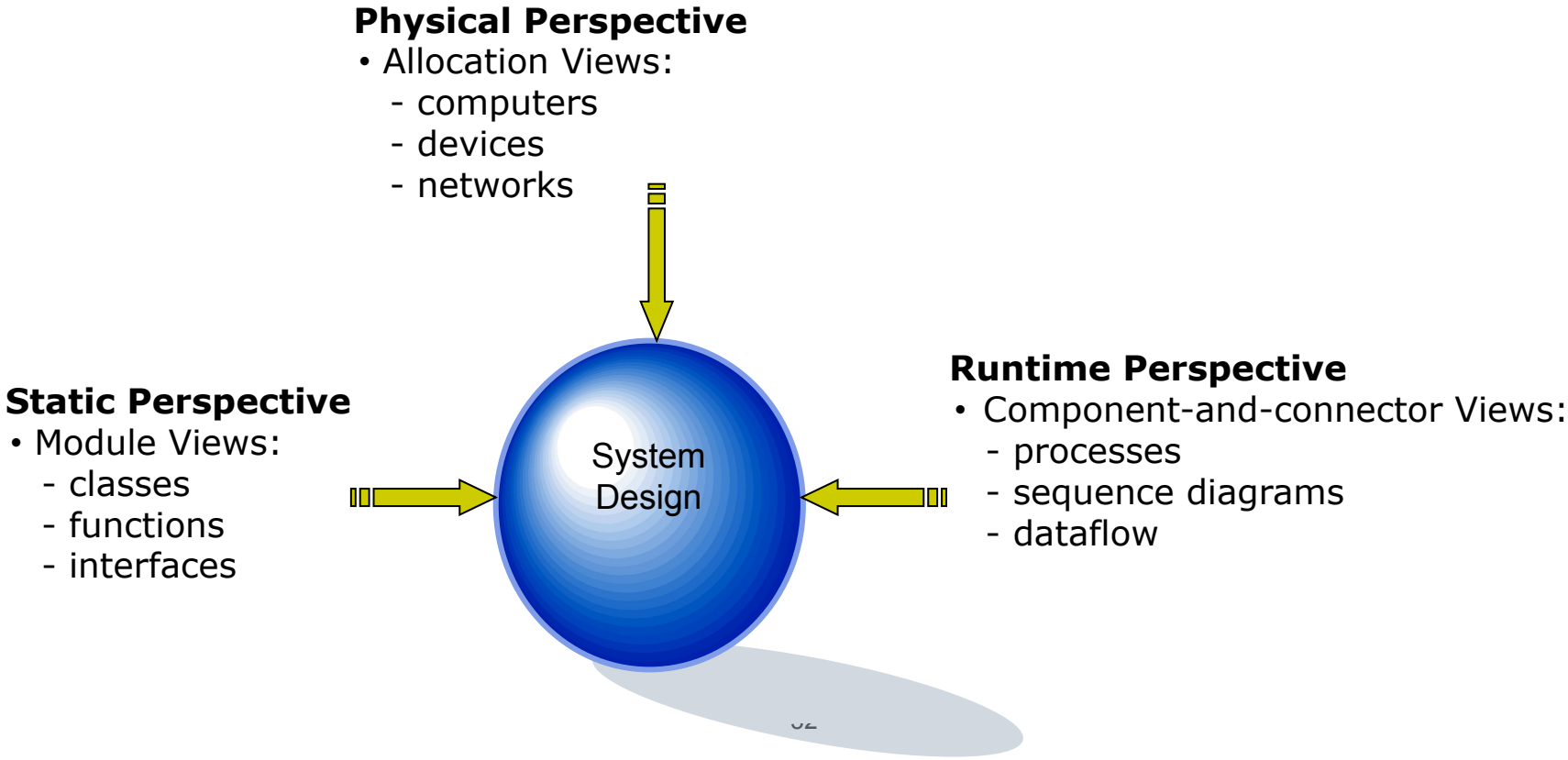
What is a Structure? – 1



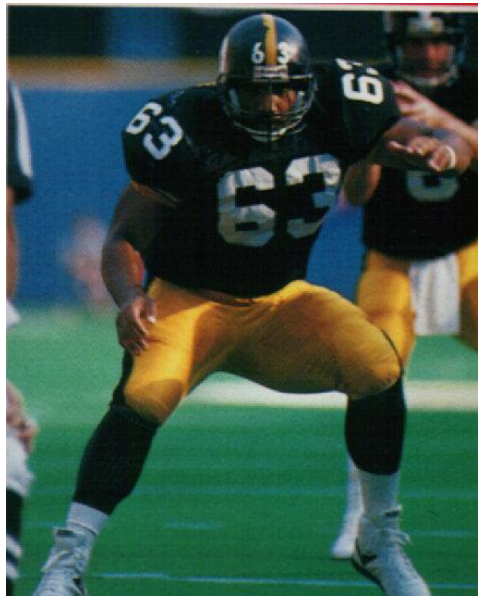
- Software architecture is an abstraction of the structures that comprise the software of a system.
- The goal is to reason about a system design
  - and thereby make good design decisions.
- But different kinds of reasoning may require different kinds of information/structures.
- Software has *many* structures, including
  - code
  - processes/threads/servers/databases …
  - files

Concept 1: Software architecture comprises different, separate (but related) views

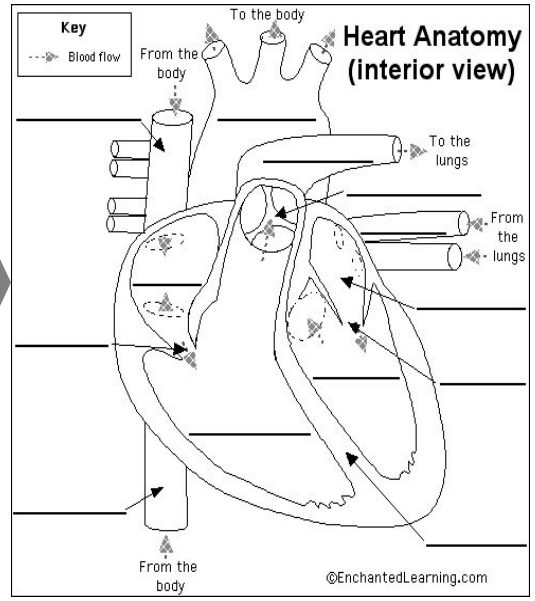
A representation of a type of structure is called *a view* of the system.



Structures and Views



A human body is comprised of multiple *structures.*



a *static view* of one human *structure*

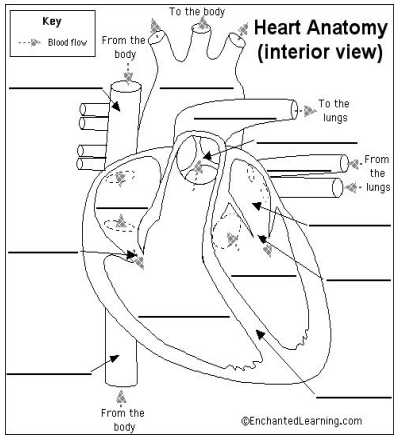


a *dynamic view* of that *structure*

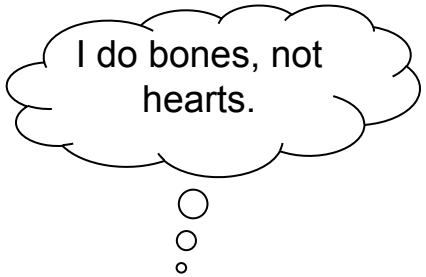
One body has many structures, and those structures have many views. So it is with software…

Structures And Views











These *views* are needed by the cardiologist…

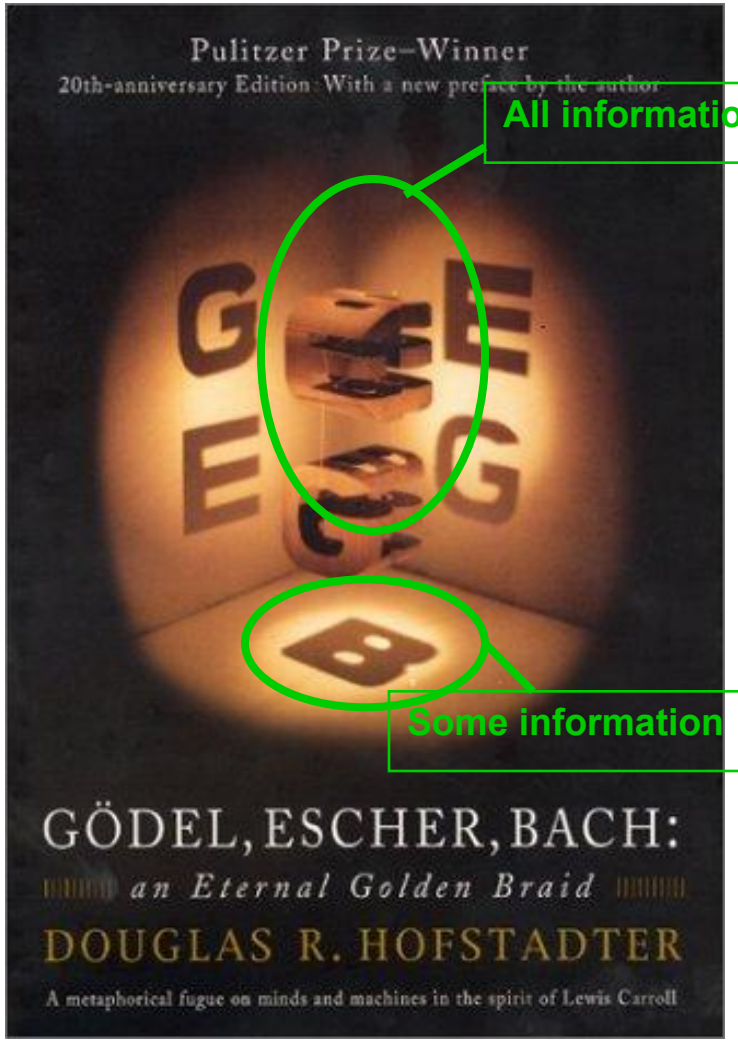
…but will these *views* work for the orthopedist?

Different stakeholders are interested in different structures.

Views must represent the structures that the stakeholders are interested in. *So it is with software…*

Views

- A view is a representation of a set of system elements and the relations associated with them.
- Not *all* system elements -- *some* of them.
- A view selects *element types* and *relation types* of interest, and shows those.

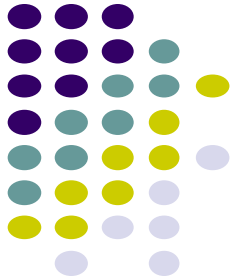


What Views Are Available?



- In this course we will consider how an architect depicts a system in three ways:
  1. How is it structured as a set of code units?
  2. How is it structured as a set of elements that have run-time behavior and interactions?
  3. How does it relate to non-software structures in its environment?

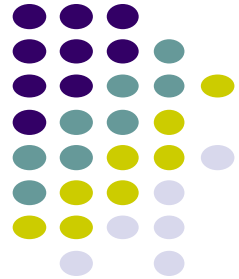
What Views Are Available?



This suggests leads to the definition of 3 kinds of views:

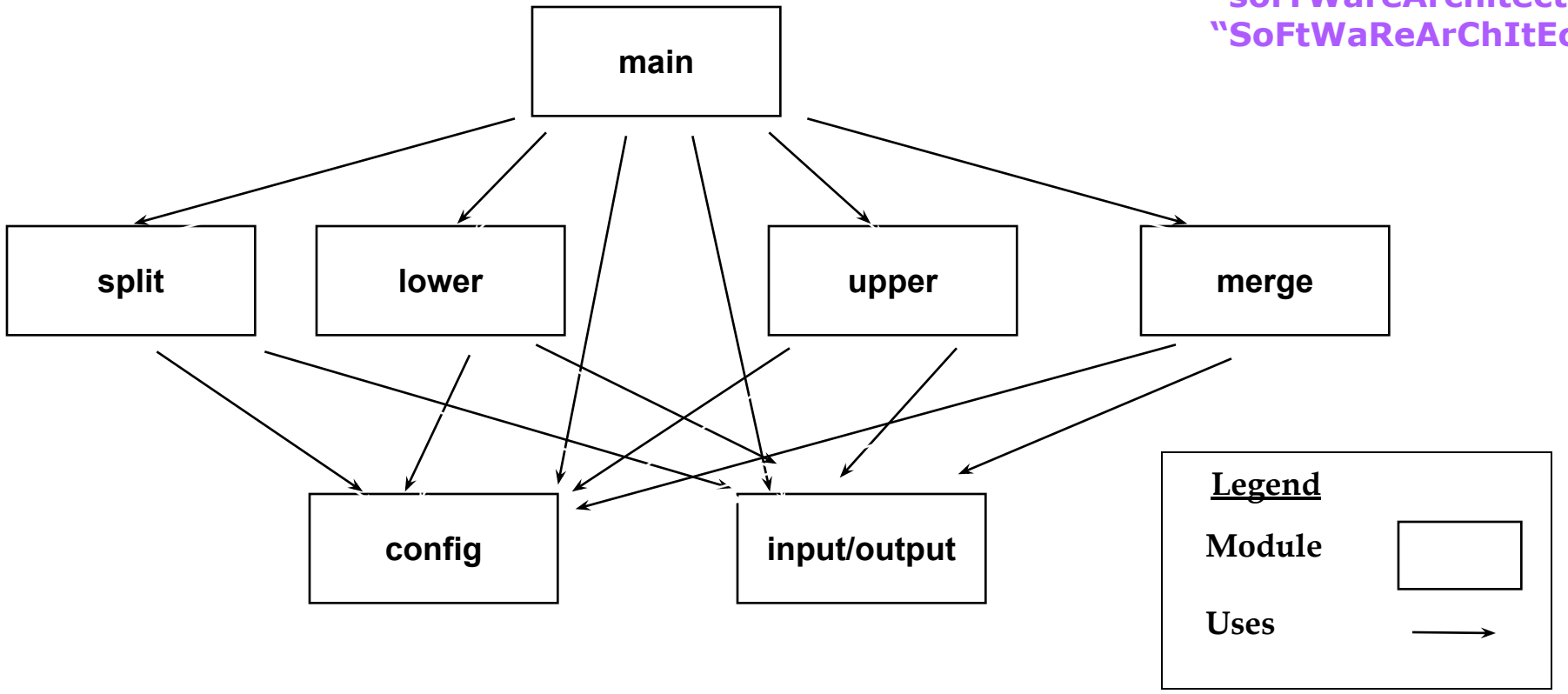
1. How it is structured as a set of code units.
   (*Module Views*)
2. How it is structured as a set of elements that have run-time behavior and interactions.
   (*C&C Views*)
3. How it relates to non-software structures in its environment.
   (*Allocation Views*)

Example: Alternating Characters Code (Module) View



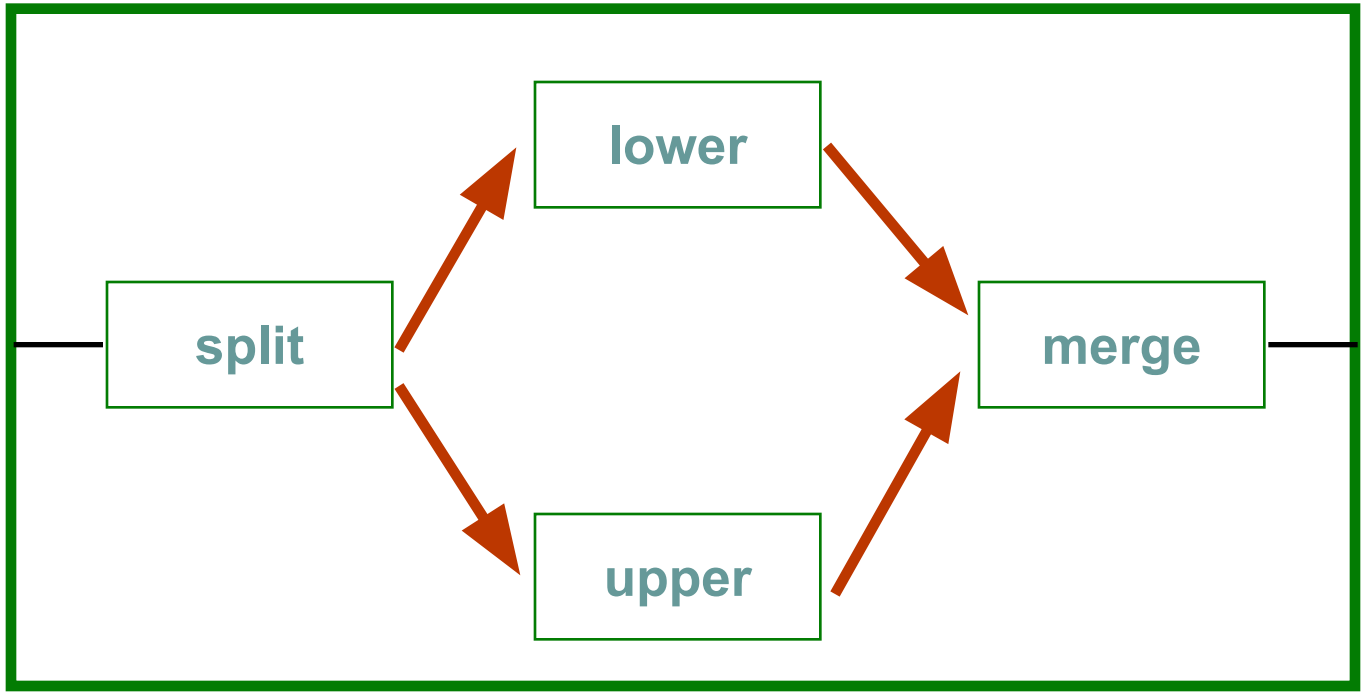
Produce alternating case of characters in a stream

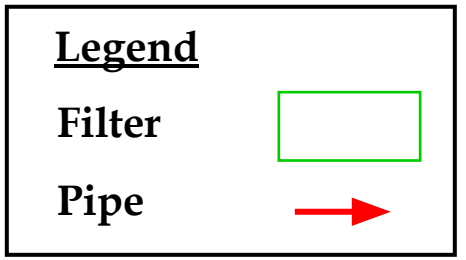
"sofTWareArchitecture" =>
"SoFtWaReArChItEcTuRe"



Definition/Use Modularization

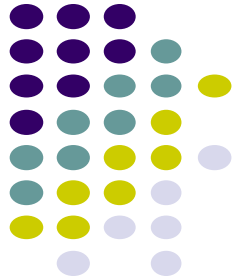
Example continued: Run-time (C&C) View





Components and Connectors

Concept 2: Architectural Styles

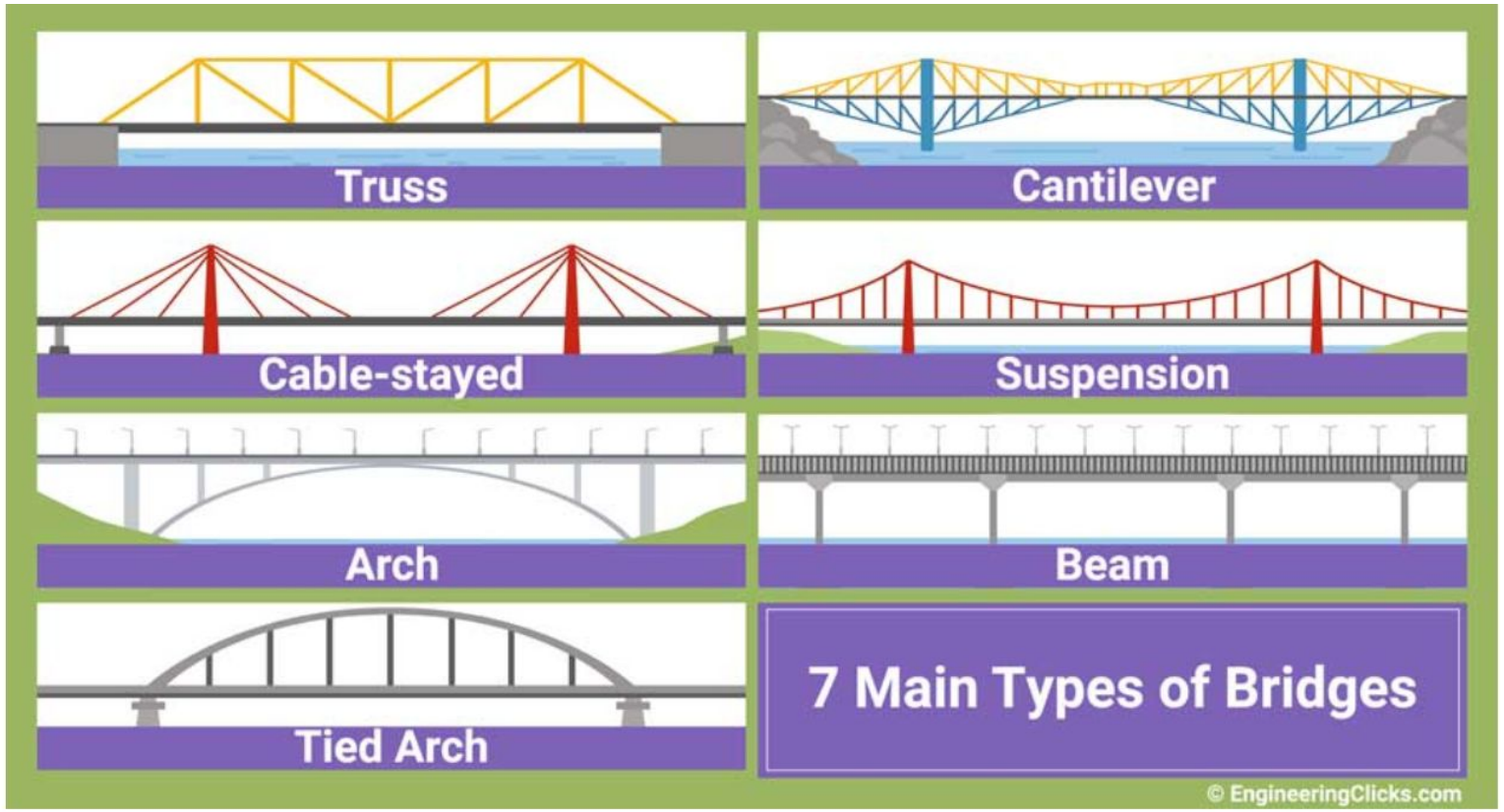


- Within each kind of view, recurring forms have been widely observed in different systems.
- These forms, or patterns, are worth capturing because they have known properties and can be re-used.
- We call these styles:

  An architectural style is a set of element and relation types, together with a set of constraints on how they can be used.

Consider Bridge Design

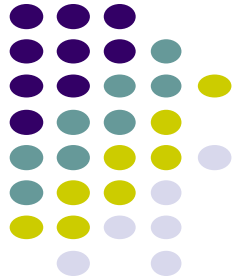
- Bridge designers have a vocabulary of different kinds (or styles) of bridges.



- These form the starting point for any bridge design.
- Which styles are appropriate will depend on many factors.

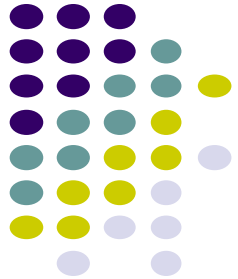
https://mechanical-engineering.com/types-of-bridges/amp/

Architectural Styles



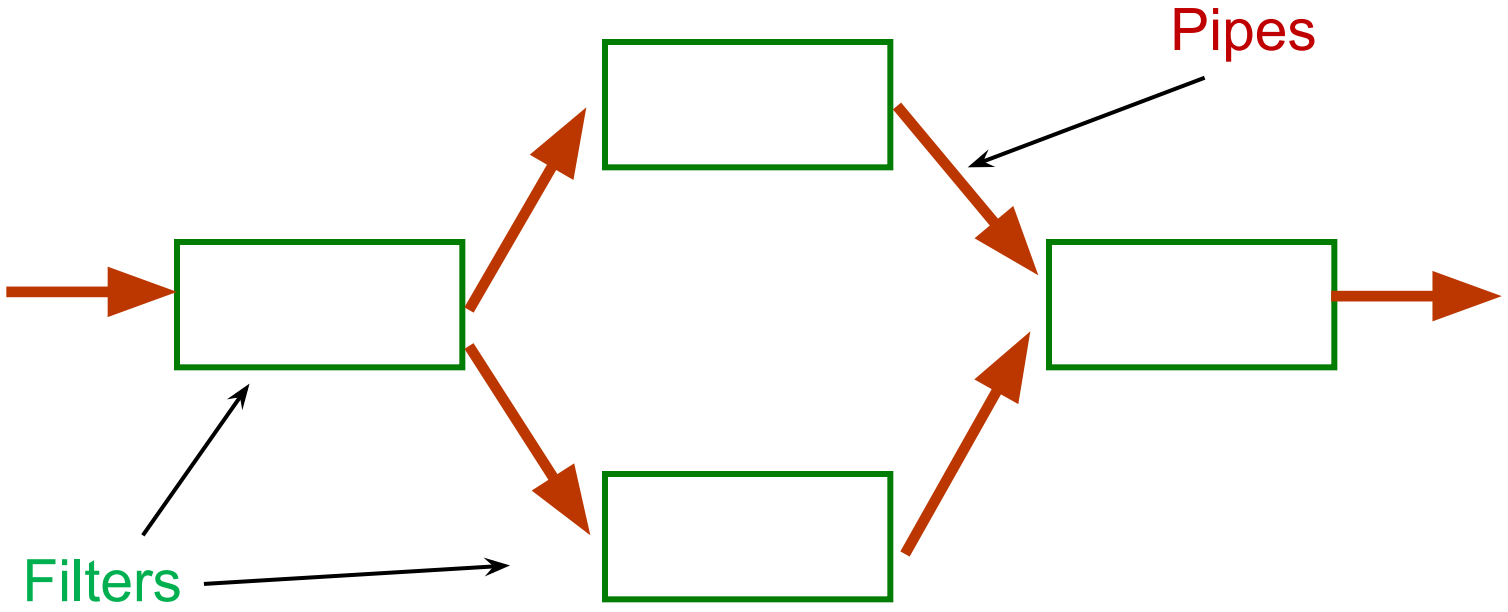
- C&C styles
  - These represent ways of structuring the running system.
  - They determine the primary form of interaction between run-time components.
- Module styles
  - These represent ways of structuring the implementation artifacts.
  - They determine what modules exist and how they depend on each other.
- Allocation styles
  - How software is allocated to physical elements.

Component-and-connector (C&C) Views



- Elements:
  - Components: principal units of run-time computation and data stores
  - Connectors: interaction mechanisms
- Relations: Attachment of components' *ports* to connectors' *roles* (interfaces with protocols)
- Properties: specify information for construction & analysis
  - Quality attributes (to help analysis)
  - Others, depending on style

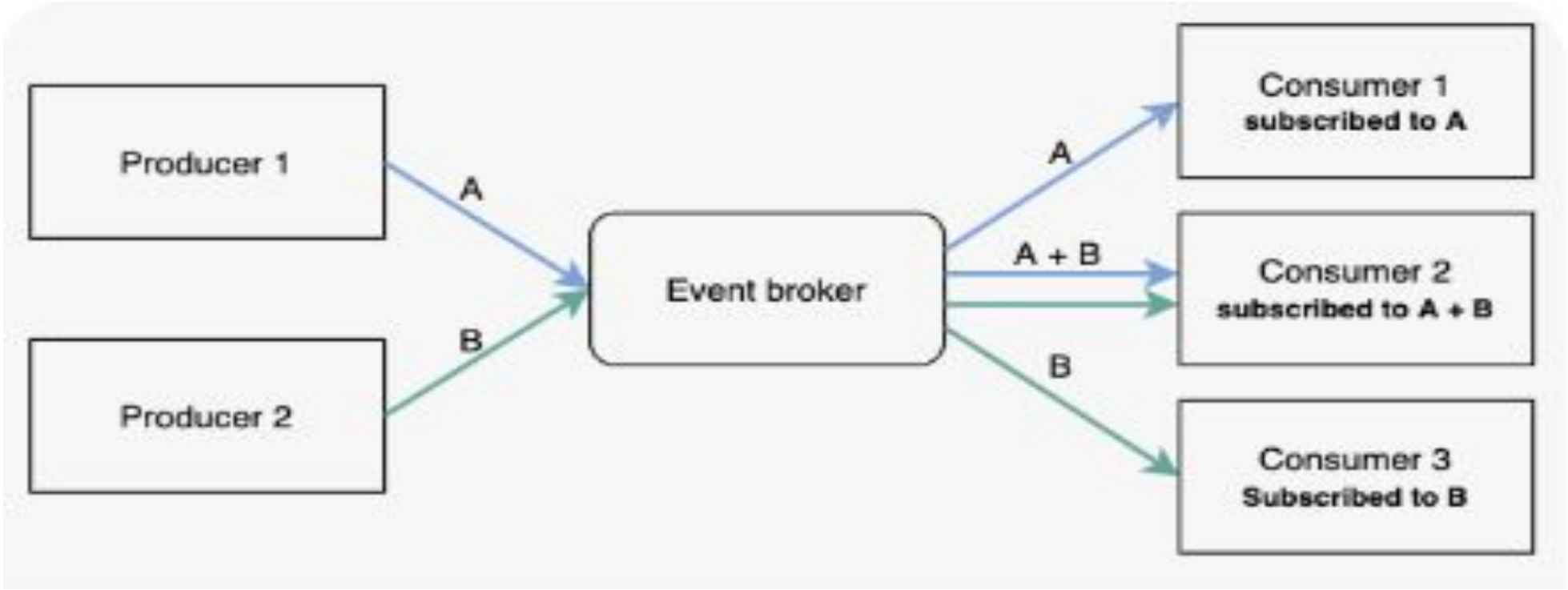
Example Style: Pipe-and-Filter Systems



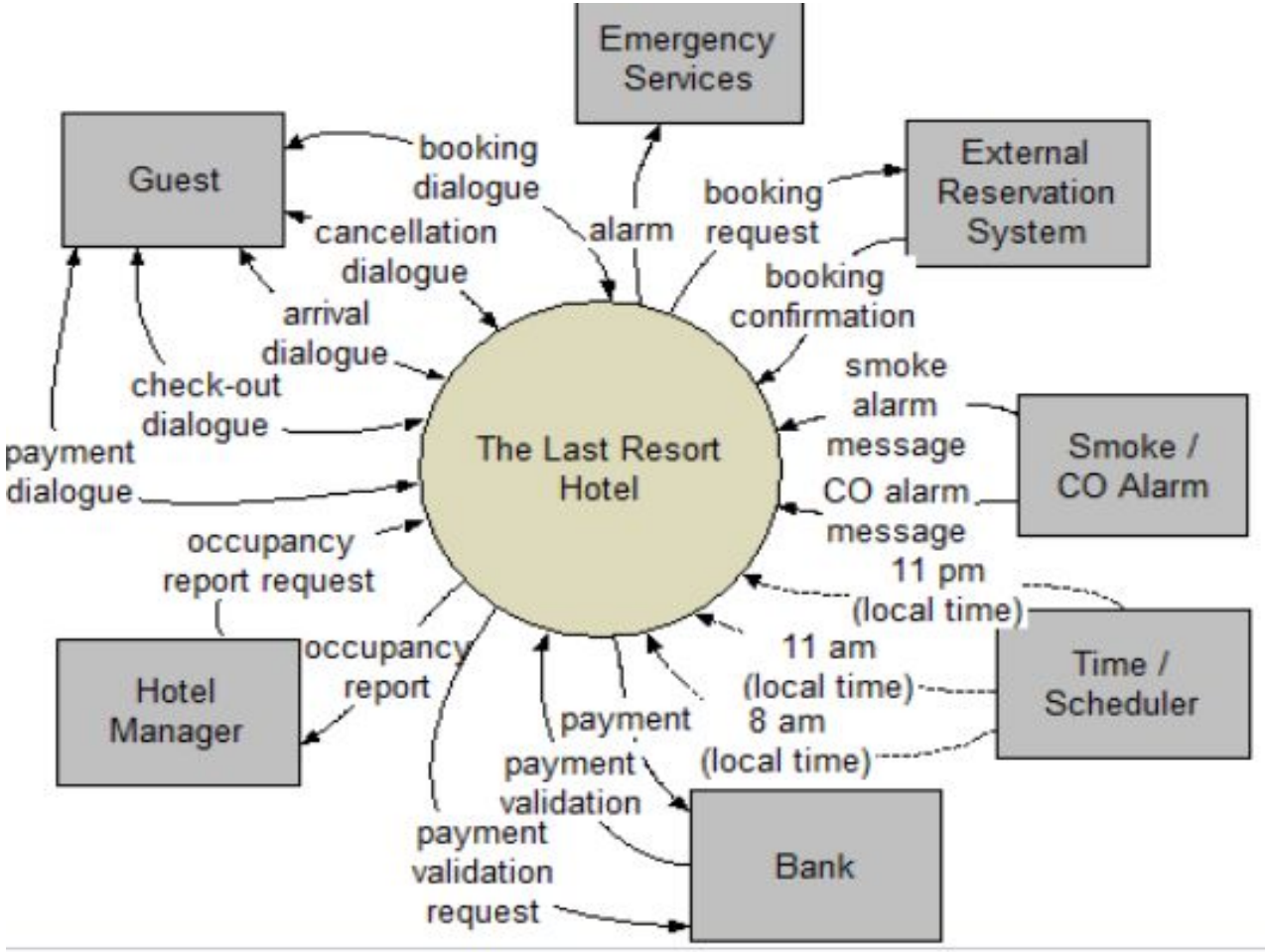
Example Pipe-and-Filter Systems



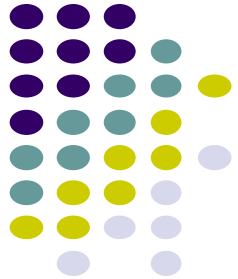
Example Style: Event-oriented Systems



Example Event-based System

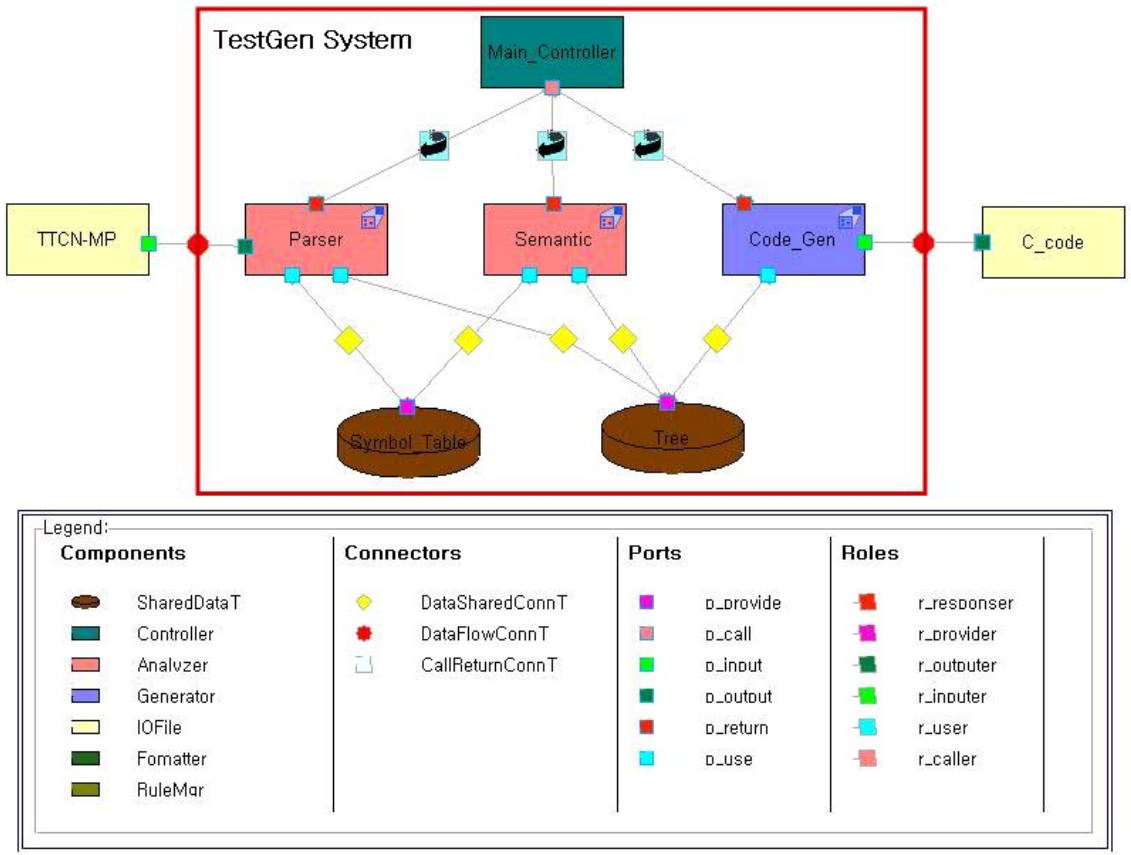


A Taxonomy of (run-time) Styles

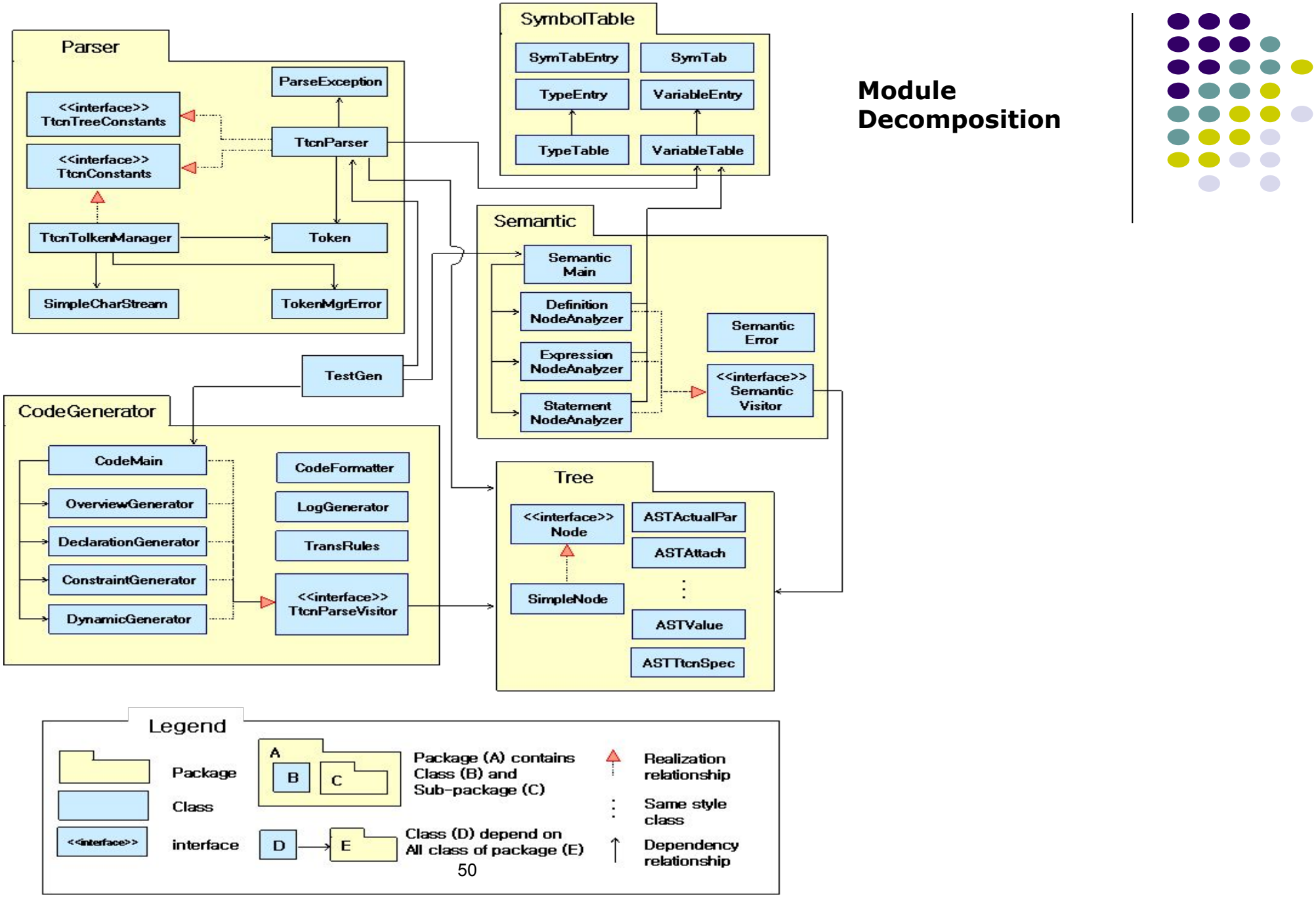


- Data flow
  - batch sequential
  - pipes and filters
  - workflow
- Call-return
  - main program-subroutine
  - object-oriented
  - component-based
  - peer-to-peer
  - service-oriented
  - N-tiered
- Event-based
  - asynchronous messaging
  - publish-subscribe
  - implicit invocation
  - data-triggered
- Data-centered
  - repository
  - blackboard
  - shared variables

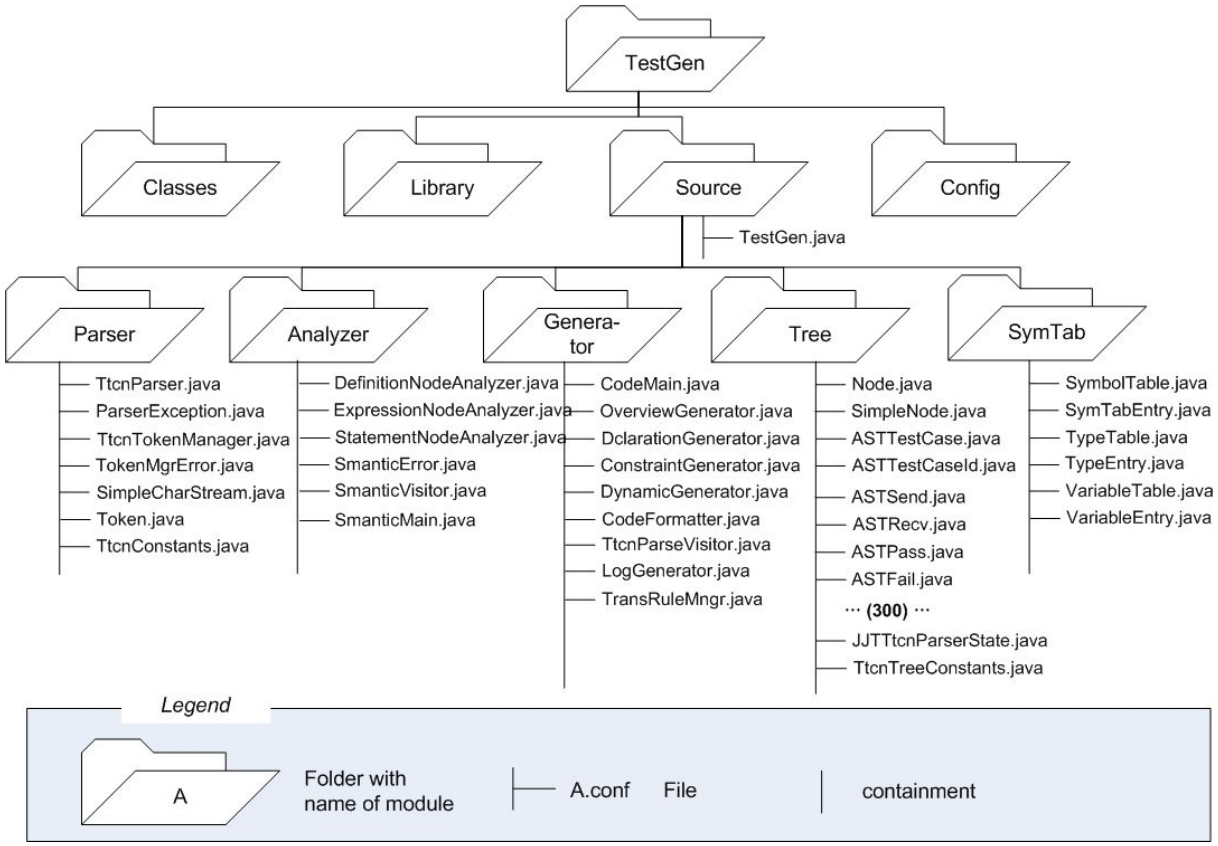
C&C View of Test Parsing System



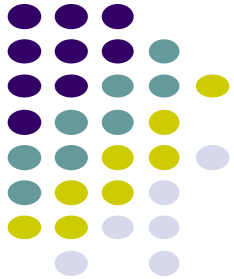
Module Decomposition



Implementation Style for the same system

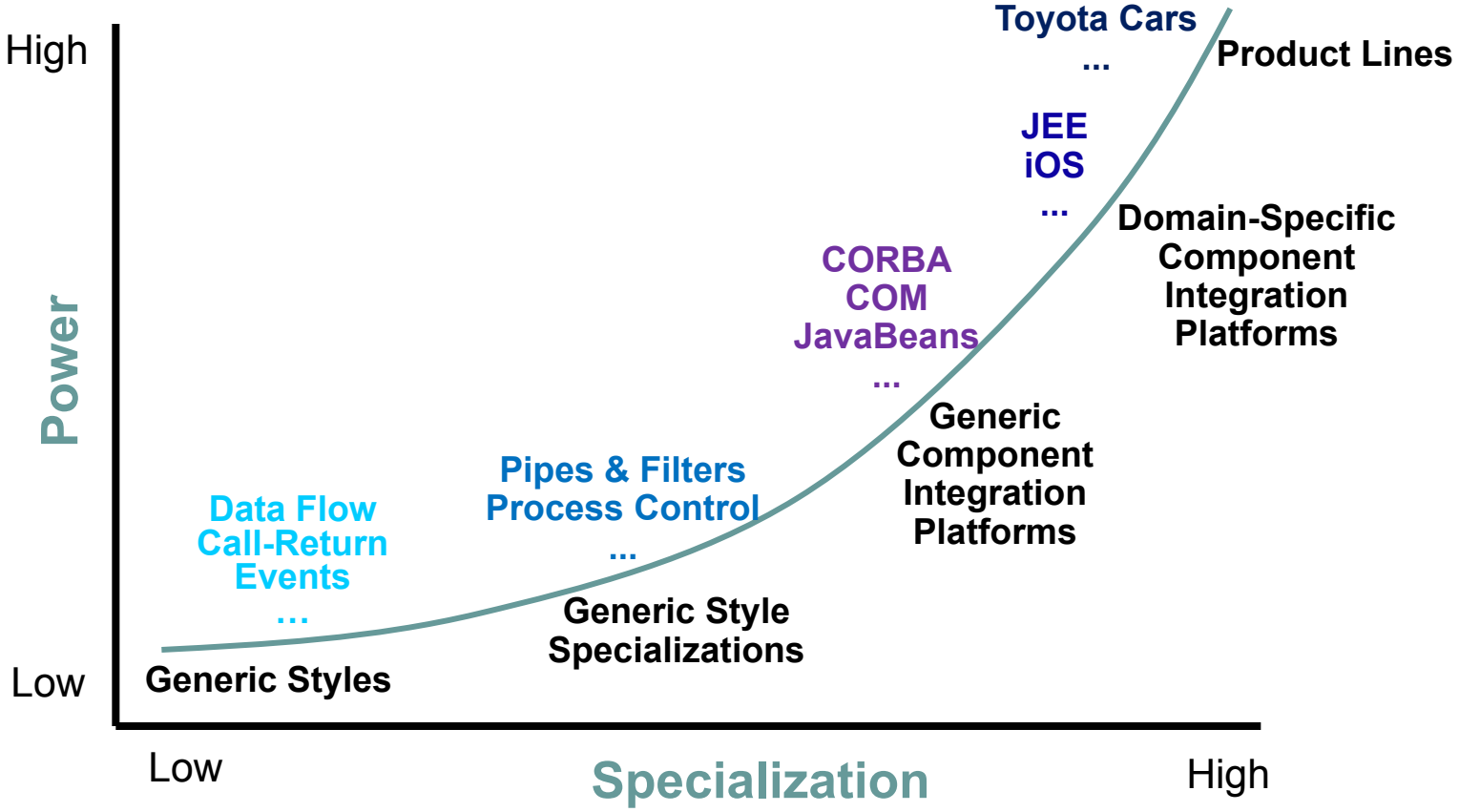


Styles in Practice

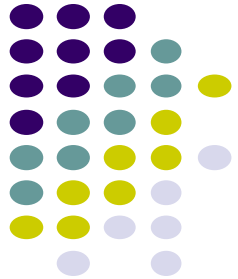


- Many styles are specializations of general styles
  - Targeted to specific domains – see next slide
- Systems often use a combination of styles
  - Example: tiers + repository + events
- But we should not ignore the pure versions
  - We need to understand the basic conceptual building blocks
- Moreover, in general it's a bad idea to mix styles from *different perspectives*
  - Example: layers and tiers

Styles, Platforms, and Product Lines



Concept 3: Tactics

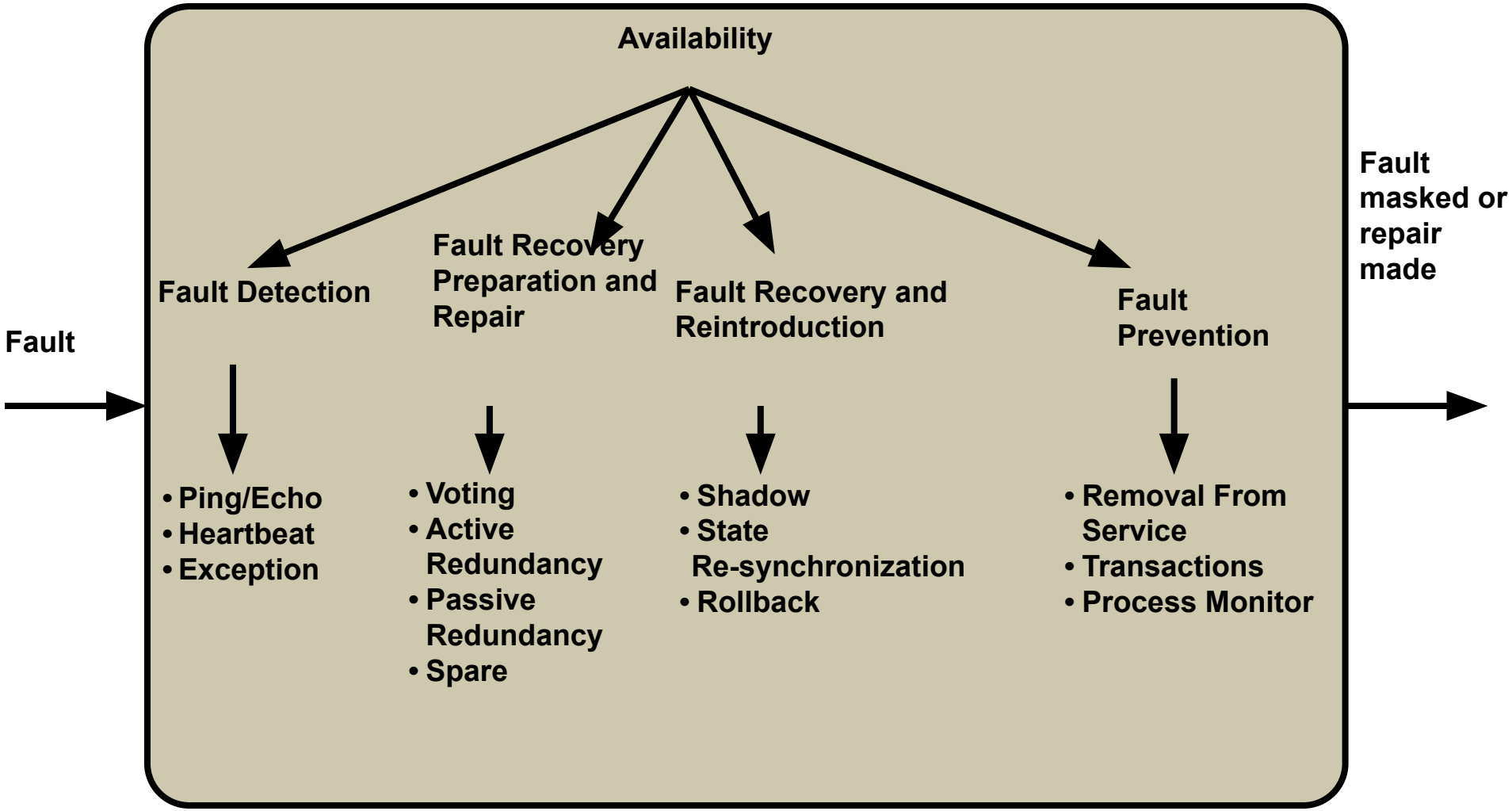


- Architectural styles determine what kinds of elements and relationships will exist in a system.
- But they do not by themselves necessarily achieve quality objectives.
- In addition, we need a set of techniques that work within a style and are focused on improving some quality.
- We will call these techniques “tactics”.

Examples of Tactics

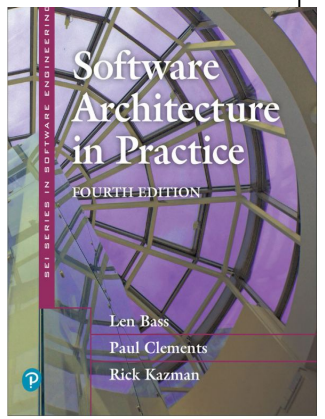
- Availability
  - Use of redundancy
  - Self-monitoring
  - Degraded system modes
- Performance
  - Caching
  - Pipelining
- Security
  - Firewalls
  - Encryption

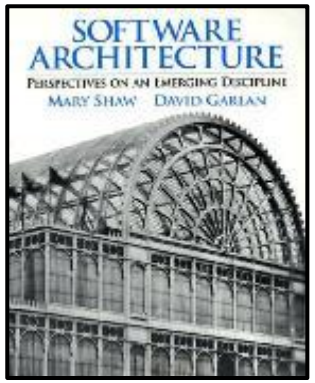
Example: Availability Tactics

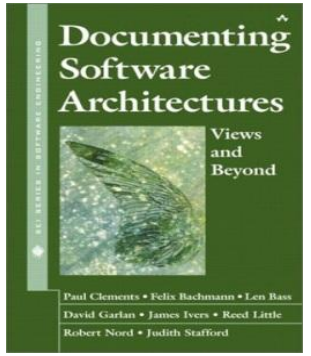


Readings

- Text and Readings
  - *Software Architecture in Practice, 4th Edition.* Bass, Clements, & Kazman. Addison Wesley, 2021. [BCK21]
  - Readings, available on course website.
- Additional Reference Materials
  - *Software Architecture: Perspectives on an Emerging Discipline.* Mary Shaw and David Garlan. Prentice-Hall, 1996. [SG96]
  - *Documenting Software Architectures: Views and Beyond, 2nd Ed.*, Clements, et al. Addison Wesley, 2010. [C+10]







© Garlan & Schmerl