Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Current Best Practices in Software Architecture

Paul Clements
Software Engineering Institute
Carnegie Mellon University
April 2006
Tandil, Argentina

Sponsored by the U.S. Department of Defense

Carnegie Mellon
Software Engineering Institute

Software Engineering Institute

Applied R&D laboratory situated as a college-level unit at Carnegie Mellon University, Pittsburgh, PA, USA

Established in 1984

Technical staff of 335

Offices in Pittsburgh, Pennsylvania, Arlington, Virginia, and Frankfurt, Germany

Purpose: Help others make measured improvements in their software engineering practices

Product Line Systems Program

One of 5-6 programs at the SEI; approx. 30 people.
Our goal is to make improvements in

- Software product line engineering
- Predictable assembly of certifiable components
- Software architecture
  - Creation
  - Documentation
  - Evaluation
  - Use in system-building

Introductions

Who are you?

Who am I?

Topic

What do we mean by software architecture?

Why is it important?

Software Architecture's role in Software Engineering

Some say that software engineering is about creating *high-quality multi-version multi-person* software.

If we were just building a single system, all by ourselves, that was never going to change, all we would need is programming.

Multi-person

For non-trivial software, teams of people cooperate to build it. Teams may be

- In the same room
- In the same building
- In the same country
- On the same planet
- …or not.

Our systems need to be decomposable into pieces, such that

- Teams can work in parallel
- Inter-team communication is not too burdensome

Multi-version

At the dawn of computer science, people were most concerned with programs that computed the right answer.

- Ballistic calculations
- Numerical problems
- Simulations and models of real-world processes

Multi-version

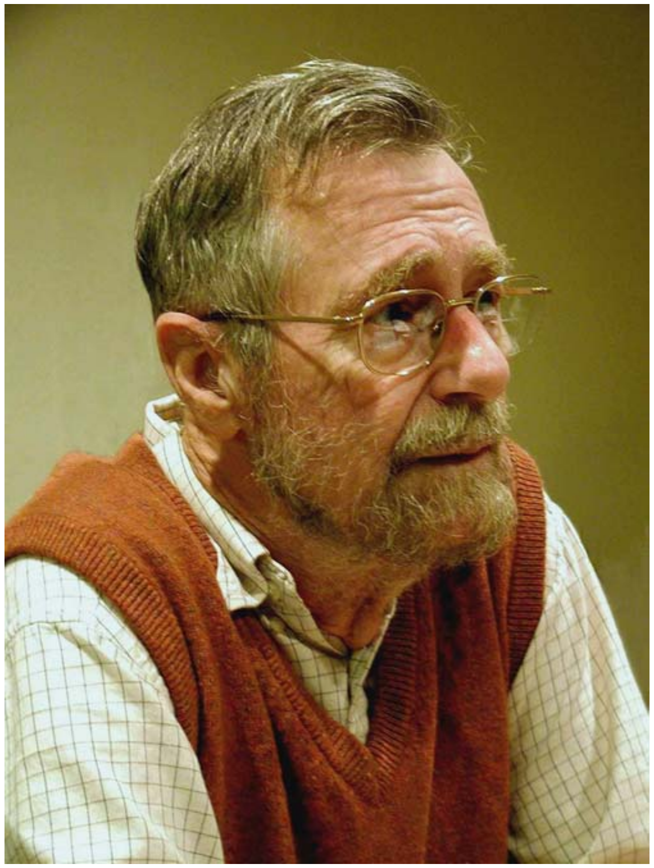
In the late 1960s and early 1970s, people like Edsger Dijkstra, David Parnas, Fred Brooks, Anthony Hoare, and Harlan Mills were arguing that this was not enough.

They argued that systems could be constructed in better ways

- To make changes easier.
- To make the programs easier to understand
- To make the programs less likely to contain errors
- To make the programs easier to test

They argued that *structure matters*!

Edsger Dijkstra (1930-2002)



Contributions to architecture

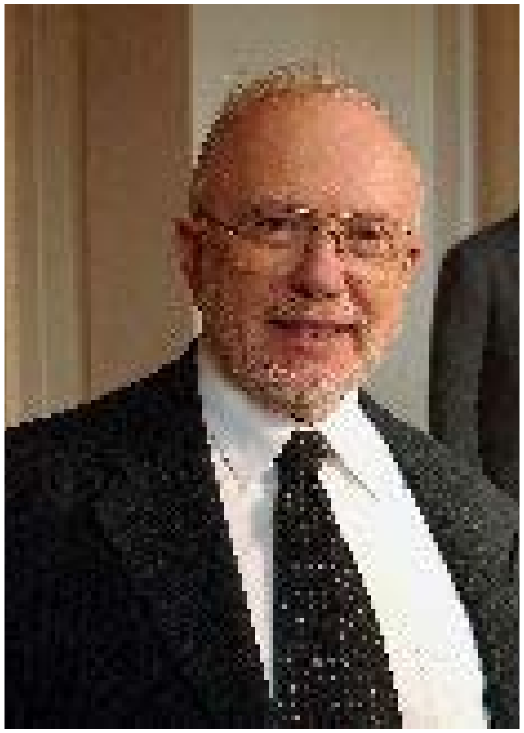
•"On the T.H.E. Operating System" (1968)

•Described the classic operating system design,

•A brief appendix describes semaphores.

•May have been the first significant and/or popular report on layered systems

David Parnas



Contributions to architecture

•"On the Criteria for Decomposing Systems into Modules" (1972) gave us information-hiding as a design principle. From outside, we can work with a component only by its interface!

•"Designing Software for Ease of Extension and Contraction" (1975) introduced a useful architectural relation: "uses"

•"On a 'Buzzword': 'Hierarchically Structured Systems'" (1976) taught us that systems have many structures.

Managing Change

Today we know that usually, most of the cost of a piece of software comes *after* it has been deployed for the first time.

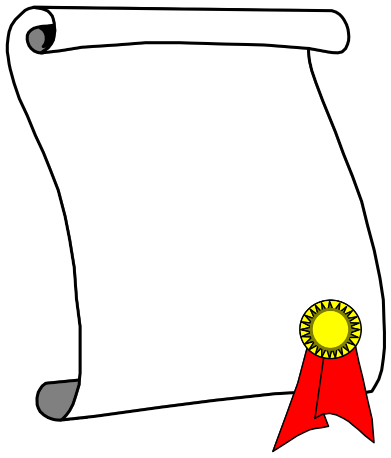
Maintenance is very expensive. Anything we can do to make systems easier to change will save money in the long run.

Planning and designing for change is a very important part of software engineering.

High quality

Since the early times, other qualities have joined “ease of change” as important parts of software engineering.

- Performance
- Security
- Availability
- What others?



High quality

In fact, some say that software engineering is about the achievement of quality attributes in a software-intensive system.

It turns out that getting the right answer has become the easy part.

Does this surprise you?

Quality attributes

A system that computes the right answer – i.e., has the right *functionality* -- but

- Takes too long to do it
- Allows hackers to break in and steal its data
- Is down too much of the time
- Cannot be changed in less than six months

…is not going to be a successful system. Nobody will want to use it. Nobody will buy it.

Quality attributes

If we accept the importance of quality attributes, then we need to understand how to specify them…

- Our customer has to tell us what he wants
- Our architect and designers must understand it
- Our programmers have to achieve it
- Our testers have to test for it

…and how to design and build software to achieve them.

Software architecture helps with this.

QA’s fall into two groups

“Run-time” QA’s

- We can measure how well a system exhibits these by watching the system in operation
- Performance, security, availability, …

“Non-run-time” QA’s

- We can measure these by watching a team in operation
- Maintainability, portability, buildability, time to market…

Software architecture

The rise of software architecture has resulted from two trends:

- Recognition of the importance of quality attributes
  - Increasingly "time to market" is critical
- The development of very large and very complex systems.

Software architecture

Large-scale design decisions cannot be made by programmers.

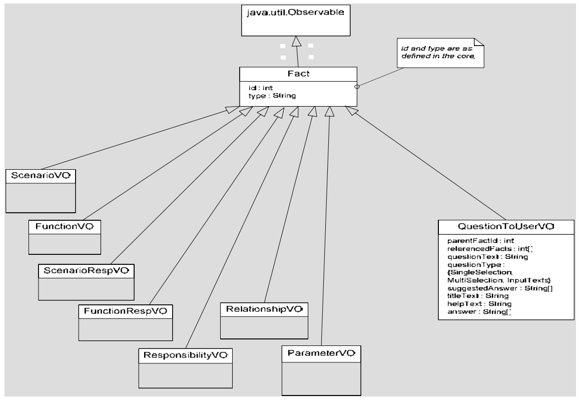
- Have limited visibility and short-term perspectives
- Trained in technology solutions to specific problems.

Teams can only be coordinated, and QA's can only be achieved, by making broad design decisions that apply to the entire system – all of its elements.

Summary so far

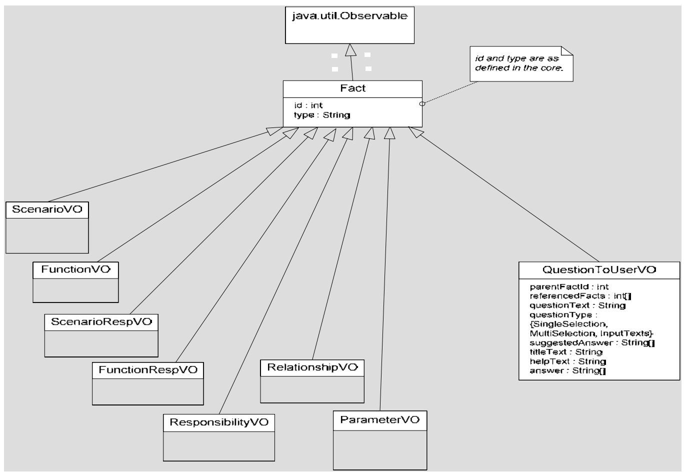
What do we know so far? A software architecture

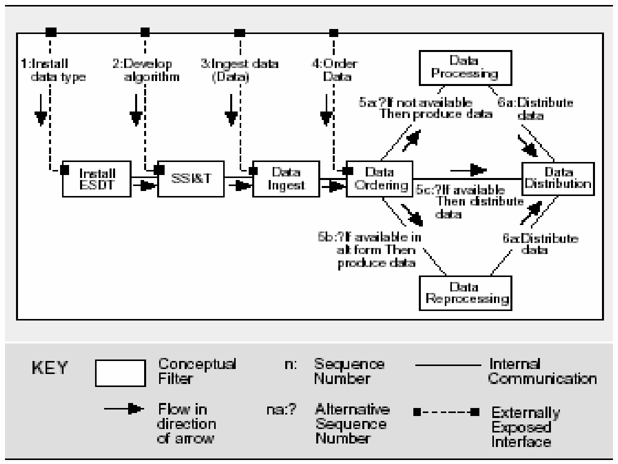
- Exists to achieve a system’s quality attributes
- Exists to allow parallel development by distributed teams (a special kind of quality attribute)
- Involves decomposing a whole into parts
- Involves system-wide design decisions such as
  - How the parts work together to achieve the system’s function and goals



Software architecture and structure

Software architecture is largely about structure:
- What the pieces are
- What each one’s responsibility is
- How the pieces work together





How does structure help?

By concentrating on structure, we treat the pieces as atomic, as black boxes. This reduces detail we have to tend to, and we can postpone that consideration until later.

It separates concerns between structure (the pieces) and the details of implementing the pieces (which is a job we give to programmers).

Suppressing the internal details of the elements does not affect how the elements are used or how they relate to or interact with other elements.

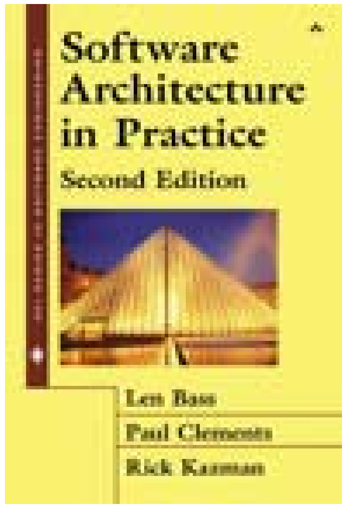
It makes an architecture an *abstraction* of a system – which is a simplification.

The Ascendance of Software Architecture

Over the past 10 years, software architecture has emerged as the prominent paradigm in large-system development.

There are:

- worldwide conferences devoted to it
- books devoted to it
- defined "architect" roles in organizations
- courses and training for it



And yet...

It is still not well understood in some circles.

Some organizations have no “architect” position. Others have the position but it is informally defined.

Some organizations are still proceeding to development without an architecture in place.

The tools of the trade -- styles and patterns, views, evaluation -- are used sparingly if at all.

Role of Software Architecture

If the only criterion for software was to get the right answer, we would not need architectures—*unstructured, monolithic systems would suffice*.

But other things also matter, such as

- modifiability
- time of development (time to market)
- performance
- coordination of work teams

System qualities are largely dependent on architectural decisions.

- All design involves tradeoffs in system qualities.
- The earlier we reason about tradeoffs, the better.

What is your definition of software architecture?

The way that you partition a system or software process for defining the teams that work on it.

A view or a structure of the main components of the system and how they interact.

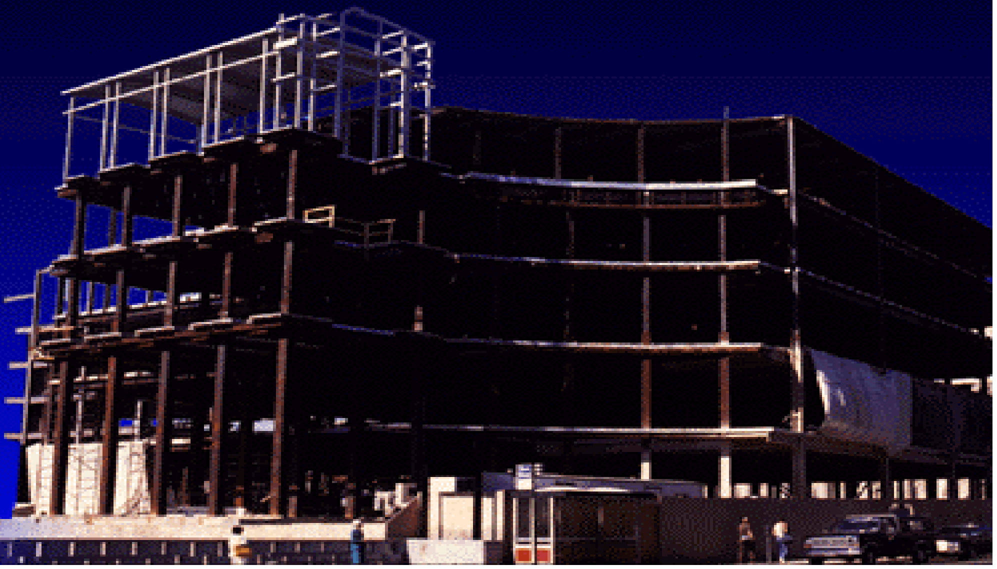
A tool for managing complexity and an abstraction of reality.

Carnegie Mellon
Software Engineering Institute

What Is Software Architecture?

Software architecture is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them.

Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition.* Boston, MA: Addison-Wesley, 2003.



Implications of this Definition

Every system *has* an architecture. If you don't explicitly develop an architecture, you will get one anyway.

- Every system is composed of elements and there are relationships among them.

Just having an architecture is different from having an architecture that is known to everyone:

- Communicating (documenting) the architecture becomes an important concern.

The architecture might not be the right one.

- Evaluating the architecture becomes an important concern.

“Externally visible properties”?

This refers to those assumptions that one element can make about another element such as

- the services it provides
- how long it takes
- how it handles failures
- how it uses shared resources

Elements interact with each other via *interfaces* that partition details into public and private parts.

Architecture is concerned with the public side of this division.

Structures: Plural!

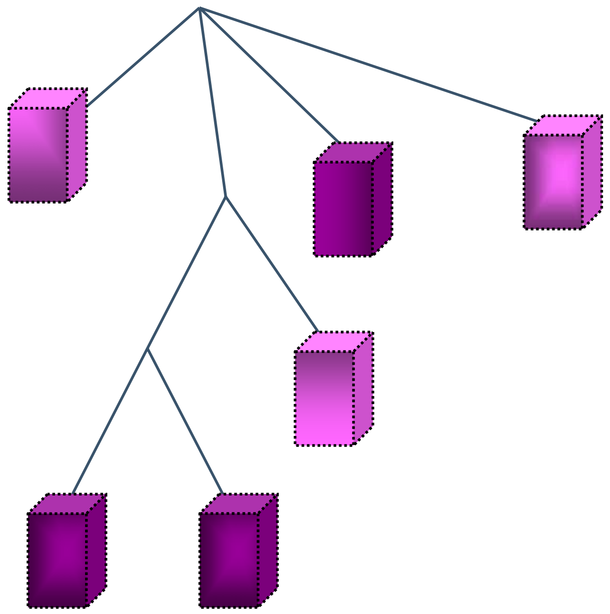
Systems can and do have many structures.

- No single structure can be *the* architecture.
- The set of candidate structures is not fixed or prescribed.
- Relationships and elements might be runtime related such as
  - “sends data to,” “invokes,” or “signals”
  - processes or tasks
- Relationships and elements might be nonruntime related such as
  - “is a submodule of,” “inherits from,” or “is allocated to team X for implementation”
  - a class or library

Structures: Plural!

This means that box-and-line drawings alone are *not* architectures; but they are just a starting point.

- You might *imagine* the behavior of a box labeled “database” or “executive” -- but that’s all
- You need to add specifications and properties.
- You need to specify what the boxes are and what the lines mean!



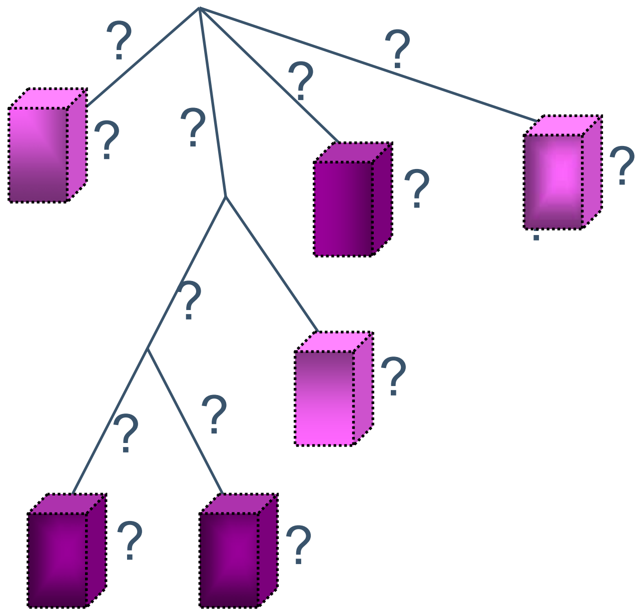
Box-and-line drawings

Box-and-line diagrams are a common form of architectural notation.

But what do they mean?

If you use a box-and-line diagram, always define precisely what the boxes and lines mean.

If you see a box-and-line diagram, ask the owner what it means. The result is usually entertaining.



Are these modules? Objects? Classes? Processes? Functions? Code units? Execution units? Other?

Old ideas die hard

If you hear someone say

“Architecture is the overall structure of the system.”

…I hope you will disagree.

…I hope you will know how to answer them.

Why is architecture so valuable?

Why is it worth studying and building?

Why is it worth *investing* in?

Communication Vehicle among Stakeholders

Architecture provides a common language in which the architect can communicate with the stakeholders, and stakeholders can communicate with each other.

This happens when

- negotiating requirements with users and other stakeholders
- keeping the customer informed of progress and cost
- implementing management decisions and allocations
- Informing stakeholders about design decisions and tradeoffs

Architecture Constrains the Implementation

An architecture defines constraints on an implementation.

- Architectures are descriptive and prescriptive
  - descriptive for communication
  - prescriptive for design and implementation
- Global resource allocation decisions constrain implementations of individual components
- System tradeoffs regarding quality attributes are architectural.
  - Not all QA's are possible all at once. We might have to (for example) give up some reliability to gain some performance. Architects make these tradeoffs.

The Development Project is Organized Around Architectural Elements

The architecture influences the organizational structure for development/maintenance efforts. Examples include

- division into teams
- assignment of work
- units for budgeting, planning by management
- basis of work breakdown structure
- organization of documentation
- organization of CM libraries
- basis of integration
- basis of test plans, testing
- basis of maintenance
- Incremental deployment

Carnegie Mellon
Software Engineering Institute

Architecture Permits/Precludes Achievement of Quality Attributes

For example

If you desire	*Examine*
performance	inter-component communication
modifiability	component responsibilities
security	inter-component communication, specialized components (e. g., kernels)
scalability	localization of resources
ability to subset	inter-component usage
reusability	inter-component coupling

Architecture is Key to Managing Change

An architecture helps reason about and manage change.

- important since ≈80% of effort in systems occurs *after* deployment

Architecture divides all changes into three classes:

- <u>local</u>: modifying a single component
- <u>non-local</u>: modifying several components
- <u>architectural</u>: modifying the gross system topology, communication, and coordination mechanisms

A “good” architecture is one in which the most likely changes are also the easiest to make.

Architecture is Basis for Incremental Development

An architecture helps with evolutionary prototyping and incremental delivery.

- Architecture serves as a skeletal framework into which components can be plugged.
- By segregating functionality into appropriate components, experimentation is easier.
- Risky elements of the system can be identified via the architecture and mitigated with targeted prototypes.

Architecture is a Reusable Model

An architecture is an abstraction: enables a one-to-many mapping (one architecture, many systems).

Systems can be built from large, externally developed components that are tied together via architecture.

Architecture is the basis for product (system) commonality.

Entire *software product lines* can share a single architecture.

Architecture and structure, re-visited

Architecture is about structure. But which structure? Software has more than one.

Parnas made this observation in 1976 ("On a 'Buzzword': 'Hierarchically Structured Systems'").

- Systems have many kinds of "pieces": programs, objects, classes, modules, processes, frameworks, tasks, threads…
- Each one defines a different structure.
- Which one is the architecture?

Answer: All of them might be.

Structures and Views

A representation of a structure (or a set of structures) is a *view*.

Modern treatments of architecture all recognize the importance of multiple architectural views.

Modern software systems are too complex to grasp all at once. At any moment, we restrict our attention to a small number of a software system's structures.

To communicate meaningfully about an architecture, we must make it clear which structure or structures we are discussing…that is, which **view** we are taking of the architecture.

Structures and Views - 2

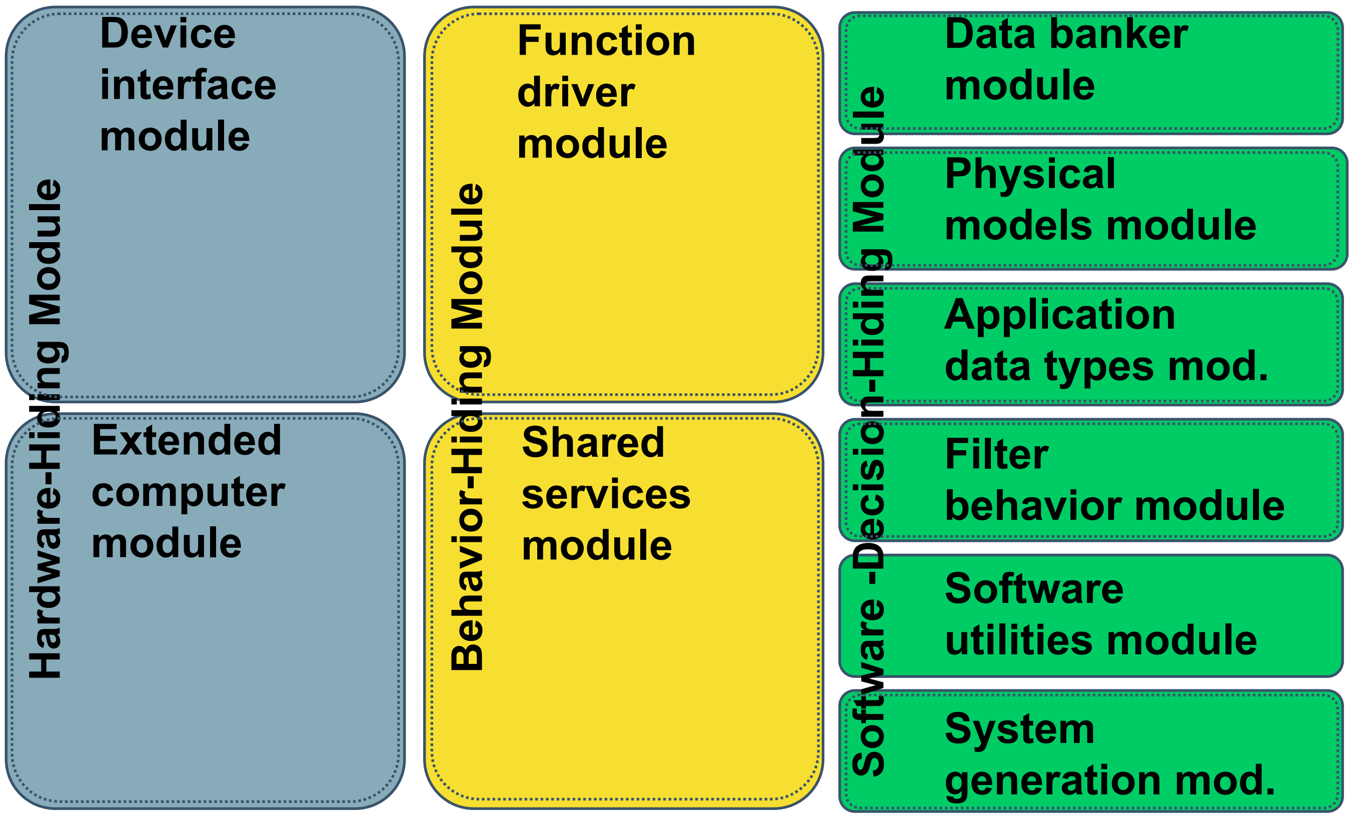
- *structure* – an actual set of architectural elements as they exist in software or hardware
- *view* – a representation of a coherent set of architectural elements, as written by and read by system stakeholders. A view represents a a set of elements and the relationships between those elements.

Example of Multiple Views

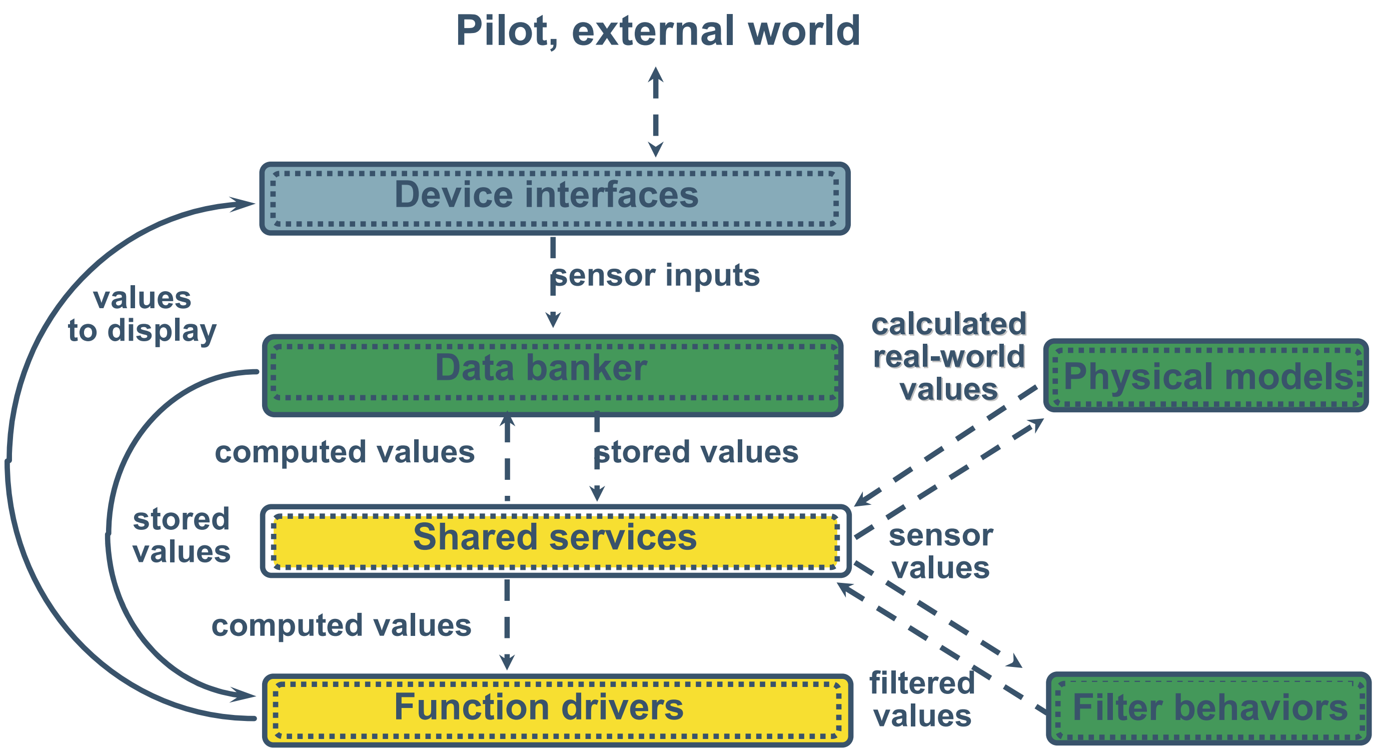
Software Architecture for A-7E Corsair II Aircraft

- U. S. carrier-based, light attack aircraft
- Used from the 1960s through the 1980s
- Small computer on board for navigation, weapons delivery

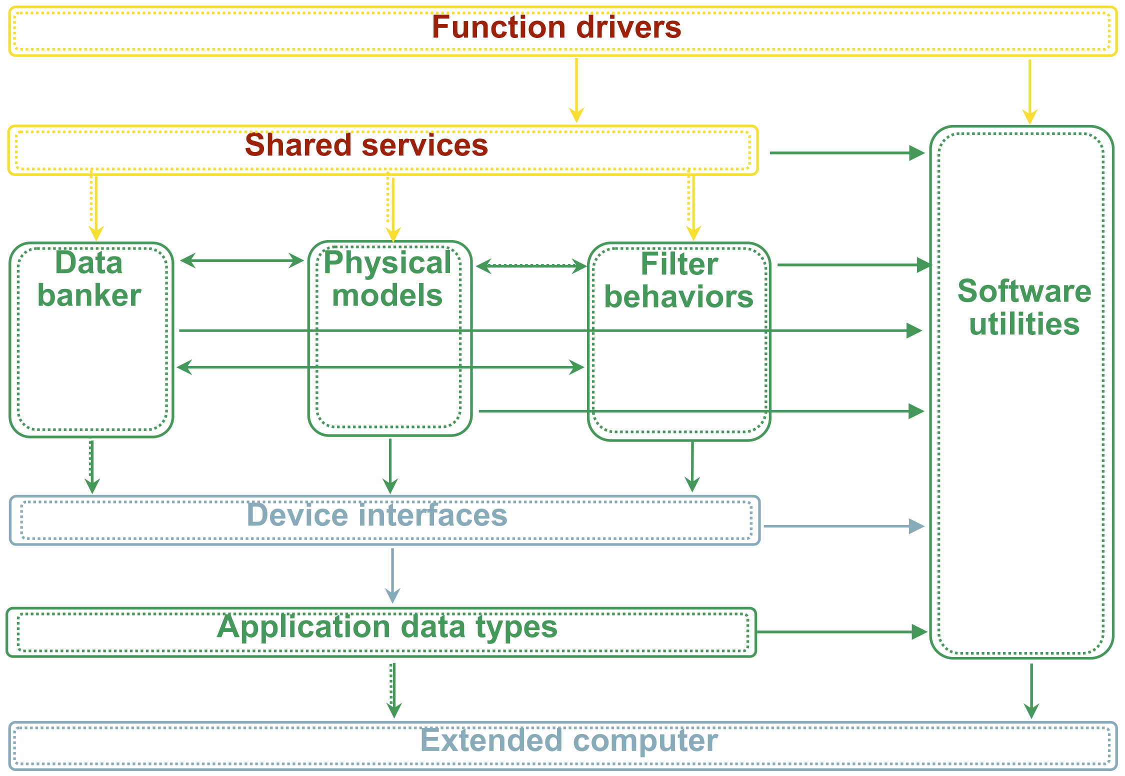
Module Decomposition View (2 Levels)



Data Flow View



Layers View


Function drivers
Shared services
Data banker
Physical models
Filter behaviors
Software utilities
Device interfaces
Application data types
Extended computer

Views -1

An architecture is a very complicated construct -- too complicated to be seen all at once.

Views are a way to manage complexity.

Each view can be used to answer a different question about the architecture

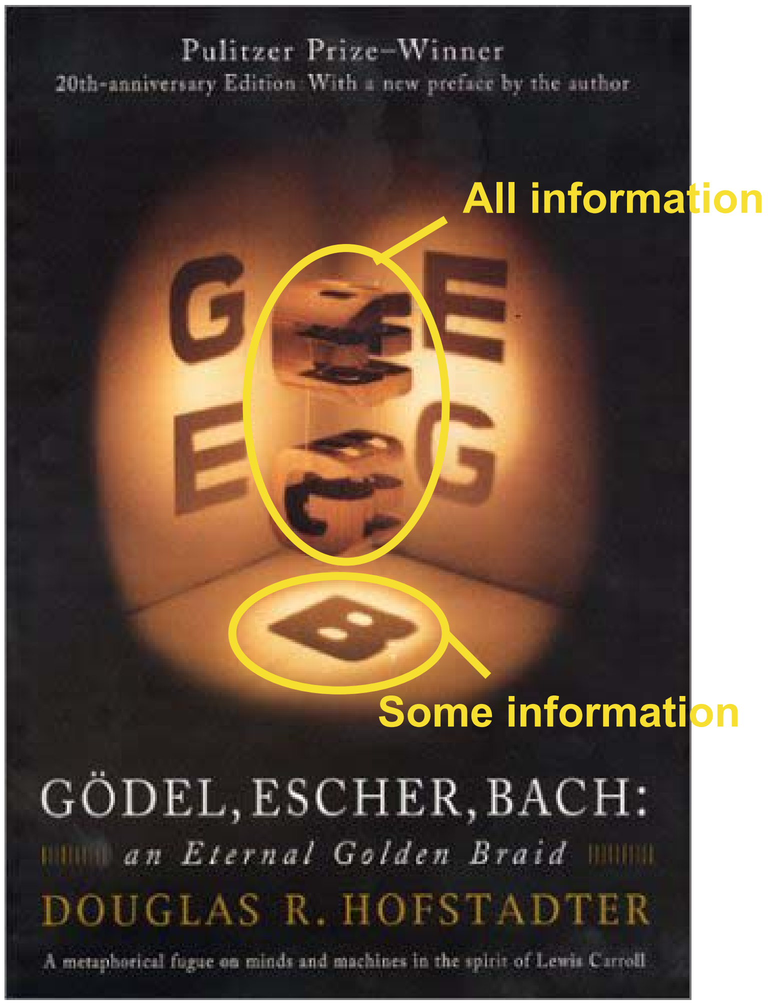
- What are the major execution units and data stores?
- What software is other software allowed to use?
- How does data flow through the system?
- How is the software deployed onto hardware?

Views -2

A view is a representation of a set of architectural elements and the relations associated with them.

Not *all* architectural elements -- *some* of them.

A view binds element *types* and relation *types* of interest, and shows those.


Pulitzer Prize-Winner
20th-anniversary Edition With a new preface by the author
All information
Some information
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll

Views -3

In the 1990s, the trend was to prescribe a set of views.

- Rational (Kruchten) 4+1 view model
- Siemens Four-Views Model for architecture
- Others

Now the trend is to prescribe choosing the *right set* of views from an open set of possibilities.

IEEE/ANSI 1471-2000 (“Recommended Practice for Architectural Description of Software-Intensive Systems”) exemplifies this approach.

More on this when we discuss documentation.

Architectural Structures

Architectural structures (and hence views) can be divided into three types:

1. **“module” structures** – consisting of elements that are units of implementation called *modules*
2. **“component-and-connector” structures** – consisting of runtime components (units of computation) and the connectors (communication paths) between them
3. **“allocation” structures** – consisting of software elements and their relationships to elements in external environments in which the software is created and executed

Example Module Structures

Decomposition structure – consisting of modules that are related via the *"is a submodule of"* relation

Uses structure – consisting of modules that are related via the *"uses"* relation (i.e., one module uses the services provided by another module)

Layered structure – consisting of modules that are partitioned into groups of related and coherent functionality. Each group represents one layer in the overall structure.

Class/generalization structure – consisting of modules called classes that are related via the *"inherits from"* or *"is an instance" of* relations

Example Component-and-Connector Structures

Process structure – consisting of processes or threads that are connected by communication, synchronization, and/or exclusion operations

Concurrency structure – consisting of components and connectors where connectors represent “logical threads”

Shared-data (repository) structure – consisting of components and connectors that create, store, and access persistent data

Client-server structure – consisting of cooperating clients and servers and the connectors between them (i.e., the protocols and messages they share)

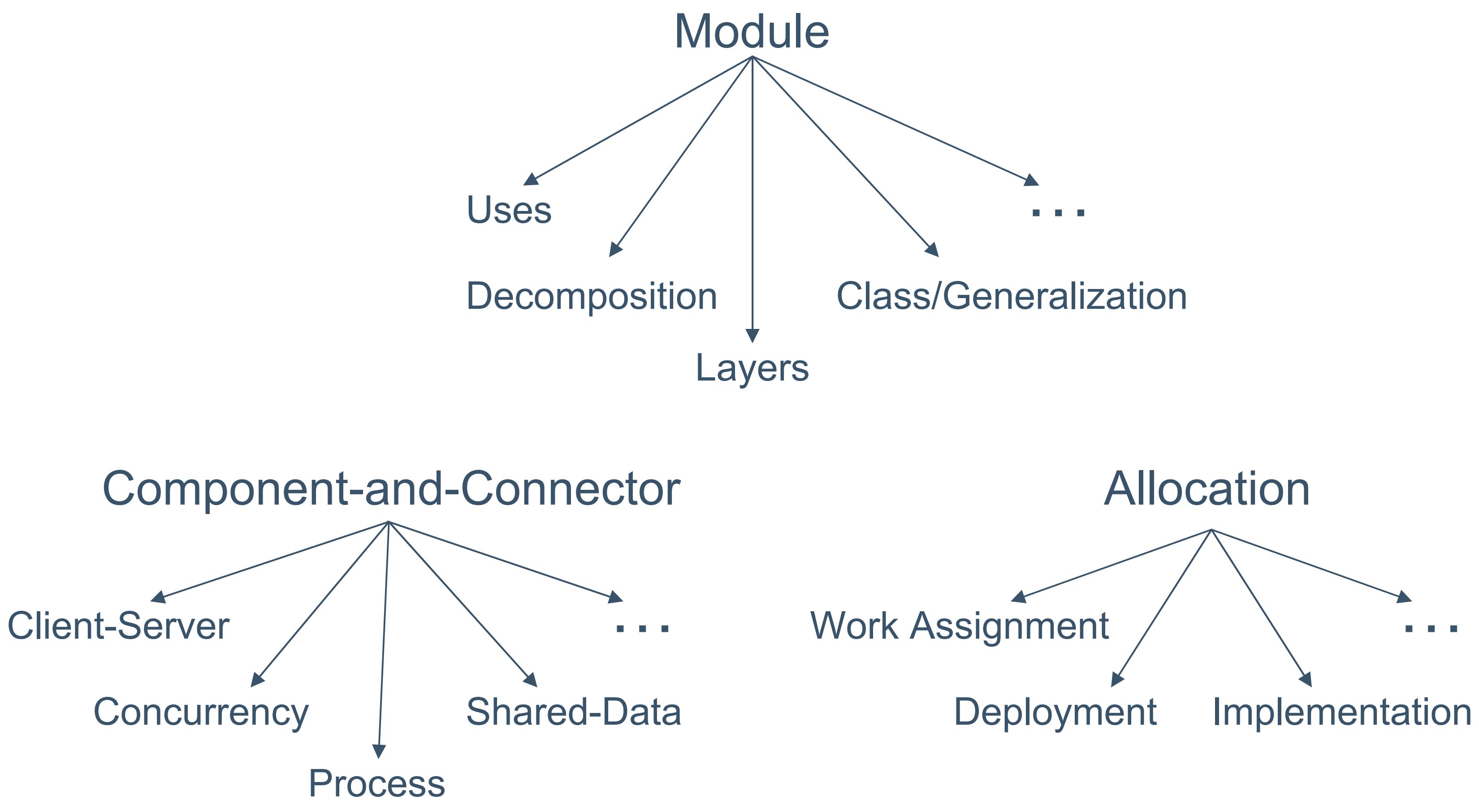
Example Allocation Structures

Deployment structure – consisting of software elements and their allocation to hardware and communication elements

Implementation structure – consisting of software elements and their mapping to file structures in the development, integration, and configuration control environments

Work assignment structure – consisting of modules and how they are assigned to the development teams responsible for implementing and integrating them

Architectural Structures Summary



Using structures and views

Each structure provides the architect with an *engineering handle* on some aspect of the system. For example:

- Carefully designing the module decomposition structure has a powerful effect on modifiability.
- Carefully designing the module “uses” structure has a powerful effect in the ability to field subsets and develop incrementally.
- Carefully designing the deployment structure has a powerful effect on performance and availability.
- Carefully designing the various C&C structures has a powerful effect on run-time QA’s such as performance or security.

Using structures and views

Architects choose the structures that need to engineer based on the important quality attribute drivers.

They record their designs using the corresponding views.

Other kinds of architecture

Since the ascendance of software architecture, other kinds of architecture have arisen. Two in particular are:

- enterprise architecture
- system architecture

What do these terms mean to you?

Enterprise Architectures

Enterprise architecture is a means for describing business structures and processes that connect business structures.[1]

- It describes the flow of information and activities between various groups within the enterprise that accomplish some overall business activity.
- Enterprise architectures may or may not be supported by computer systems.
- Software and its design are not typically addressed explicitly in an enterprise architecture.

1 Zachman, John A. "A Framework for Information Systems Architecture." *IBM Systems Journal 26*, 3 (1987): 276-292.

System Architecture

A system architecture is a means for describing the elements and interactions of a complete system including its hardware elements and its software elements.

System architecture is concerned with the elements of the system and their contribution toward the system’s goal, but not with their substructure.

See: Rechtin, E. *Systems Architecting: Creating and Building Complex Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Where Does Software Architecture Fit?

Enterprise architecture and system architecture provide an environment in which software lives.

- Both provide requirements and constraints to which software architecture must adhere.
- Elements of both are likely to contain software architecture.

Summary

Software architecture refers to the structures of systems: their elements, externally visible properties, and relationships among them.

Software architecture represents the earliest and farthest-reaching design decisions about a system.

Software architecture permits or precludes nearly every quality attribute for a system…

…which is what software engineering is about achieving.

Next:

We'll discuss where architectures come from:

- Understanding architectural requirements: Quality attributes, quality attribute scenarios, quality requirements elicitation and capture

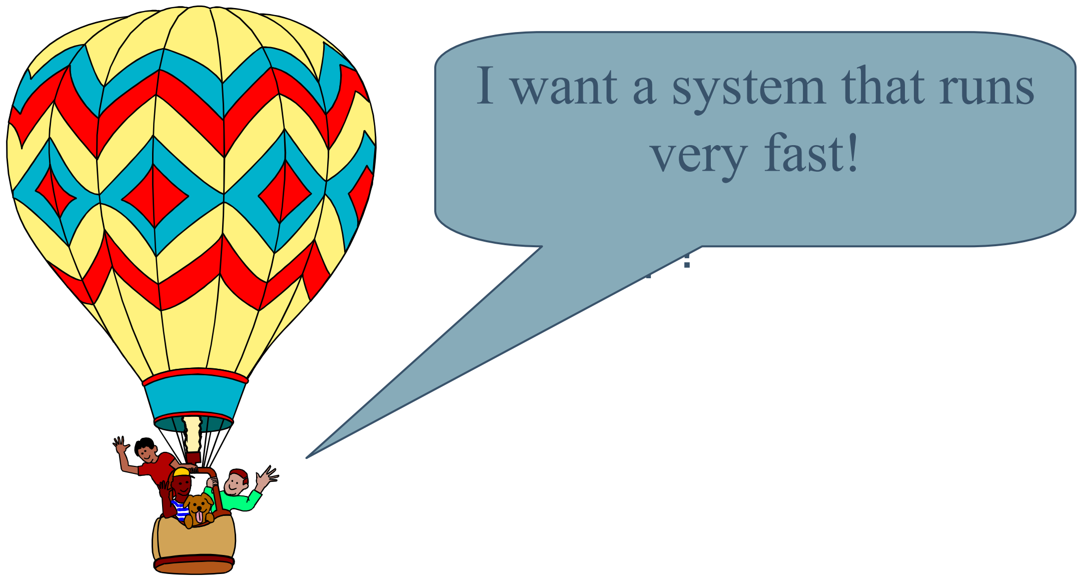
- The Architecture Business Cycle: forces that shape the architecture, and where they come from

Specifying quality attributes

If quality attributes are so important, we need a way to communicate them unambiguously.

Specifying quality attributes

Suppose our customer tells us he wants a system that runs very fast.



Specifying quality attributes

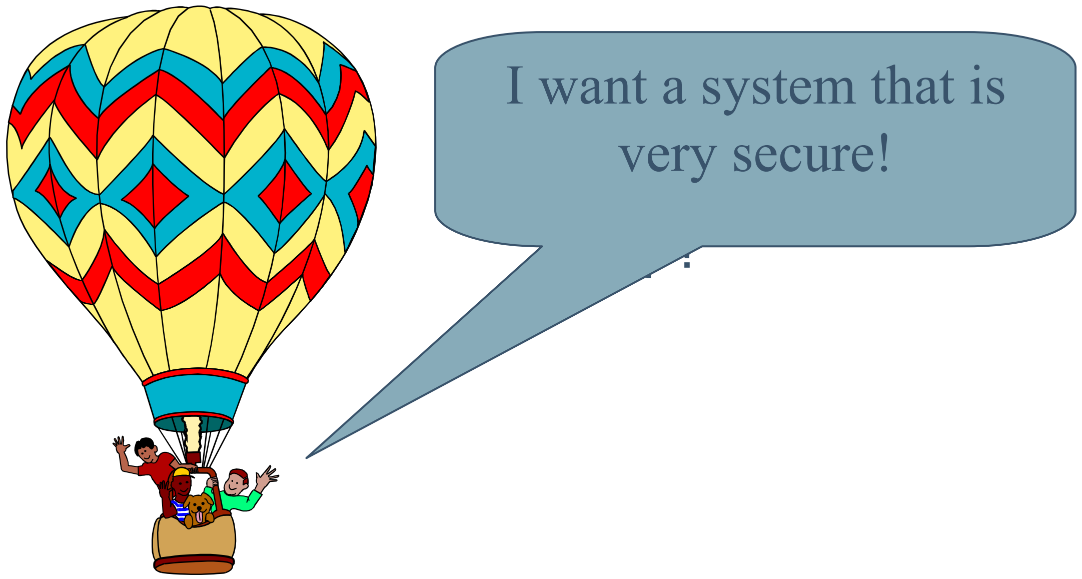
How fast?

How do you measure that?

You can’t everything run fast. What do you really care about?

Specifying quality attributes

Suppose our customer tells us he wants a system that is very secure.



Specifying quality attributes

How secure?

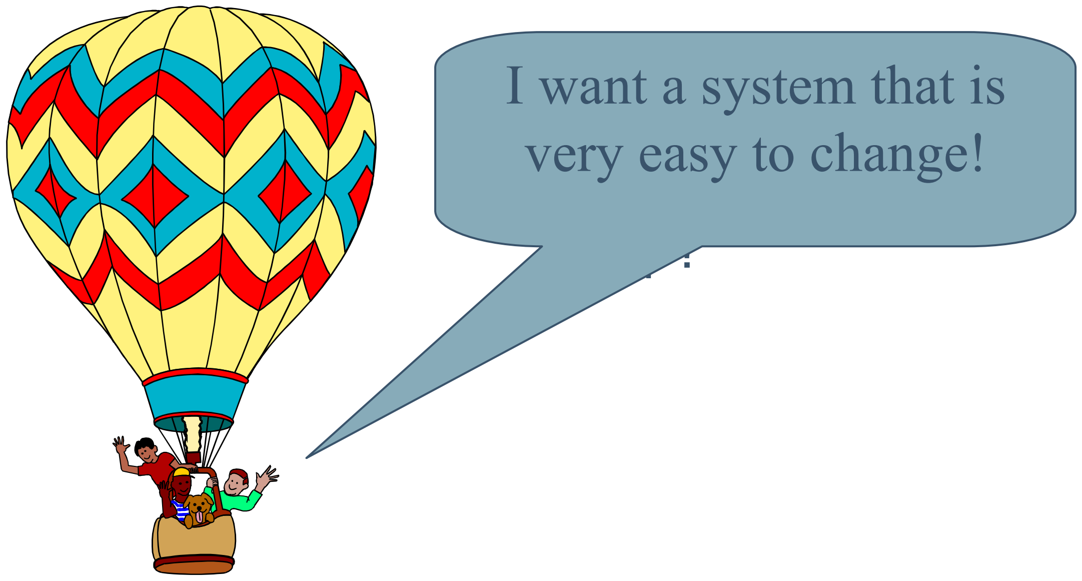
How do you measure that?

You can’t have totally secure software. What do you really care about, or what threats are the most important to guard against?

Carnegie Mellon
Software Engineering Institute

Specifying quality attributes

Suppose our customer tells us he wants a system that is very easy to change.



Specifying quality attributes

How easy?

How do you measure that?

You can’t make everything equally easy to change. Which changes do you really care about?

Specifying quality attributes

I want a system that…
Be quiet!

Specifying quality attributes

Conclusion: Just naming a quality attribute doesn't help very much.

We can't build software with just that.

We need to be more specific.

Most people use *quality attribute scenarios* to capture quality attributes.

Scenarios

A scenario is a little story describing an interaction between a stakeholder and a system.

A use case is a kind of scenario. The stakeholder is the user. The interaction is a functional use of the system.

“The user pushes this button, and this result occurs.”

Scenarios

We can generalize the notion of a use case to come up with *quality attribute scenarios*.

A quality attribute scenario is a short description of how a system is required to respond to some stimulus.

QA Scenarios

A quality attribute scenario has six parts:

- source – an entity that generates a stimulus
- stimulus – a condition that affects the system
- artifact – the part of that was stimulated by the stimulus
- environment – the condition under which the stimulus occurred
- response – the activity that results because of the stimulus
- response measure – the measure by which the system’s response will be evaluated

A QA Scenario for Availability

- *An unanticipated external message is received by a process during normal operation. The process informs the operator of the message's receipt, and the system continues to operate with no downtime.*

1. source – external
2. stimulus – unanticipated message received
3. artifact – process
4. environment – during normal operation
5. response – system continues to operate
6. response measure – zero downtime

A QA Scenario for Modifiability

- *During maintenance, a change is made to the system’s rules engine. The change is completed in one day.*

1. source – requestor of the change
2. stimulus – a change is made
3. artifact – rules engine
4. environment – during maintenance
5. response – the change is completed
6. response measure – …in one day

A QA Scenario for Security

- *During peak operation, an unauthorized intruder tries to download prohibited data via the system administrator's interface. The system detects the attempt, blocks access, and notifies authorities within 15 seconds.*

1. source – an unauthorized intruder
2. stimulus – tries to download prohibited data
3. artifact – system administrator's interface
4. environment – during peak operation
5. response – the attempt is detected, blocked, reported
6. response measure – …within 15 seconds

Scenarios for other QA’s

Can you imagine QA Scenarios for

- Usability?
- Testability?
- Time to market?
- Freedom from error?
- Others

Q: How many scenarios does it take to specify a quality attribute?

A: As many as you need.

One QA, many scenarios

For a system we're about to build:

We might capture several <u>performance</u> scenarios, one for each of:

- (Min, max, average) transaction throughput under (peak, normal) load
- (Min, max, average) end-to-end latency for a transaction

We might capture several <u>security</u> scenarios, one for each of:

- Denial of service
- Unauthorized access
- Non-repudiatability

One QA, many scenarios

For a system we're about to build:

We might capture several modifiability scenarios, one for each of:

- Adding a new function
- Correcting a bug
- Changing the platform or middleware
- Changing the behavior
- Replacing one component with another
- Changing the user interface
- Etc.

And so forth.

Exercise

Write a quality attribute scenario that expresses a requirement for

- Modifiability
- Security
- Usability
- Performance
- Testability

Take 30 minutes. Plan to read your results to the class.

More about QAs

There is no standard set of quality attributes

- People disagree on names: Maintainability/modifiability/portability
- People come up with new ones: “calibrate-ability”
- There is no standard meaning of what it means to be “secure”

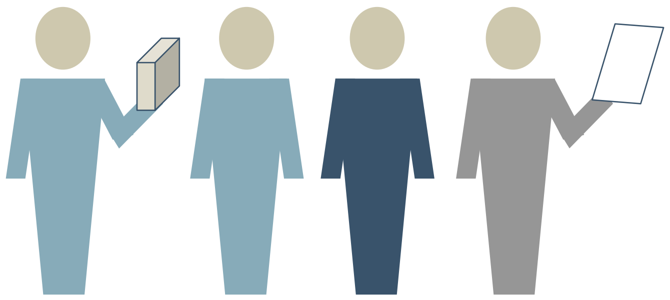
Scenarios let us avoid all of these problems!

The QAs are defined by the scenarios!

Who tells us what QA’s are important? **Stakeholders!**

Stakeholders

Stakeholders are people with a vested interest in the system. They are the people who can tell us what is needed. They are the people who can tell us if what we are building is the right thing.



We usually think of the user as telling us what is required, but there are many kinds of stakeholders.

Carnegie Mellon
Software Engineering Institute

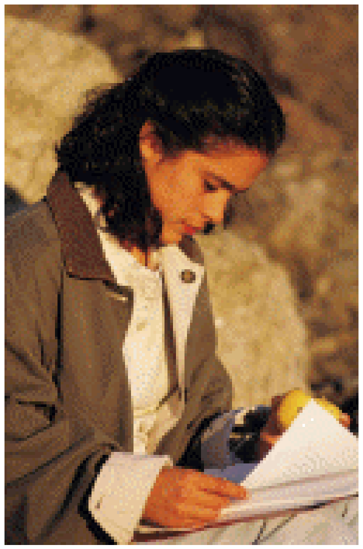
Stakeholders

Who are the stakeholders of an architecture?
Name some of the roles.

Influence of System Stakeholders

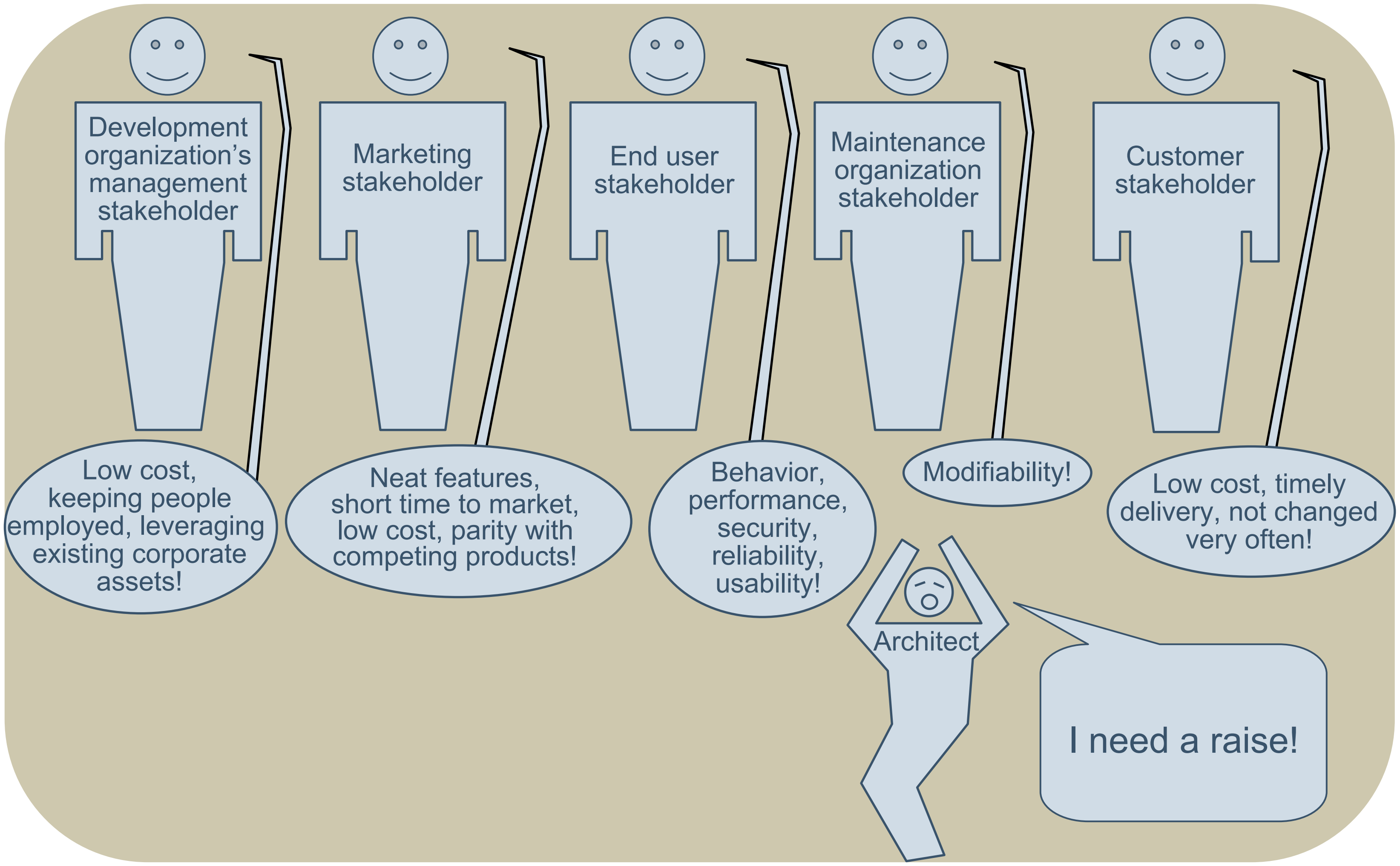
Stakeholders have an interest in the construction and operation of a software system. They might include:

- customers
- users
- developers
- project managers
- marketers
- maintainers



Stakeholders have different concerns that they want to guarantee and/or optimize.

Concerns of System Stakeholders


Development organization's management stakeholder
Marketing stakeholder
End user stakeholder
Maintenance organization stakeholder
Customer stakeholder
Low cost, keeping people employed, leveraging existing corporate assets!
Neat features, short time to market, low cost, parity with competing products!
Behavior, performance, security, reliability, usability!
Modifiability!
Low cost, timely delivery, not changed very often!
Architect
I need a raise!

Stakeholder Involvement

Stakeholders' quality attribute requirements are seldom documented, which results in

- goals not being achieved
- conflict between stakeholders

Architects must identify and actively engage stakeholders early in the life cycle to

- understand the real constraints of the system (many times, stakeholders ask for everything!)
- manage the stakeholders' expectations (they can't have everything!)
- negotiate the system's priorities
- make tradeoffs

SEI Quality Attribute Workshop (QAW)

The QAW is a facilitated method that engages system stakeholders early in the life cycle to discover the driving quality attributes of a software-intensive system.

Key points about the QAW are that it is

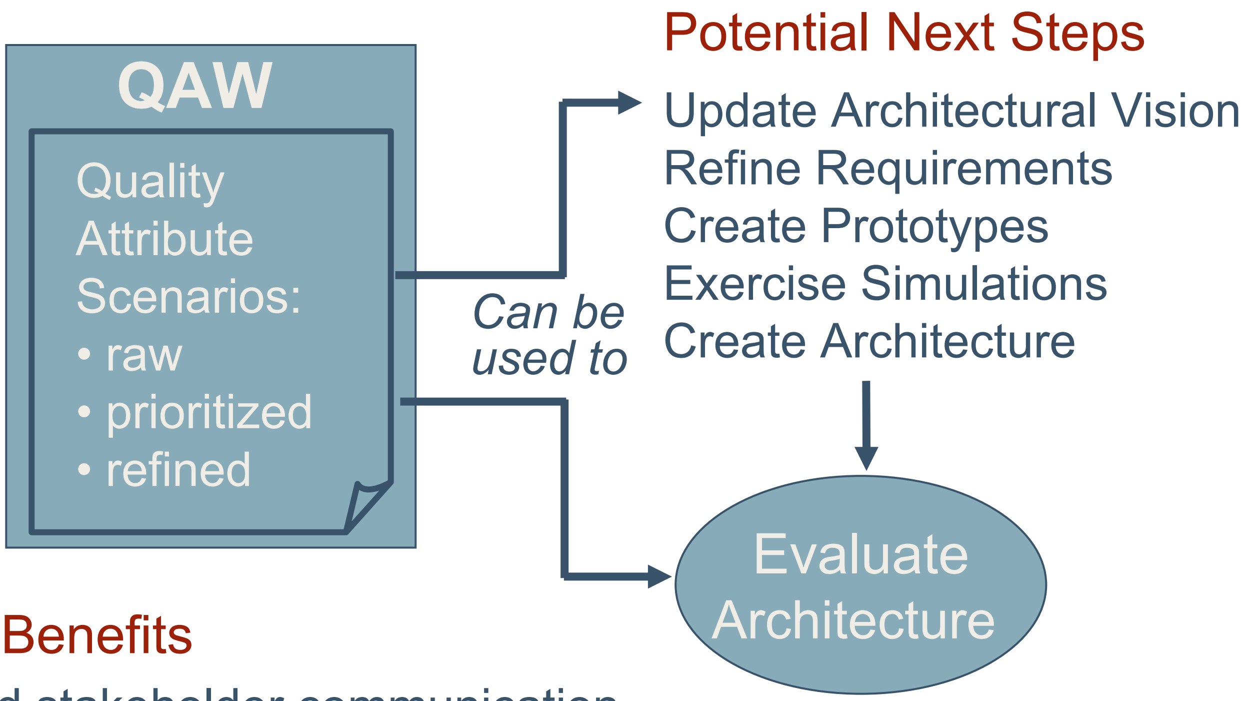
- system-centric
- stakeholder focused
- used before the software architecture has been created

QAW Steps

1. QAW Presentation and Introductions
2. Business/Mission Presentation
3. Architectural Plan Presentation
4. Identification of Architectural Drivers
5. Scenario Brainstorming
6. Scenario Consolidation
7. Scenario Prioritization
8. Scenario Refinement

Iterate as necessary with broader stakeholder community

QAW Benefits and Next Steps



Potential Benefits

- increased stakeholder communication
- clarified quality attribute requirements
- informed basis for architectural decisions

Architectural Requirements

Architectural requirements are shaped by quality attribute requirements.

These come from stakeholders.

What else shapes an architecture?

Development Organization's Influence on Architectures

- An organization may have an investment in certain assets, such as
  - existing architectures and products based on them.
  - a purchased tool environment
  - training
- The architecture can form the core of a long-term investment to meet the strategic goals.
- The organizational structure can shape the architecture. E.g., a Database Division may influence the architect to include a database in the design.

Influence of Technical Environment on Architectures

The technical environment that is current when an architecture is designed will influence that architecture.

Today, a business system will almost certainly be

- Web-based
- Have a main database
- Be layered and/or tiered
- Be distributed and use commercial middleware
- Etc.

It may also use

- Agents
- Service-oriented architecture
- .NET or J2EE or…

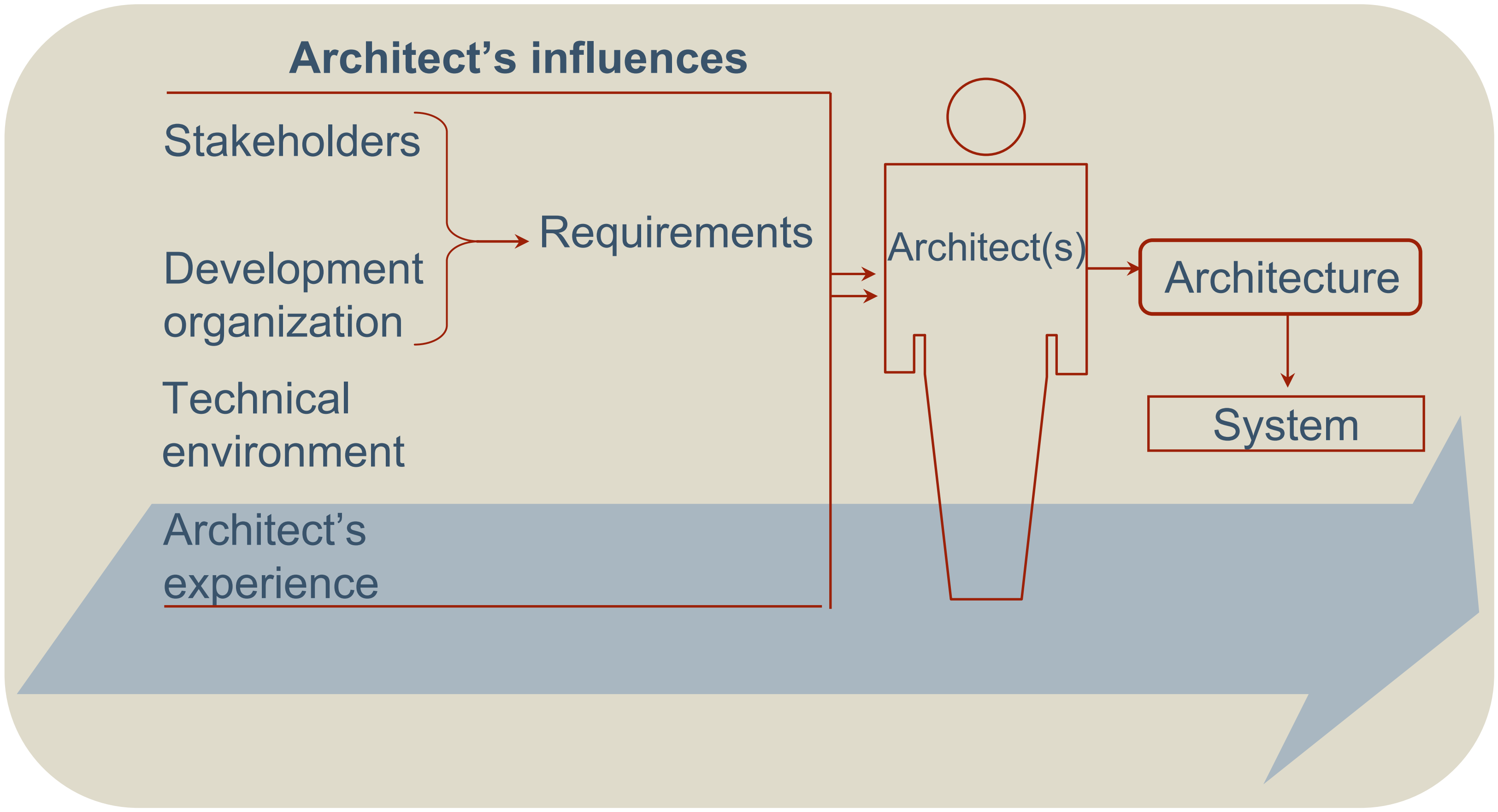
It wasn’t always like this.

Influence of Architect's Background on Architectures

Architects make choices based on their past experiences:

- Good experiences will lead to the replication of those prior designs that worked well.
- Bad experiences will be avoided in new designs, even if the methods, techniques, and/or technology that led to those bad experiences might work better in subsequent designs.
- An architect's choices might be influenced by education and training.

Influences on the Architecture



Architectures Affect the Factors That Influence Them

Once the architecture is created and a system (or systems) built from it, both will affect

- the structure and goals of the organization developing them
- customers' requirements
- the architect's experience in developing subsequent systems because the corporate experience base has been enhanced
- technology

How Architectures Affect the Organization – 1

Architectures can influence the structure of the organization developing them.

Architectures prescribe the units of software that must be implemented and integrated.

In turn, software units are the basis for

- team formation
- development, test, and integration activities
- resource allocation in schedules and budgets

How Architectures Affect the Organization – 2

Architectures can influence the goals of an organization.

The architecture can provide opportunities for the efficient production and deployment of similar systems.

The organization might adjust its goals to take advantage of new market opportunities based on its architecture-enabled capability.

How Architectures Affect Customers' Requirements

Architectures can influence customers' requirements:

- Knowledge of similarly fielded systems leads customers to ask for particular kinds of features.

  They may even ask for systems using language of the architecture: client-server, .NET, service-oriented, etc.

- Customers will alter their system requirements based on the availability of existing systems and components.

  They often save time and money this way.

How Architectures Affect the Architect's Experience

The process of building systems influences the architect's experience base. This, in turn, influences how subsequent systems in the organization are constructed:

- Successful systems built around a technology, tool, or method will engender future systems that are built in the same way.
- The architecture for a failed system is less likely to be chosen for future projects.

How Architectures Affect Technology -1

Occasionally, a system or architecture will actually change the software engineering technical environment.

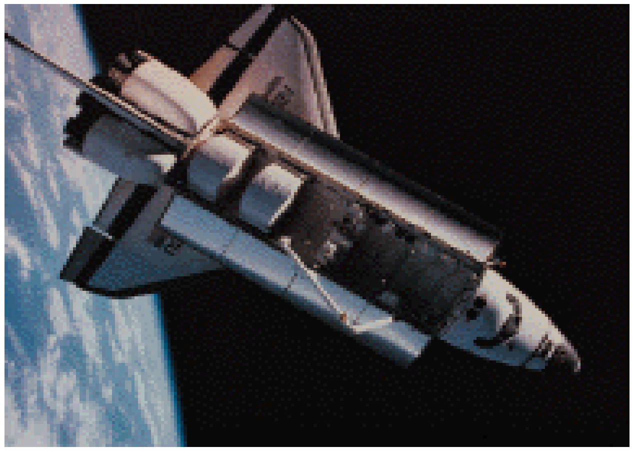
There was a “first time” for all of these architectures:

- Layered (Dijkstra, 1968)
- N-tier client-server
- Service-oriented architectures
- Java / EJB / J2EE
- Object-oriented

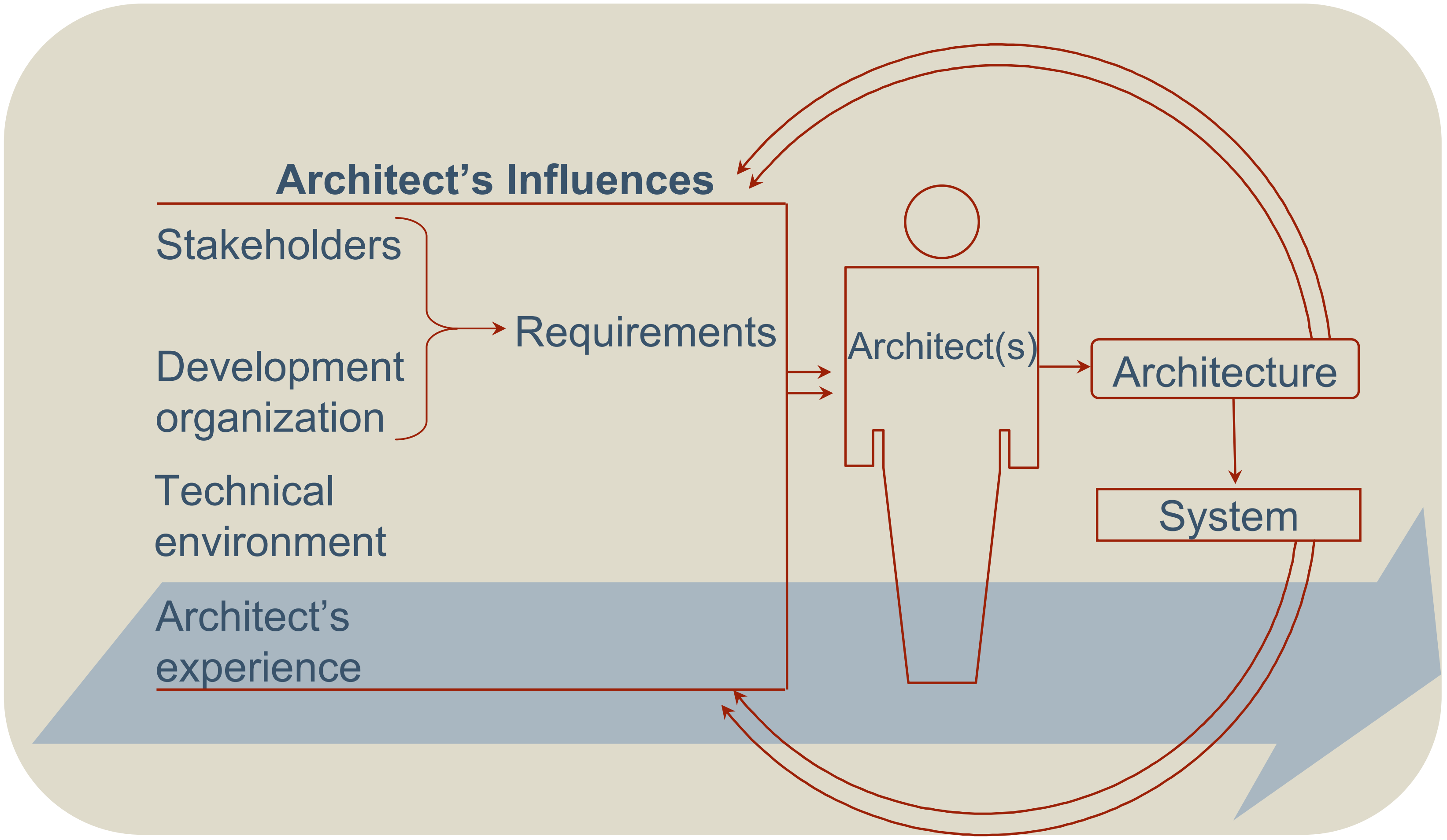
How Architectures Affect Technology -2

Also, applications that were very successful “donate” their architectures into the technical environment:

- Large relational databases and systems that use them
- Web-based e-commerce systems
- The World Wide Web itself
- “Standard” avionics or “vetronics” architectures
- Compilers and compiler-compilers



Architecture Business Cycle (ABC)



How to use the ABC

Architects must recognize all of the ways that architectures are influenced.

- Engage stakeholders
- Understand the goals of their organization
- Learn the current technical environment
- Be aware of their own experiences

Management should recognize the ways in which an architecture can (or should be allowed) to influence the organization.

- New market opportunities
- New ways to engage customers
- New organizational structures aligned with architecture

Many paths through the cycle

Sometimes systems traverse the cycle many times.

Example: World Wide Web

Early version of web requirements produced one architecture for clients and servers (LibWWW).

Success of that architecture led to explosive growth, which influenced the stakeholders to want even more features.

This led to the current architectures for web-based applications, which are quite different.

A Picture of Architecture-Based Development -1

Development organizations who use architecture as a fundamental part of their way of doing business often define an architecture-based development process.

This seminar series will illuminate the usual parts of that process.

Typically, the first few steps are

- Analyze the business case
- Understand the architecturally significant requirements

A Picture of Architecture-Based Development -2

1. Analyze the business case
   - The business case will tell you why we're building the system, and why the customer is buying it.
   - The business case will start to reveal the driving QA requirements
   - No formal method for analyzing; architect uses experience

2. Understand the architecturally significant requirements
   - Not all requirements have an equal impact on the architecture.
   - Usually, QA requirements have the most impact.
   - Capturing those QA requirements is critical

A Picture of Architecture-Based Development -3

We now have tools in hand to carry out these steps.

- Architecture Business Cycle (ABC) – helps us identify business case factors that will shape the architecture
- Quality Attribute Workshop (QAW) – first way to engage the stakeholders.
- QA scenarios – the way to capture QA requirements.

Summary

Architectures are shaped by quality attributes.

We need help capturing and expressing quality attributes.
Scenarios help.

Quality attributes come from stakeholders.

But other influences are at work also:

- Developing organization
- Technical environment
- Architect's experience

There is a cycle of influences.

Topics

How to create an architecture:

- Designing architectures
- Patterns, styles, and tactics

Review

Each structure provides the architect with an *engineering handle* on some aspect of the system. Architects choose the structures they need to engineer based on the important quality attribute drivers.

Architectures are documenting by capturing *views*: A view is a representation of a set of architectural elements and the relations associated with them.

Review

We need help capturing and expressing quality attributes. Quality Attribute *scenarios* help.

Quality attributes come from stakeholders. Use a Quality Attribute Workshop to elicit them.

Other influences on the architecture are at work also:

- Developing organization
- Technical environment
- Architect’s experience

The architect must recognize and capture these.

Organizations must recognize that an architecture can influence these very factors: An *Architecture Business Cycle* exists.

Creating the Architecture

How does the architect create an architecture? (Multiple choice):

a. By re-using approaches from other architectures

b. By inventing new approaches out of thin air

c. By magic

Carnegie Mellon
Software Engineering Institute

Creating the architecture

Architects primarily work by using previously-tried solutions

- Large scale: Patterns and styles
- Small scale: Tactics

Styles, patterns, and tactics represent conceptual tools in the architect’s “tool bag.”

Professional architects always keep their tool bag up to date.

Patterns and styles

The modern term is “patterns” but early papers on software architecture wrote about “software architecture styles.”

Styles in architecture were analogous to styles in houses:

- Victorian (multi-story, lots of frilly wood decorations, tall windows, basically square footprint…)
- Colonial (brick front, pillars or columns, usually symmetrical front…)
- Ranch (single-story, sprawling, not very decorated…)

Patterns and styles

Authors such as Shaw and Garlan wrote "style catalogs"

Independent component patterns

- communicating-processes
- event systems
  - -implicit invocation
  - -explicit invocation

Data flow patterns

- batch sequential
- pipe-and-filter
- **layers**

Data-centered patterns

- blackboard
- repository

Virtual machine patterns

- interpreters
- rule-based systems

Call-return patterns

- main program and subroutine
- object oriented

Styles → Patterns

Then, the design patterns community arrived.

Architectural styles were clearly just patterns, whose scope of design was the whole system – that is, whose scope was the architecture.

Now, *architectural patterns* is the term in use.

There are books of architectural patterns, e.g.,

- Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley.

Architectural patterns

These are broadly-scoped solutions to previously encountered problems.

An architectural pattern

- is found repeatedly in practice
- is a package of design decisions
- has known properties that permit reuse
- describes a class of architectures

Architectural patterns

A pattern is determined and described by

- a set of element types
  - for example, data repositories, processes, and objects
- a set of interaction mechanisms or connectors
  - for example, subroutine calls, events, and pipes
- a topological layout of the components
- a set of semantic constraints covering topology, element behavior, and interaction mechanisms

Architectural patterns

These are widely known and include many familiar design approaches:

- Layered
- Pipe-and-filter
- Client-server
  - Thin client
  - Thick client
  - Asynchronous
  - Synchronous
  - N-tier client-server
  - Etc.
- Peer-to-peer
- Agent-based systems
- Service-oriented architectures
- Etc.

Observe:

- No "universal" list
- Patterns can be combined: e.g., layered client-server
- Patterns can be specialized
- Choice of patterns to use is not random!

Architectural patterns

These are widely known and include many familiar design approaches:

- Layered
- Pipe-and-filter
- Client-server
  - Thin client
  - Thick client
  - Asynchronous
  - Synchronous
  - N-tier client-server
  - Etc.
- Peer-to-peer
- Agent-based systems
- Service-oriented architectures
- Etc.

A pattern is determined by

- **a set of element types**
- **a set of interaction mechanisms or connectors**
- **a topological layout of the components**
- **a set of semantic constraints for topology, element behavior, and interaction mechanisms**

In addition, a pattern is described by

- **when and why to use it**

Patterns are coarse-grained solutions

While there are dozens (hundreds?) of patterns, there are thousands of design problems.

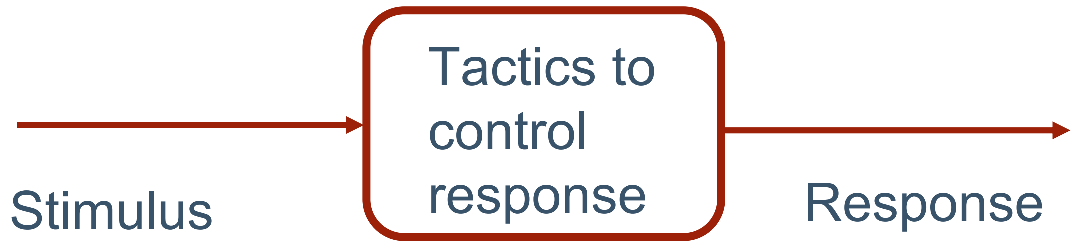
Expecting a complete list of patterns is not realistic.

What if we can’t find a pattern to solve our problem?

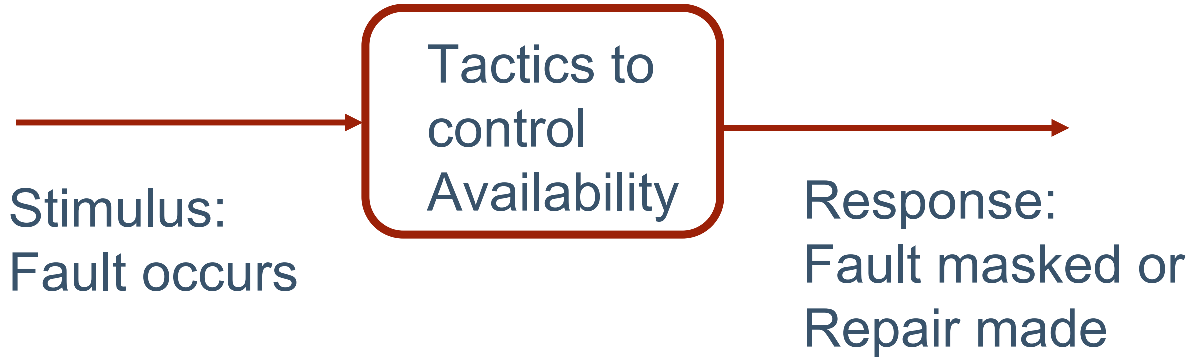
Tactics

An architectural *tactic* is a fine-grained design approach used to achieve a quality attribute response.

Tactics are the “building blocks” of design from which architectural patterns are created.



Tactics for Availability



Availability Tactics – 1

Fault detection

- ping/echo: when one component issues a ping and expects to receive an echo within a predefined time from another component
- heartbeat: when one component issues a message periodically while another listens for it
- exceptions: using exception mechanisms to raise faults when an error occurs

Availability Tactics – 2

Fault recovery

- voting: when processes take equivalent input and compute output values that are sent to a voter
- active redundancy: when redundant components are used to respond to events in parallel
- passive redundancy: when a primary component responds to events and informs standby components of the state updates they must make
- spare: when a standby computing platform is configured to replace failed components

Availability Tactics – 3

Fault recovery and reintroduction

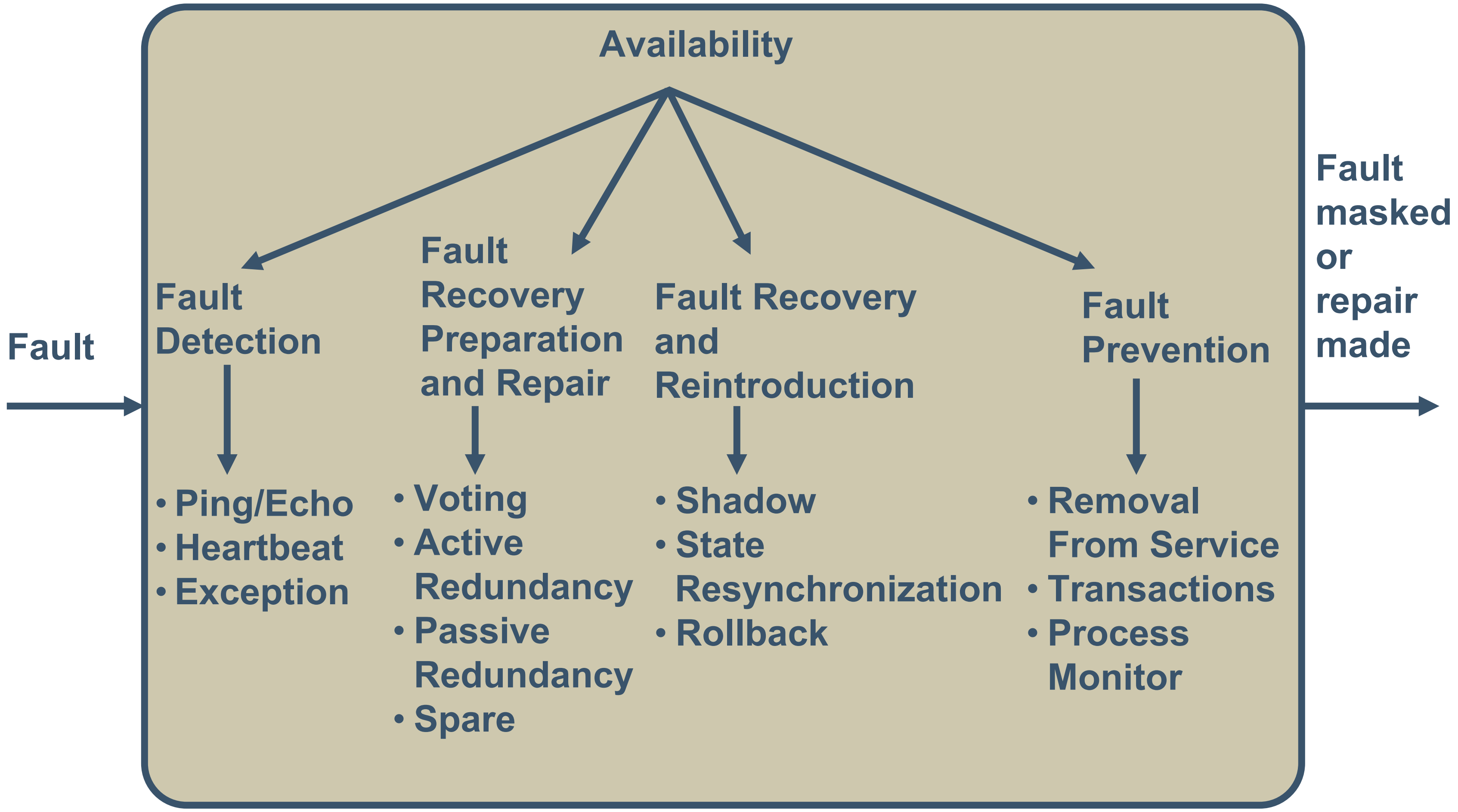
- shadow operation: running a previously failed component in “shadow mode” before it is returned to service
- state resynchronization: saving a state periodically and then using it to resynchronize failed components
- checkpoint/rollback: recording a consistent state that is created periodically or in response to specific events

Availability Tactics – 4

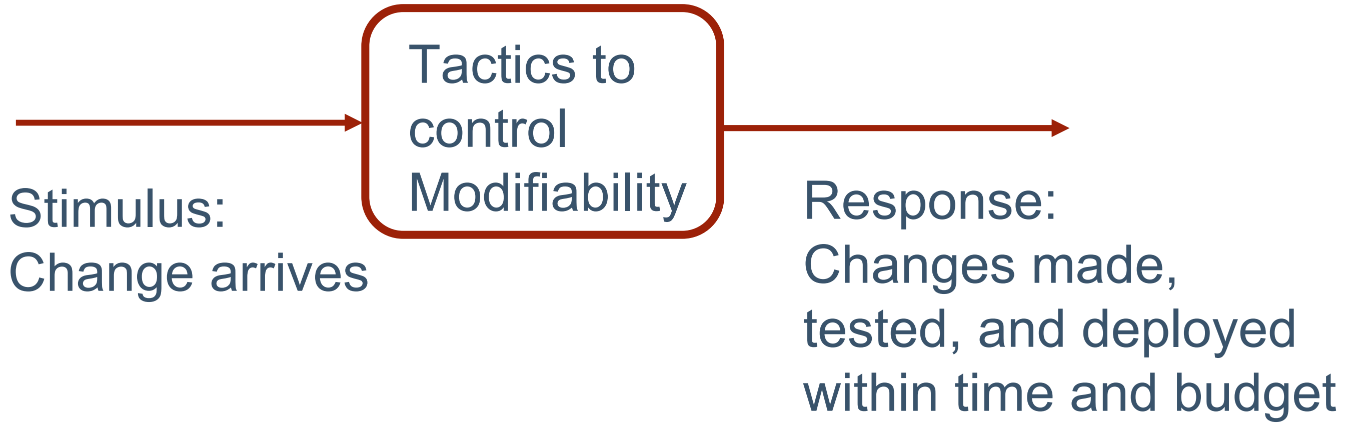
Fault prevention

- removal from service: removing a system component from operation so it can undergo some procedure that will help it avoid failure in the future (e.g., rebooting a component prevents failures caused by memory leaks)
- transactions: the bundling of several sequential steps such that the entire bundle can be undone at once
  - prevents data from being affected if one step in a process fails
  - prevents simultaneous access to data by concurrent threads
- process monitor: Monitoring processes are used to monitor critical components, remove them from service. and re-instantiate new processes in their place.

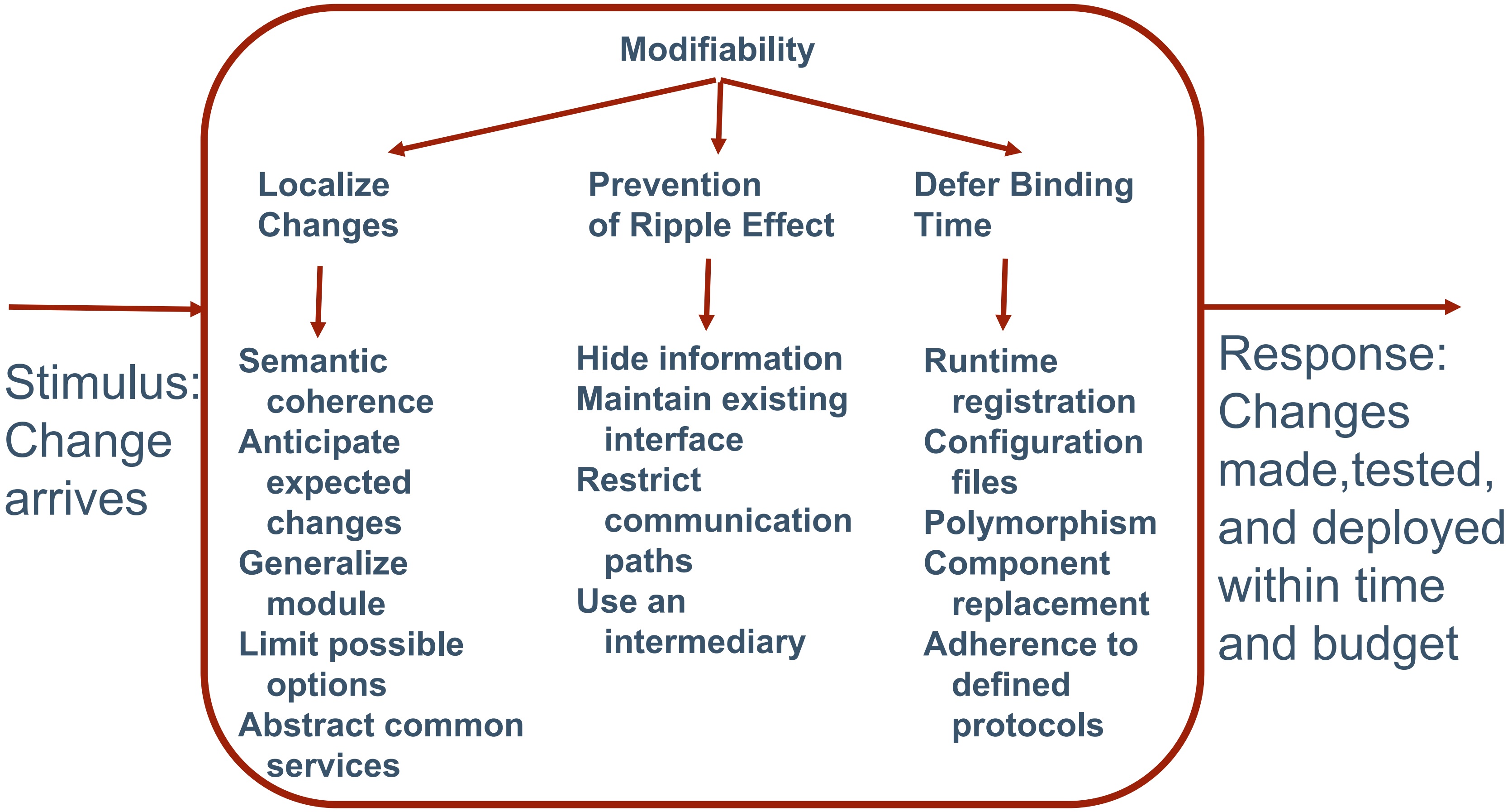
Summary of Availability Tactics



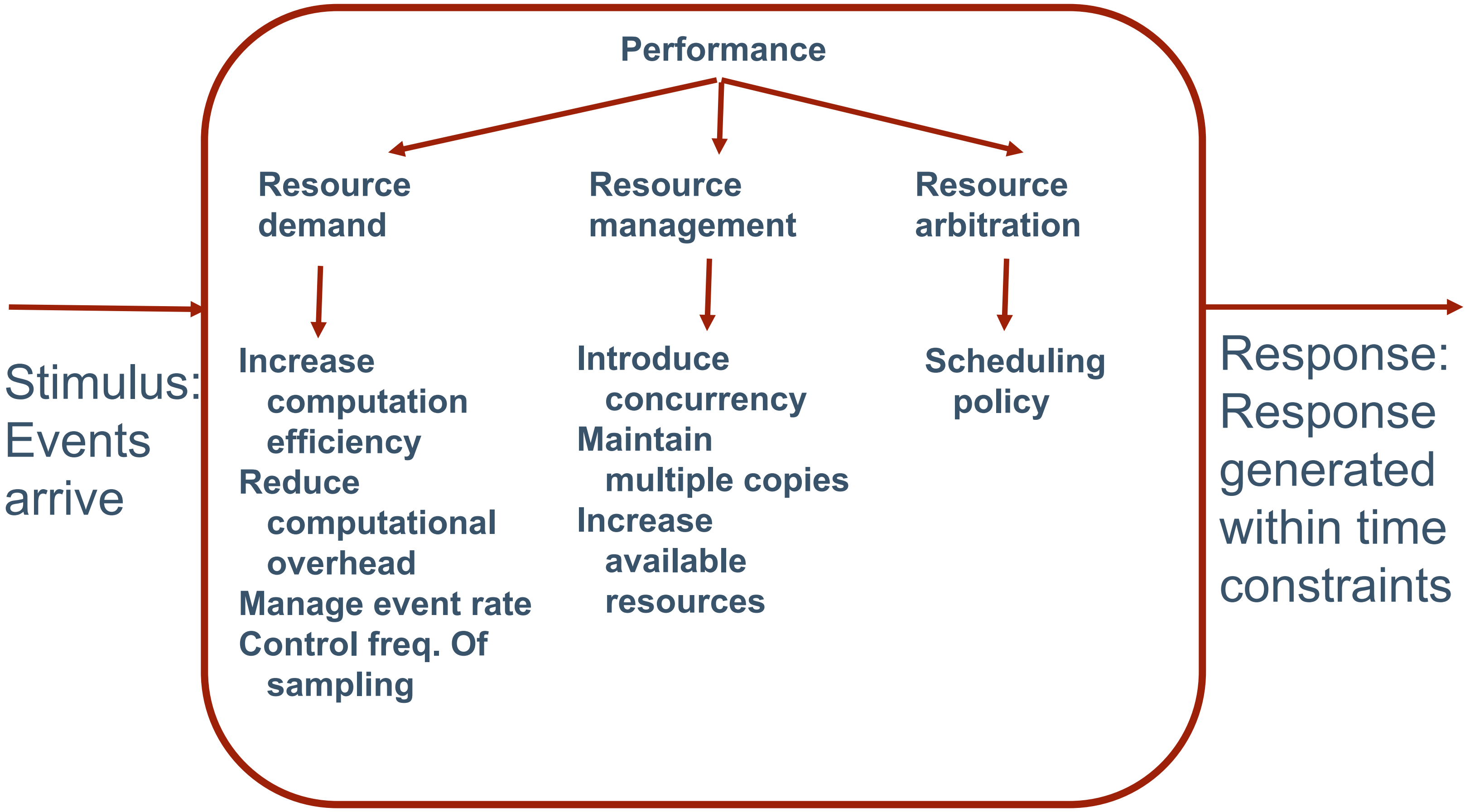
Tactics for Modifiability


Stimulus:
Change arrives
Tactics to control Modifiability
Response:
Changes made, tested, and deployed within time and budget

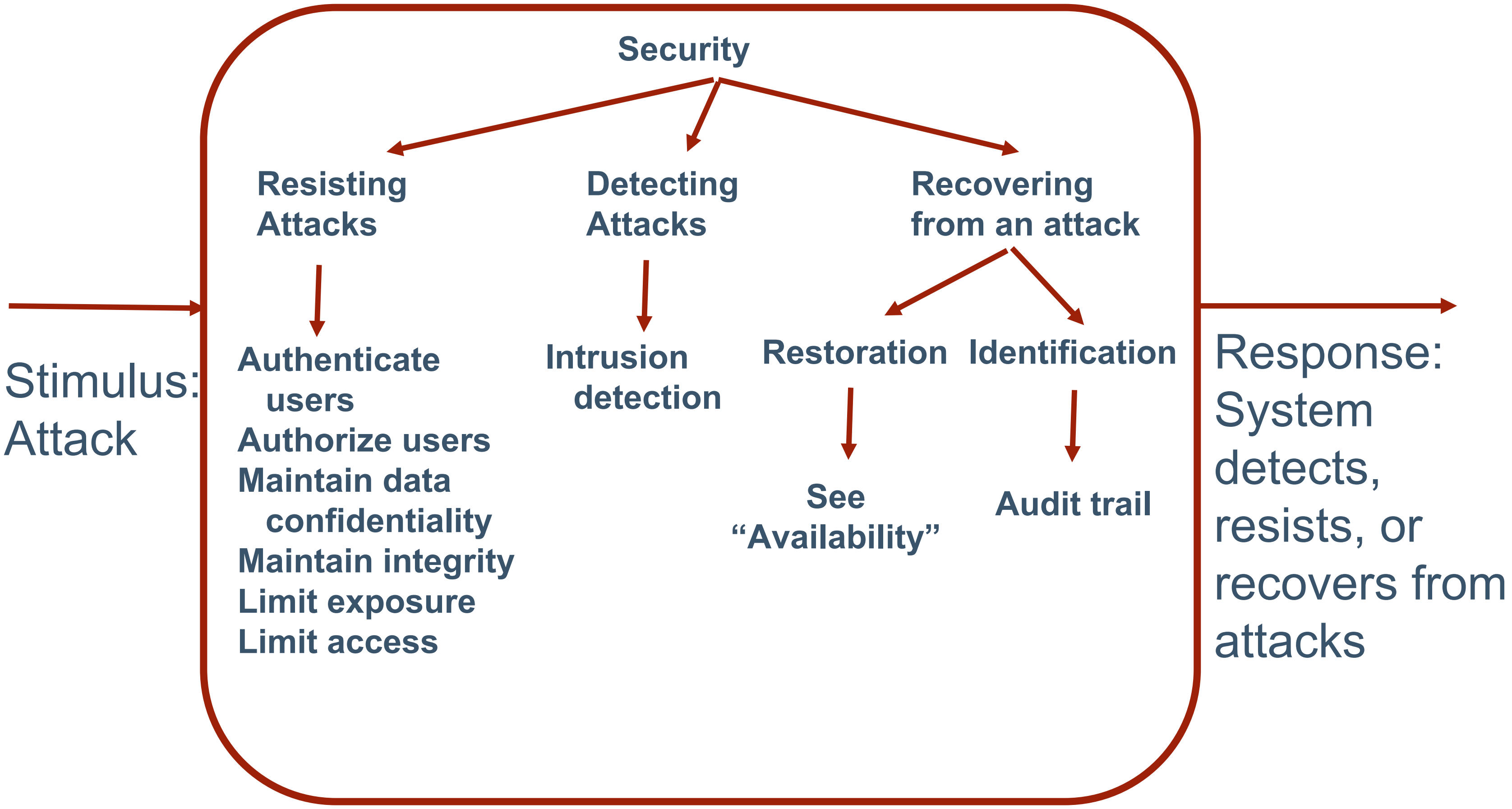
Summary of Modifiability Tactics



Tactics for Performance

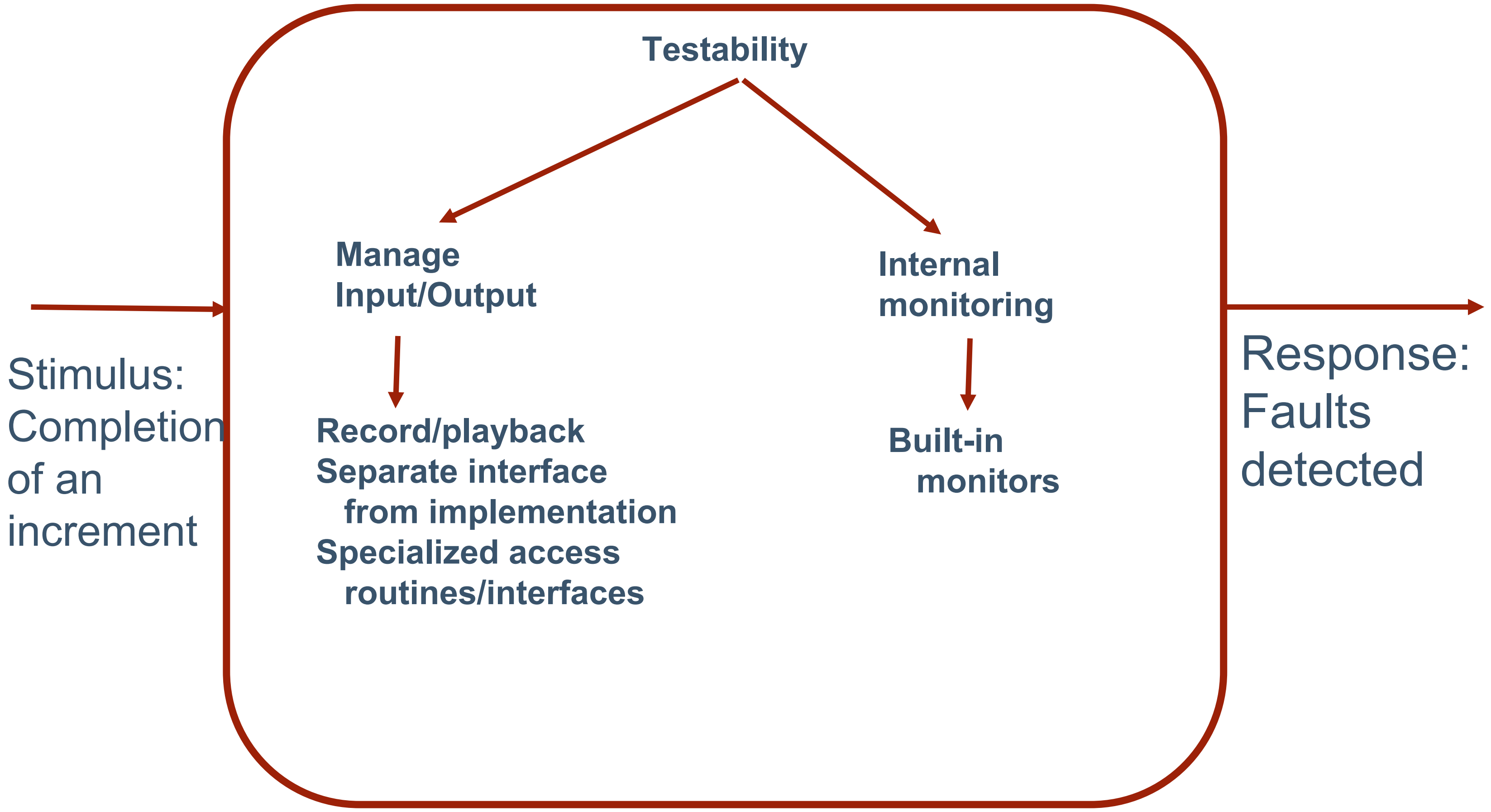


Tactics for Security



Carnegie Mellon
Software Engineering Institute

Tactics for Testability


Testability
Manage Input/Output
Internal monitoring
Record/playback
Separate interface from implementation
Specialized access routines/interfaces
Built-in monitors
Stimulus: Completion of an increment
Response: Faults detected

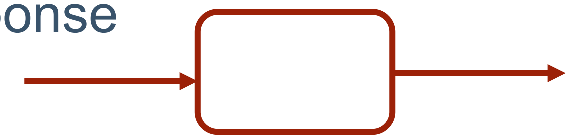
Tactics for other QAs

Tactics exist for other QA's as well.

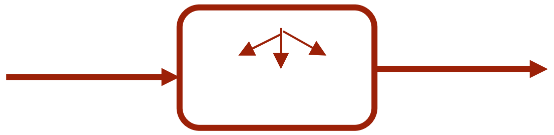
To catalog tactics for a QA.

1. Begin with a *general scenario* for the QA of interest.

2. Capture stimulus and the response



3. Capture the broad approaches



4. Fill in specific design approaches for each

Exercise

Create a list of tactics that promote usability.

Work in teams if you wish.

Tools – and how to use them

Tactics round out an architect's bag of tools.
- Patterns are the large-grained solution tools.
- Tactics fill in the gaps.

But tools aren't enough. An architect – like a carpenter -- has to know how to use the tools to build something.

Architecture – like carpentry – is more than a matter of bringing some tool out of the bag and using it on the problem.
- A hammer is not the best tool for cleaning glass.

A method for using the tools would be very helpful.

Attribute-Driven Design (ADD) Method

ADD is a step-by-step method for systematically producing the first architectural designs for a system.

ADD results

- Overall structuring decisions
- Interconnection and coordination mechanisms
- Application of patterns and tactics to specific parts of architecture
- Explicit achievement of quality attribute requirements
- NOT detailed interfaces

ADD requires as input:

- Quality attribute requirements
- Functional requirements
- Constraints

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 2: Choose Part of the System to Decompose – 1

ADD is a decomposition method:

- Just starting out? Then the “part” is the whole system
- Otherwise, choose a part identified from an earlier iteration

All required inputs *for the part you choose to decompose* should be available. They include

- functional requirements
- quality attribute requirements
- constraints

Step 2: Choose Part of the System to Decompose – 2

How to choose? It might depend on

- Risk. Design the high-risk pieces first.
- Progress and hand-off. Design the low-risk (i.e., simple) pieces quickly, to begin implementation.
- Importance. Design the important pieces (in terms of business context) first.
- Depth first. Choose a part of the system and “drive” its design to completion
- Breadth first. Make sure there are no major unknowns lurking at the high levels.
- Prototype building. Design enough (and in the right areas) to build a prototype early on.

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 3: Prioritize requirements and identify architectural drivers

Some requirements are more influential than others in the architecture and the decomposition of each module.

Influential requirements can be

- functional (e.g., training crews in flight simulator)
- quality attribute related (e.g., high security)
- business oriented (e.g., product line)

Architectural drivers are the combination of functional, quality attribute, and business requirements that “shape” the architecture or the particular module under consideration.

Step 3: Prioritize requirements and identify architectural drivers

To identify the key architectural drivers

- Locate the quality attribute scenarios that reflect the highest priority business goals relative to the module.
- Locate the quality attribute scenarios that have the most impact on the decomposition of the module.

Try to keep the number of architectural drivers to five or less.

Prioritize the architectural drivers.

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

The goal of this step is to establish an overall architectural approach that satisfies the architectural drivers.

- Start by trying to apply an architectural pattern.
  - E.g. client-server
- If necessary, apply a combination of patterns.
  - E.g., layered client-server
- If necessary, augment the pattern(s) with tactics.
  - E.g., layered client-server with ping-echo interaction

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 5: Instantiate architectural elements and allocate functionality

Patterns define the types of elements but not a specific number.

- A layered pattern doesn't tell you how many layers
- A pipe-and-filter pattern doesn't tell you how many pipes and filters
- A shared data pattern doesn't tell you how many data repositories and data accessors

The architect now has to *apply* the chosen pattern(s) to define a new set of elements that conform to it.

Functionality is allocated to the instantiated elements.

Step 5: Instantiate architectural elements and allocate functionality

The responsibilities of each module type must be documented:

- This usually requires the refinement of the parent module's responsibilities and the reallocation of its responsibilities to the child modules.

Note: This is the step that "creates" new elements.

These elements might need to be further refined – that is, decomposed and given sub-structure – during the next iteration of the method.

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality

These are bookkeeping and consolidation steps.

We must “hook together” designs of different parts of the system.

We must make sure that no requirements have “fallen through the cracks”.

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 8: Define interfaces for instantiated elements

The interface for each instantiated element is identified.

Interfaces consist of

- the services and properties that a element requires and produces
  - identified during the allocation of functionality
- the data and control flow information needed by each element as defined by the architectural pattern

At this point, interfaces need not be as detailed as a signature, but they document what elements need, what they can use, and on what they can depend.

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Each child element has responsibilities that are derived partially from the decomposition of requirements of the child's parent.

Those responsibilities must be translated into requirements that are derived and refined from the parent's requirements.

For example, a use case that initializes the whole system can be decomposed into use cases that initialize the subsystems.

Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information

Step 2: Choose part of the system to decompose

Step 3: Prioritize requirements and identify architectural drivers

Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we've chosen to decompose.

Step 5: Instantiate architectural elements and allocate functionality

Step 6: Merge designs completed thus far

Step 7: Allocate remaining functionality

Step 8: Define interfaces for instantiated elements

Step 9: Verify and refine requirements and make them constraints for instantiated elements

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

After each iteration, we have:

- A set of elements that decomposes an element we started the iteration with
- Each element will have
  - a collection of responsibilities
  - an interface
  - quality and functional requirements that pertain to it
  - constraints

Now we have the input for the next iteration of decomposition.

ADD: Summary

ADD is a general-purpose architecture design *method.*

As you can see, it

- Relies heavily on patterns and tactics
- Relies heavily on quality attribute requirements
- Results in a fully-justified architecture

We haven’t discussed architecture documentation yet, but the architect needs to document the selection and instantiation of patterns as he/she goes along.

More on that topic later.

A Picture of Architecture-Based Development -1

Development organizations who use architecture as a fundamental part of their way of doing business often define an architecture-based development process.

This seminar series will illuminate the usual parts of that process.

Typically, the first few steps are

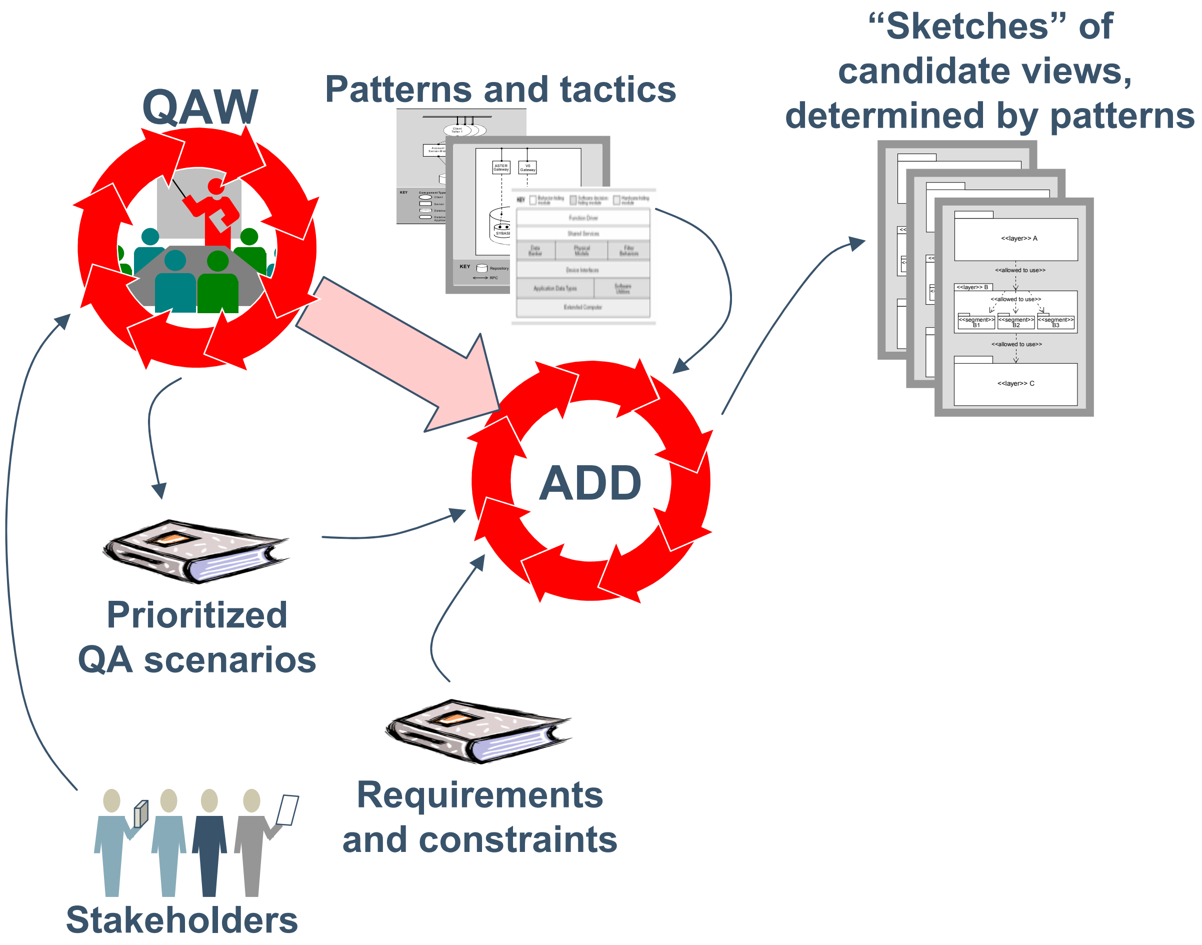
- Analyze the business case
- Understand the architecturally significant requirements
- Create an architecture to satisfy those requirements

A Picture of Architecture-Based Development -2

We now have tools in hand to carry out these steps.

- Architecture Business Cycle (ABC) – helps us identify business case factors that will shape the architecture
- Quality Attribute Workshop (QAW) – first way to engage the stakeholders.
- QA scenarios – the way to capture QA requirements.
- ADD – a method to design an architecture to meet its functional and QA requirements.

A Picture of Architecture-Based Dev.



Now what?

How do we know that our architecture is appropriate for its intended purpose?

In a large development project, an enormous amount of money may be riding on the architecture.

The company’s future may be at stake.

We need to *evaluate* the architecture.

How can we do this?

The SEI has developed the Architecture Tradeoff Analysis Method (ATAM).

The purpose of ATAM is: *to assess the consequences of architectural decisions in light of quality attribute requirements and business goals*.

Purpose of the ATAM – 1

The ATAM is a method that helps stakeholders ask the right questions to discover potentially problematic architectural decisions.

Discovered risks can then be made the focus of mitigation activities: e.g. further design, further analysis, prototyping.

Tradeoffs can be explicitly identified and documented.

Purpose of the ATAM – 2

The purpose of the ATAM is NOT to provide precise analyses . . . the purpose IS to discover risks created by architectural decisions.

We want to find *trends*: correlation between architectural decisions and predictions of system properties.

ATAM Benefits

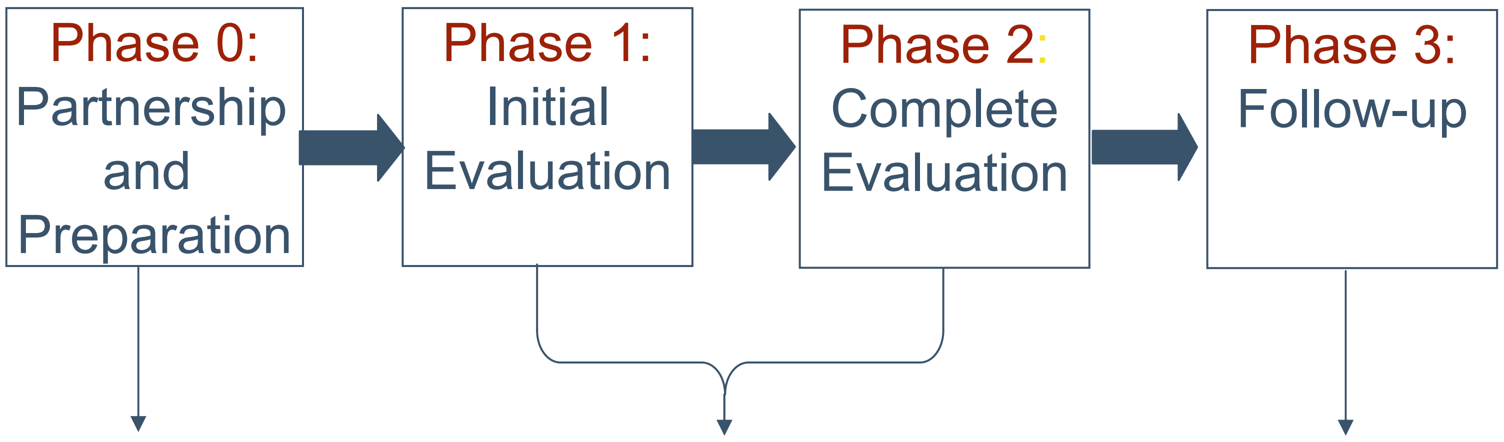
There are a number of benefits from performing ATAM evaluations

- identified risks
- clarified quality attribute requirements
- improved architecture documentation
- documented basis for architectural decisions
- increased communication among stakeholders

The results are improved architectures.

ATAM Phases

ATAM evaluations are conducted in four phases.


Phase 0: Partnership and Preparation
Phase 1: Initial Evaluation
Phase 2: Complete Evaluation
Phase 3: Follow-up

Duration: varies
Meeting: primarily phone, email

Duration: 1.5 - 2 days each for Phase 1 and Phase 2
Meeting: typically conducted at customer site

Duration: varies
Meeting: primarily phone, email

ATAM Phase 0

Phase 0: This phase precedes the technical evaluation.

- The customer and a subset of the evaluation team exchange understanding about the method and the system whose architecture is to be evaluated.
- An agreement to perform the evaluation is worked out.
- A core evaluation team is fielded.

ATAM Phase 1

Phase 1: involves a small group of predominantly technically-oriented stakeholders

Phase 1 is

- architecture centric
- focused on eliciting detailed architectural information and analyzing it
- top down analysis

ATAM Phase 1 Steps

1. Present the ATAM
2. Present business drivers
3. Present architecture
4. Identify architectural approaches
5. Generate quality attribute utility tree
6. Analyze architectural approaches
7. Brainstorm and prioritize scenarios
8. Analyze architectural approaches
9. Present results

Phase 1

1. Present the ATAM

The evaluation team presents an overview of the ATAM including:

- ATAM steps in brief
- Techniques
  - utility tree generation
  - architecture elicitation and analysis
  - scenario brainstorming/mapping
- Outputs
  - architectural approaches
  - utility tree and scenarios
  - risks, non-risks, sensitivity points, and tradeoffs

2. Present Business Drivers

ATAM customer representative describes the system’s business drivers including:

- business context for the system
- high-level functional requirements
- high-level quality attribute requirements
  - architectural drivers: quality attributes that “shape” the architecture
  - critical requirements: quality attributes most central to the system’s success

3. Present Architecture

Architect presents an overview of the architecture including:

- technical constraints such as an OS, hardware, or middleware prescribed for use
- other systems with which the system must interact
- architectural approaches used to address quality attribute requirements

Evaluation team begins probing for and capturing risks.

4. Identify Architectural Approaches

Identify predominant architectural approaches such as

- client-server
- 3-tier
- watchdog
- publish-subscribe
- redundant hardware

The evaluators begin to identify places in the architecture that are key to realizing quality attribute goals.

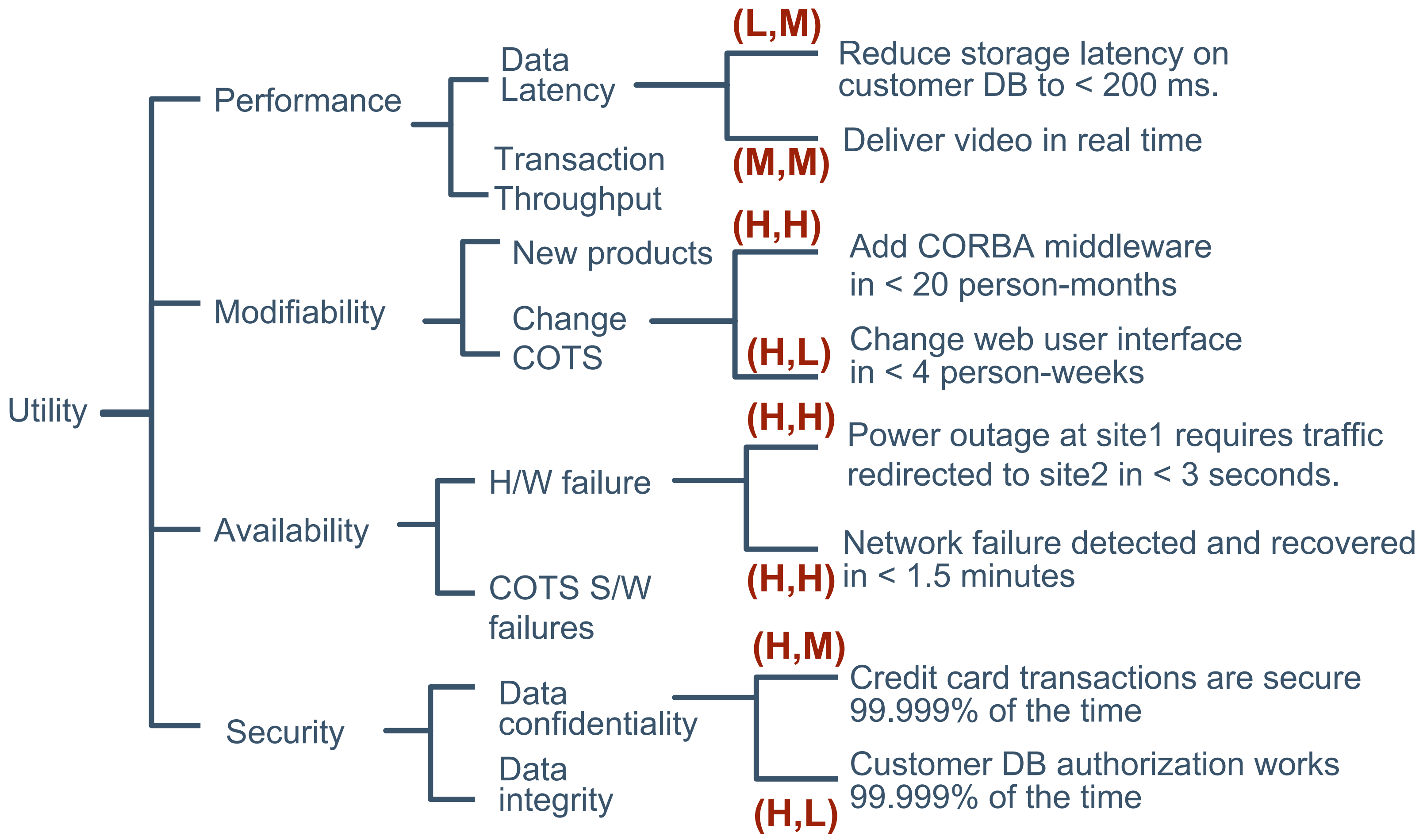
5. Generate Quality Attribute Utility Tree

Identify, prioritize, and refine the most important quality attribute goals by building a *utility tree*.

- A utility tree is a top-down vehicle for characterizing and prioritizing the “driving” attribute-specific requirements.
- The driving quality attributes are the high-level nodes (typically performance, modifiability, security, and availability).
- Scenarios are the leaves of the utility tree.

Output: a characterization and a prioritization of specific quality attribute requirements.

Utility Tree Construction


Utility
Performance
Data Latency
Transaction Throughput
(L,M)
Reduce storage latency on customer DB to < 200 ms.
(M,M)
Deliver video in real time
Modifiability
New products
Change COTS
(H,H)
Add CORBA middleware in < 20 person-months
(H,L)
Change web user interface in < 4 person-weeks
Availability
H/W failure
COTS S/W failures
(H,H)
Power outage at site1 requires traffic redirected to site2 in < 3 seconds.
(H,H)
Network failure detected and recovered in < 1.5 minutes
Security
Data confidentiality
Data integrity
(H,M)
Credit card transactions are secure 99.999% of the time
(H,L)
Customer DB authorization works 99.999% of the time

Scenarios

Scenarios are used to

- represent *stakeholders*' interests
- understand quality attribute requirements

Scenarios should cover a range of

- use case scenarios: anticipated uses of the system
- growth scenarios: anticipated changes to the system
- exploratory scenarios: unanticipated stresses to the system

A good scenario makes clear what the stimulus is that causes it and what responses are of interest.

Example Scenarios

Use case scenario

Remote user requests a database report via the Web during peak period and receives it within 5 seconds.

Growth scenario

Add a new data server to reduce latency in scenario 1 to 2.5 seconds within 1 person-week.

Exploratory scenario

Half of the servers go down during normal operation without affecting overall system availability.

Scenarios should be as specific as possible.

Stimuli, **Environment,** Responses

Use Case Scenario

Remote user requests a database report via the Web **during peak period and receives it** within 5 seconds.

Growth Scenario

Add a new data server **to reduce latency in scenario 1 to 2.5 seconds** within 1 person-week.

Exploratory Scenario

Half of the servers go down **during normal operation** without affecting overall system availability.

6. Analyze Architectural Approaches

Evaluation team probes architectural approaches from the point of view of specific quality attributes to identify risks.

- identify the architectural approaches
- ask quality attribute specific questions for highest priority scenarios
- identify and record risks and non-risks, sensitivity points and tradeoffs

Quality Attribute Questions

Quality attribute questions probe architectural decisions that bear on quality attribute requirements.

Performance

- How are priorities assigned to processes?
- What are the message arrival rates?

Modifiability

- Are there any places where layers/facades are circumvented?
- What components rely on detailed knowledge of message formats?

Risks, Tradeoffs, Sensitivities, and Non-Risks

A *risk* is a potentially problematic architectural decision.

Non-risks are good architectural decisions that are frequently implicit in the architecture.

A *sensitivity point* is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response.

A *tradeoff point* is a property that affects more than one attribute and is a sensitivity point for more than one attribute.

Risks and Tradeoffs

Example Risk:

- *“Rules for writing business logic modules in the second tier of your 3-tier architecture are not clearly articulated. This could result in replication of functionality thereby compromising modifiability of the third tier.”*

Example Tradeoff:

- *“Changing the level of encryption could have a significant impact on both security and performance.”*

Sensitivity Points and Non-Risks

Example Sensitivity Point:

- *“The average number of person-days of effort it takes to maintain a system might be sensitive to the degree of encapsulation of its communication protocols and file formats.”*

Example Non-Risk:

- *“Assuming message arrival rates of once per second, a processing time of less than 30 ms, and the existence of one higher priority process, a 1 second soft deadline seems reasonable.”*

ATAM Phase 2

Phase 2: involves a larger group of stakeholders

Phase 2 is

- stakeholder centric
- focused on eliciting diverse stakeholder points of view and on verification of the Phase 1 results

Carnegie Mellon
Software Engineering Institute

ATAM Phase 2 Steps

1. Present the ATAM
2. Present business drivers
3. Present architecture
4. Identify architectural approaches
5. Generate quality attribute utility tree
6. Analyze architectural approaches

7. Brainstorm and prioritize scenarios
8. Analyze architectural approaches
9. Present results

Recap Phase 1

Phase 2

Do this

7. Brainstorm and Prioritize Scenarios

Stakeholders generate scenarios using a facilitated brainstorming process.

- Scenarios at the leaves of the utility tree serve as examples to facilitate the step.

In phase 2, each stakeholder is allocated a number of votes roughly equal to 0.3 x #scenarios.

8. Analyze Architectural Approaches

Identify the architectural approaches impacted by the scenarios generated in the previous step.

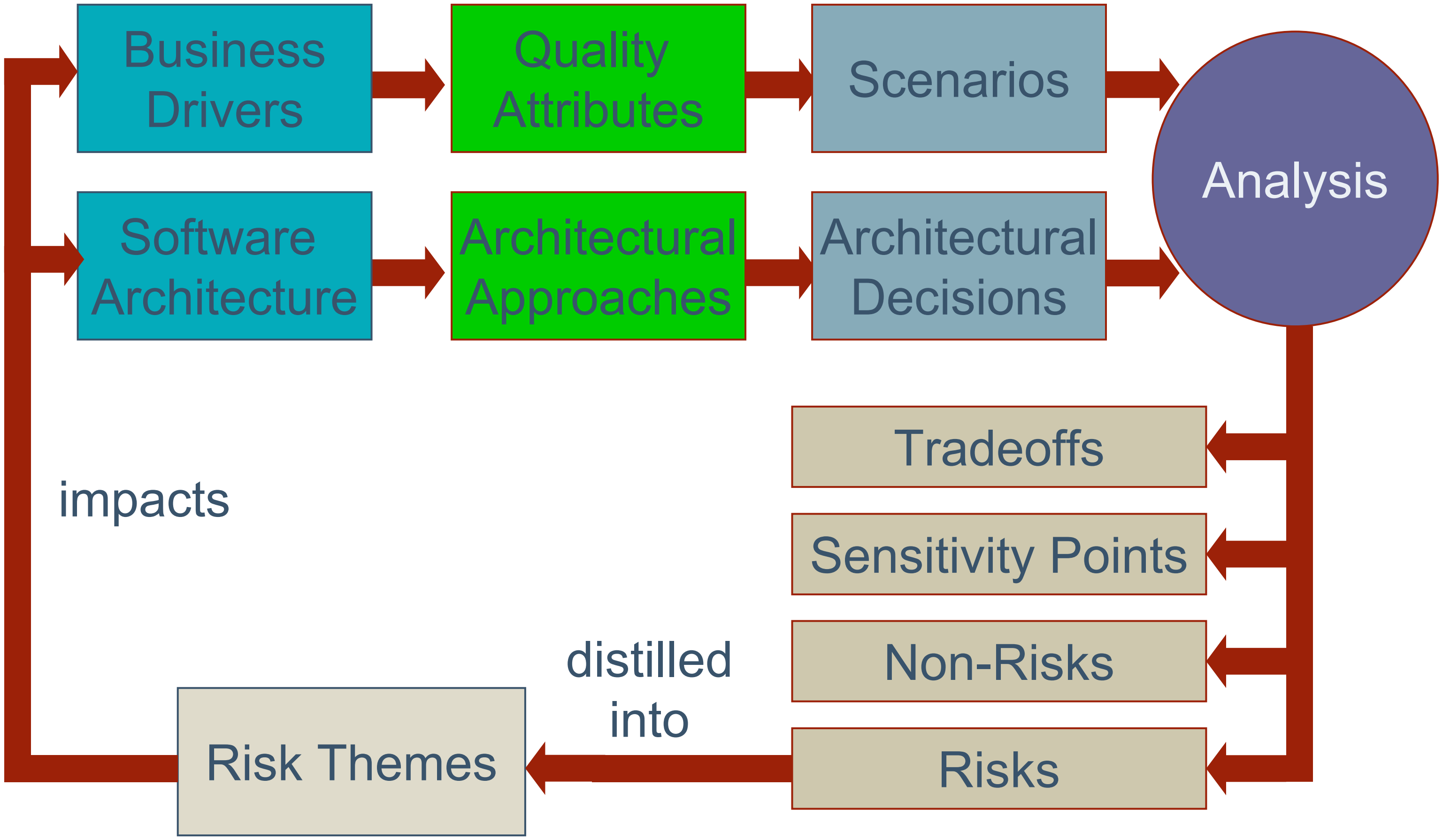
- This step continues the analysis started in step 6 using the new scenarios.
- Continue identifying risks and non-risks.
- Continue annotating architectural information.

9. Present Results

Recapitulate all the steps of the ATAM and present the ATAM outputs, including

- architectural approaches
- utility tree
- scenarios
- risks and non-risks
- sensitivity points and tradeoffs
- risk themes

Conceptual Flow of ATAM



ATAM Phase 3

Phase 3: primarily involves producing a final report for the customer as well as reflecting upon the quality of the evaluation and the ATAM materials.

The Final Report

The evaluation team will typically create the final report which includes:

- Executive summary
- Description of ATAM
- Description of business drivers and architecture
- List of phase 1 and phase 2 scenarios and utility tree
- Phase 1 and phase 2 analysis: architectural approaches, decisions, risks, sensitivities, tradeoffs, and non-risks
- Risk themes
- Next steps

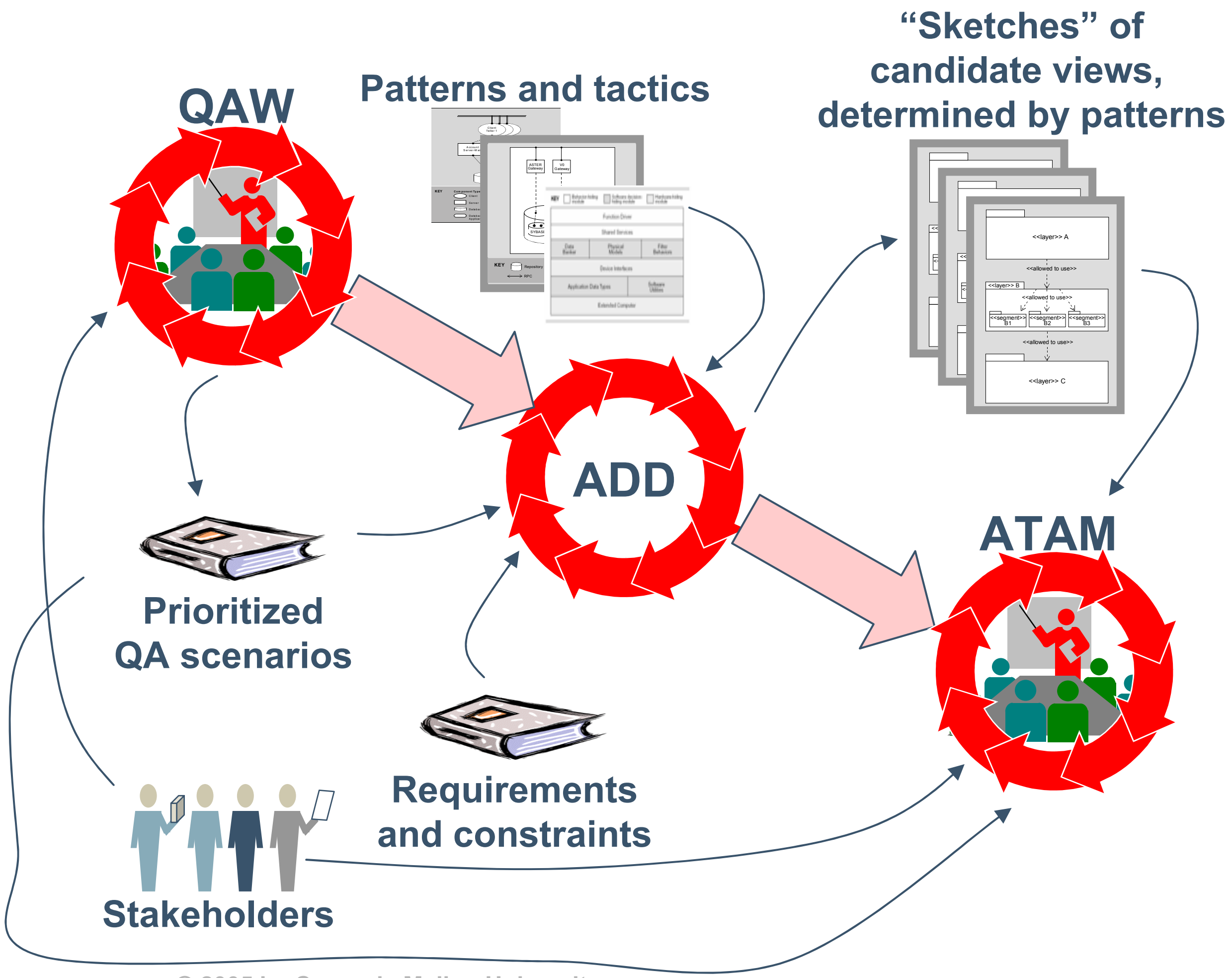
Carnegie Mellon
Software Engineering Institute

Summary

The ATAM is

- a method for evaluating an architecture with respect to multiple quality attributes
- an effective strategy for discovering the consequences of architectural decisions
- a method for identifying *trends*, not for performing precise analyses

A Picture of Architecture-Based Dev.


QAW
Patterns and tactics
"Sketches" of candidate views, determined by patterns
ADD
ATAM
Prioritized QA scenarios
Requirements and constraints
Stakeholders

Carnegie Mellon
Software Engineering Institute

Topics

Documenting software architectures:

How do we write down our architecture so that others can use it, understand it, and build a system from it?

Documenting an architecture

Architecture serves as the blueprint for the system, and the project that develops it.

- It defines the work assignments.
- It is the primary carrier of quality attributes.
- It is the best artifact for early analysis.
- It is the key to post-deployment maintenance and mining.

Documenting the architecture is the crowning step to creating it.

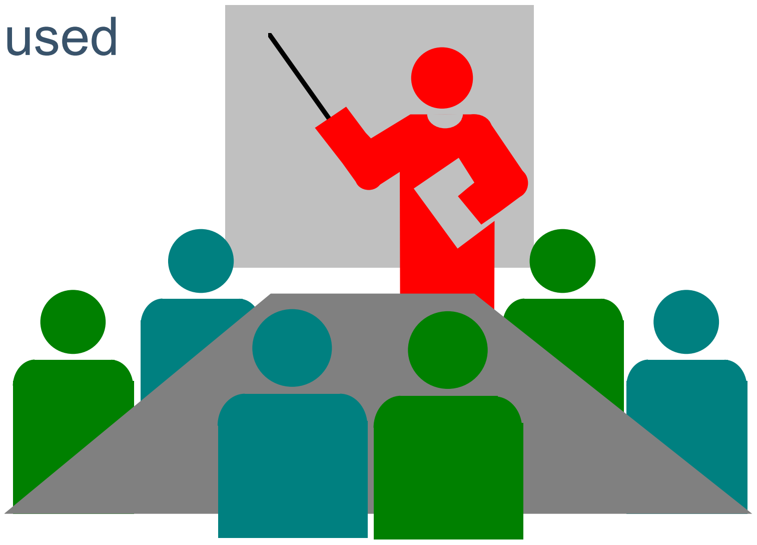
Documentation speaks for the architect, today and 20 years from today.

Is documentation *that* important?

Architecture documentation is important if and only if *communication* of the architecture is important.

How can an architecture be used if it cannot be understood?

How can it be understood if it cannot be communicated?



How do you document a software architecture?

We used to hear this question all the time!

- Via our website
- When we engage customers
- When we perform an architecture evaluation, which requires documentation.

Until now, our answer has always been “Not like that.”

What’s the answer?

“How do you document a software architecture?”

In industry, the answer seems to be

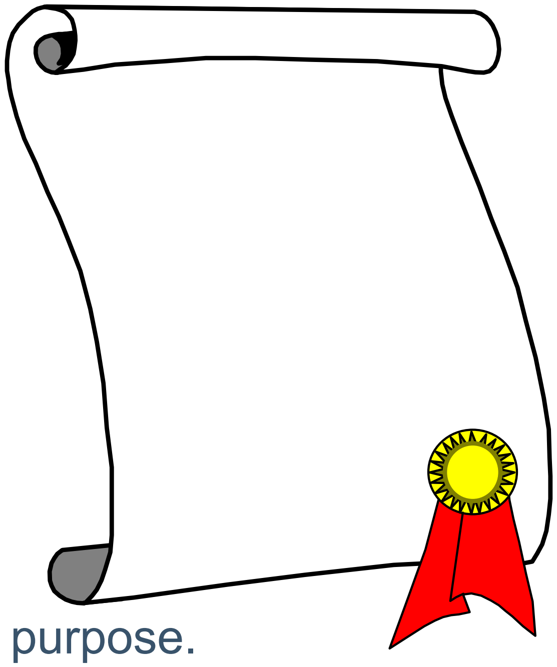
- “Use UML.”
- “Draw boxes and lines.”
- “What else do I need besides my class diagrams in Rose?”
- “Not very well.”
- “How do you document a *what*?”

Now, however, we have a much better answer.

Seven Principles of Sound Documentation

Certain principles apply to all documentation, not just documentation for software architectures.

1. Write from the point of view of the reader.
2. Avoid unnecessary repetition.
3. Avoid ambiguity.
4. Use a standard organization.
5. Record rationale.
6. Keep documentation current but not too current.
7. Review documentation for fitness of purpose.



1. Write from the point of view of the reader.

What will the reader want to know when reading a document?

- Make information easy to find!
- Your reader will appreciate your effort and be more likely to read your document.

Signs of documentation written for the writer's convenience:

- stream of consciousness: the order is that in which things occurred to the writer
- stream of execution: the order is that in which things occur in the computer

2. Avoid unnecessary repetition.

Each kind of information should be recorded in exactly one place.

This makes documents easier to use and easier to change.

Repetition often confuses, because the information is repeated in slightly different ways. Which is correct?

3. Avoid ambiguity.

Documentation is for communicating information and ideas. If the reader misunderstands, the documentation has failed.

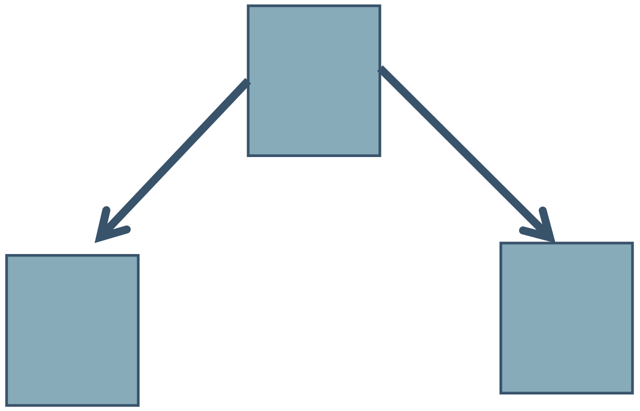
Precisely-defined notations/languages help avoid whole classes of ambiguity.

If your documentation uses a graphical language

- always include a key
- either point to the language's formal definition or give the meaning of each symbol. Don't forget the lines!

Carnegie Mellon
Software Engineering Institute

3. Avoid ambiguity (cont'd.)



Box-and-line diagrams are a very common form of architectural notation.

But what do they mean?

These do not show an architecture, but only the beginning of one.

If you use one, always define precisely what the boxes are and what the lines are.

If you see one, ask the owner what it means. The result is usually very entertaining.

4. Use a standard organization.

Establish it, make sure your documents follow it, and make sure that readers know what it is.

A standard organization

- helps the reader navigate and find information
- helps the writer place information and measure work left to be done
- embodies completeness rules, and helps check for validation

4. Use a standard organization (cont'd.)

Corollaries:

- Organize the documentation for ease of reference.
  - A document may be *read* once, if at all.
  - A successful document will be *referred to* hundreds or thousands of times.
  - Make information easy to find.

- Don't leave incomplete sections blank; mark them "to be determined"

5. Record rationale.

Why did you make certain design decisions the way you did?

Next week, next year, or next decade, how will you remember? How will the next designer know?

Recording rationale requires discipline, but saves enormous time in the long run.

Record rejected alternatives as well.

6. Keep documentation current but not too current.

Keep it current:

- Documentation that is incomplete, out of date, does not reflect truth, and does not obey its own rules for form is not used.
- Documentation that is kept current is used.
- With current documentation, questions are most efficiently answered by referring the questioner to the documentation.
- If a question cannot be answered with a document, fix the document and then refer the questioner to it.
- This sends a powerful message.

6. Keep documentation current but not too current (cont'd.)

Don't keep it *too* current

- During the design process, decisions are considered and re-considered with great frequency.
- Revising the documentation every five minutes will result in unnecessary expense.
- Choose points in the development plan when documentation is brought up to date
- Follow a release strategy that makes sense for your project.

Some key documentation questions

→ 1. Who will use the documentation and for what purposes?

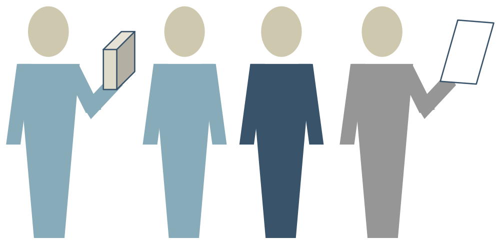
2. What kind of information shall we record about an architecture?

3. What languages and notations shall we use to record that information?

4. How shall we record and organize the information we've chosen, using the languages/notations we've chosen, to best meet the purposes we've identified?

1. Who will use the documentation and for what purposes?

Who are the stakeholders of architecture documentation?



What purposes do they need it for?

- Communication and understanding
- Education of people new to the system
- Analysis

Some key documentation questions

1. Who will use the documentation and for what purposes?

2. What kind of information shall we record about an architecture?

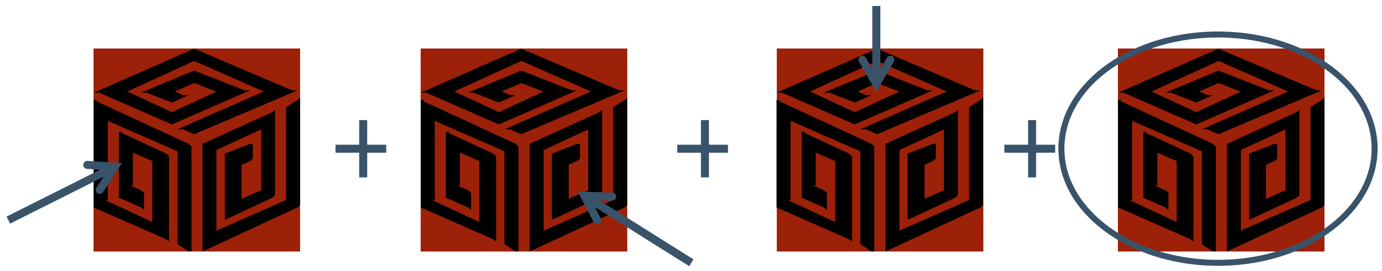
3. What languages and notations shall we use to record that information?

4. How shall we record and organize the information we've chosen, using the languages/notations we've chosen, to best meet the purposes we've identified?

2. What kind of information shall we record about an architecture?

The concept of a "view" gives us our main principle of architecture documentation:

Document the relevant views,
and then add information
that applies to more than one view,
thus tying the views together.



We call this the "Views and Beyond" approach to architecture documentation.

Carnegie Mellon
Software Engineering Institute

Review: Views

Architecture and structure, re-visited

Architecture is about structure. But which structure? Software has more than one.

Parnas made this observation in 1976 (“On a ‘Buzzword’: ‘Hierarchically Structured Systems’”).

- Systems have many kinds of “pieces”: programs, objects, classes, modules, processes, frameworks, tasks, threads…
- Each one defines a different structure.
- Which one is the architecture?

Answer: All of them might be.

Using structures and views

Each structure provides the architect with an *engineering handle* on some aspect of the system.

Architects choose the structures that need to engineer based on the important quality attribute drivers.

They record their designs using the corresponding views.

Structures and Views - 2

- *structure* – an actual set of architectural elements as they exist in software or hardware
- *view* – a representation of a coherent set of architectural elements, as written by and read by system stakeholders. A view represents a a set of elements and the relationships between those elements.

Views -1

Views are a way to manage complexity.

Each view can be used to answer a different question about the architecture

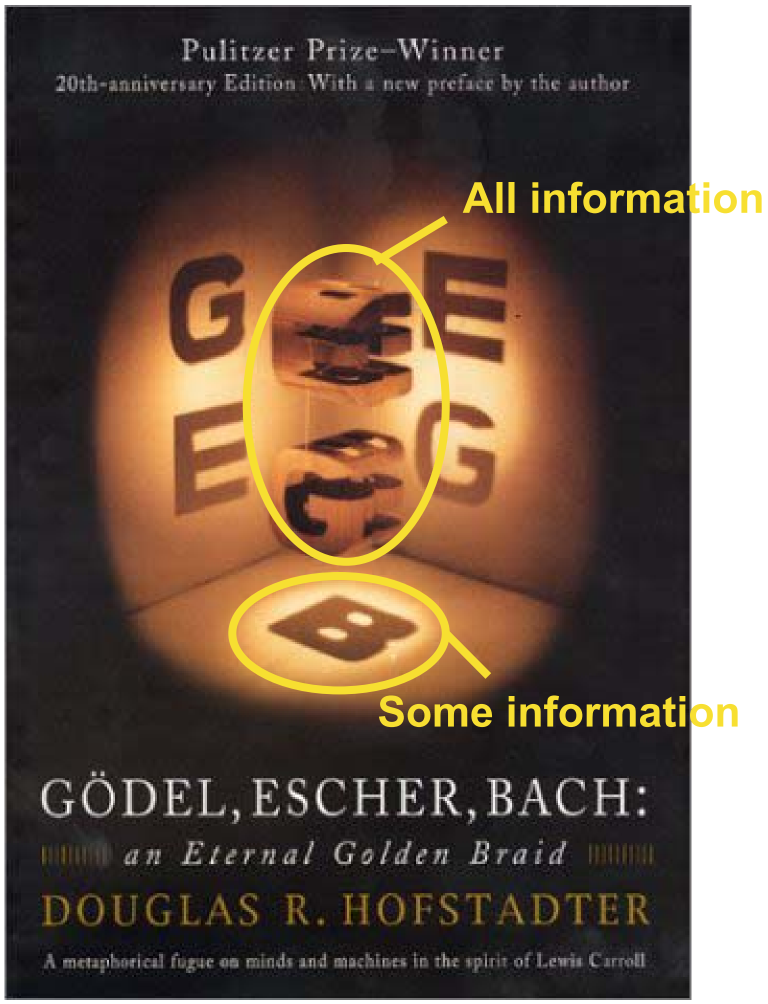
- What are the major execution units and data stores?
- What software is other software allowed to use?
- How does data flow through the system?
- How is the software deployed onto hardware?

Views -2

A view is a representation of a set of architectural elements and the relations associated with them.

Not *all* architectural elements -- *some* of them.

A view binds element *types* and relation *types* of interest, and shows those.


Pulitzer Prize-Winner
20th-anniversary Edition With a new preface by the author
All information
Some information
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll

Architectural Structures

Architectural structures (and hence views) can be divided into three types:

1. **"module" structures** – consisting of elements that are units of implementation called *modules*
2. **"component-and-connector" structures** – consisting of runtime components (units of computation) and the connectors (communication paths) between them
3. **"allocation" structures** – consisting of software elements and their relationships to elements in external environments in which the software is created and executed

Example Module Views

Decomposition view – shows modules that are related via the *"is a submodule of"* relation

Uses view – shows modules that are related via the *"uses"* relation (i.e., one module uses the services provided by another module)

Layered view – shows modules that are partitioned into groups of related and coherent functionality. Each group represents one layer in the overall structure.

Class/generalization view – shows modules called classes that are related via the *"inherits from"* or *"is an instance" of* relations

Example Component-and-Connector Views

Process view – shows processes or threads that are connected by communication, synchronization, and/or exclusion operations

Concurrency views – shows components and connectors where connectors represent “logical threads”

Shared-data (repository) views – shows components and connectors that create, store, and access persistent data

Client-server view – shows cooperating clients and servers and the connectors between them (i.e., the protocols and messages they share)

Example Allocation Views

Deployment view – shows software elements and their allocation to hardware and communication elements

Implementation view – shows software elements and their mapping to file structures in the development, integration, and configuration control environments

Work assignment view – shows modules and how they are assigned to the development teams responsible for implementing and integrating them

Carnegie Mellon
Software Engineering Institute

End of Review on Views

Carnegie Mellon
Software Engineering Institute

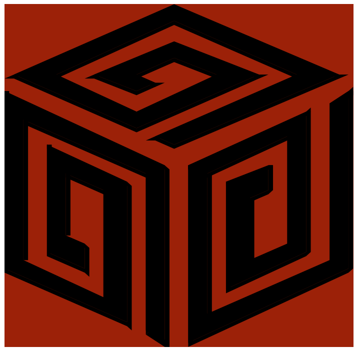
Other Views -1

Kruchten's 4+1 views
(1995, later adopted for RUP):

- Logical view: supports behavioral requirements. Key abstractions, which are objects or object classes
- Process view: addresses concurrency and distribution. Maps threads to objects.
- Development view: organization of software modules, libraries, subsystems, units of development.
- Physical view: maps other elements onto processing and communication nodes.
- "Plus one" view: Maps the other views onto important use cases to show how they work.

Carnegie Mellon
Software Engineering Institute

Other Views -2



Siemens Four-Views (Hofmeister, Nord, Soni, *Applied Software Architecture*, 2000):

- Conceptual view
- Module interconnection view
- Execution view
- Code view

Herzum & Sims (*Business Component Factory*, 1999):

- Technical architecture
- Application architecture
- Project management architecture
- Functional architecture

Other Views -3

Software Cost Reduction method
(Parnas, et al., 1980s)

- Module decomposition view: shows modules as units of encapsulation; used to isolate changes and achieve modifiability
- Process view: shows processes and how they synchronize and communicate at run-time; used to achieve performance
- Uses view: shows programs and how they depend on each other; used to achieve incremental development and the ability to quickly field subsets

Which Views to Use?

As you can see, many authors and methods prescribe a standard set of views.

However, a more modern approach is to say that an architect should choose the views that best serve the purposes for the system and its stakeholders.

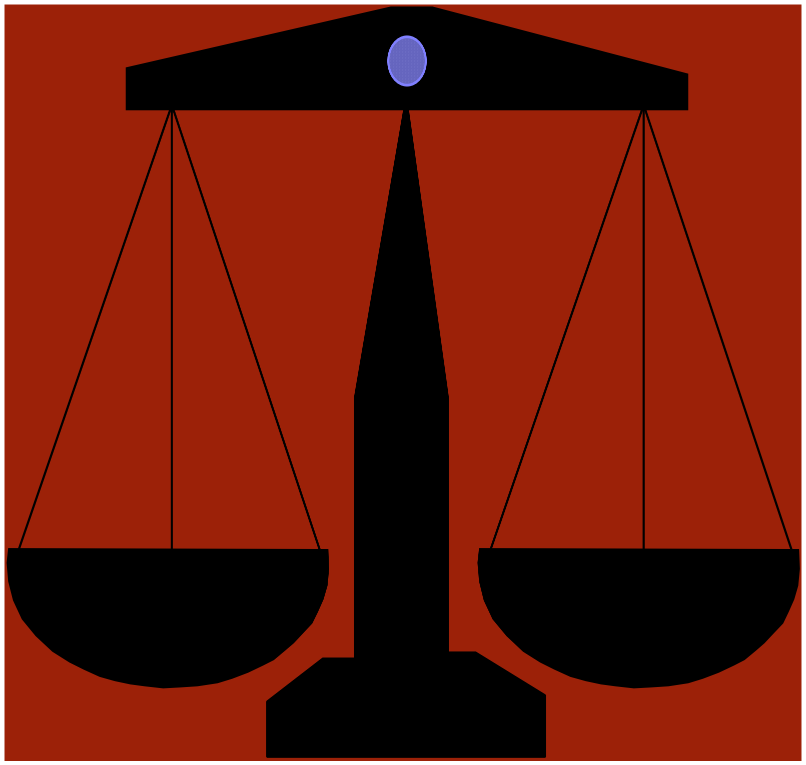
Hence, we need a method for choosing views.

How many views do we need in our documentation package?

Each view comes with a cost.

Each view comes with a benefit.

Planning a view set requires understanding the needs of the stakeholders, and the resources available.



How to proceed?

1. Build a table.
   - ROWS: Enumerate the stakeholders
   - COLUMNS: Enumerate the set of styles that *could* apply to the architecture being documented. This is our potential set of views.
   - Check box (x,y) if stakeholder x would like view y.

2. Combine views appropriately to reduce number.

3. Prioritize views based on need. (Some stakeholders may have extra weight.)

Some key documentation questions

1. Who will use the documentation and for what purposes?
2. What kind of information shall we record about an architecture?
3. What languages and notations shall we use to record that information?
4. How shall we record and organize the information we've chosen, using the languages/notations we've chosen, to best meet the purposes we've identified?

3. What languages and notations shall we use to record that information?

UML

- Not designed to document architecture information (Evidence: No concept of “layer” – although you can stereotype a package to represent a layer)
- Nevertheless, the de facto standard language
- UML 2.0 is better, with architecture constructs like “component” and “connector”.

Informal “box and line” notations

- Always use a key!
- Advantage: Flexibility. Disadvantage: Vagueness, no tool support.

Architecture description languages (ADLs)

- Subject of much research in 1990s; not used much in practice: Rapide, Wright, UniCon, ACME, …
- AADL recently became an IEEE standard

If you use UML

Do not be seduced by the power of UML diagrams.

For example, a UML class diagram is a notation to show module views. But UML class diagrams are so powerful (some would say so conceptually confused) that you can show all kinds of run-time information in them as well.

Adopt a discipline for using UML diagrams. This serves the same role as a coding standard for programmers.

Carnegie Mellon
Software Engineering Institute

Some key documentation questions

1. Who will use the documentation and for what purposes?

2. What kind of information shall we record about an architecture?

3. What languages and notations shall we use to record that information?

→ 4. How shall we record and organize the information we’ve chosen, using the languages/notations we’ve chosen, to best meet the purposes we’ve identified?

Documenting a view -1

1. A primary presentation

- Usually graphical (we call this a *cartoon*)
- May be textual -- e.g., a table
- If graphical, includes a key explaining the notation (or pointing to explanation)
- Shows elements and relationships among them
- Shows information you wish to convey about the view (view packet) first

Many times, the primary presentation is all you get. It's not enough!

Documenting a view -2

2. An element catalog
   - Explains the elements depicted in the primary presentation
   - Lists elements and their properties (as defined by the relevant style guide)
   - Explains relations, and any exceptions or additions to the relations shown in the primary presentatio
   - Interfaces of elements

3. A context diagram
   - Shows how system (or portion shown in this view packet) relates to its environment.

Documenting a view -3

4. A variability guide
   - Shows the architectural mechanisms available to change the element

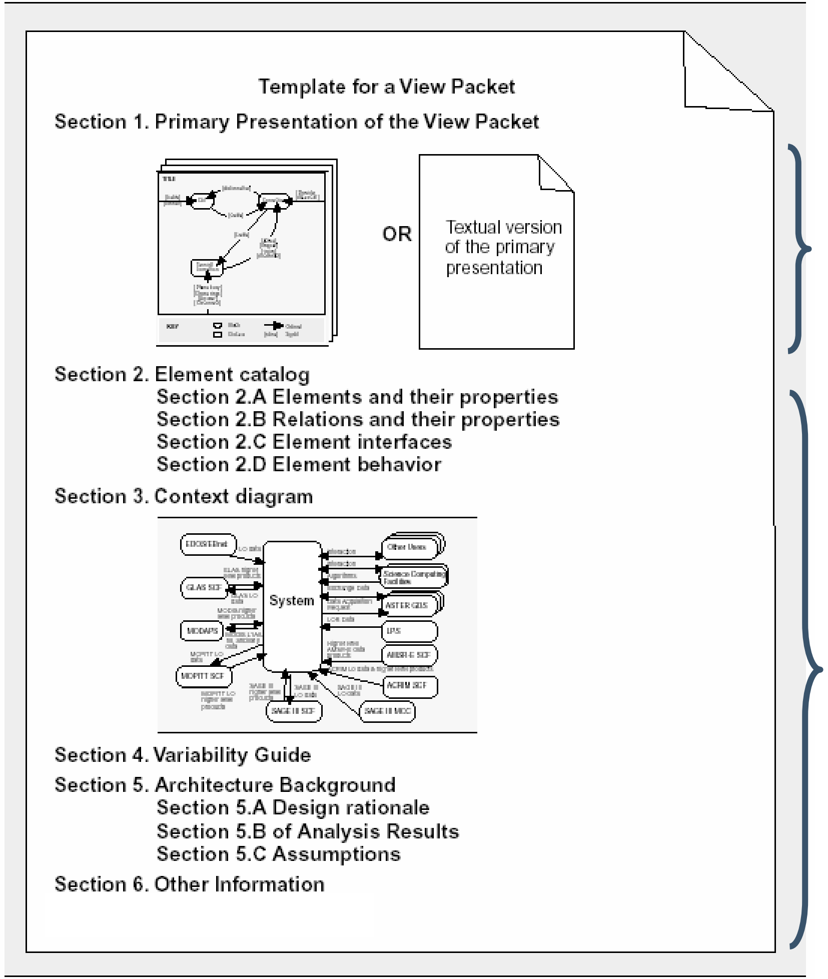
5. Architecture background
   - Rationale for design decisions that apply to the entire view (or to that portion of the view being shown), including rejected alternatives and factors that constrained the design
   - Analysis results validating the design decisions
   - Assumptions about the environment and about the need that the system is fulfilling

Documenting a view -4

6. Other information
   - System- and project-specific.
   - CM information, ownership information
   - Mapping to requirements
   - Not architectural, strictly speaking. But useful to capture alongside the architecture anyway.

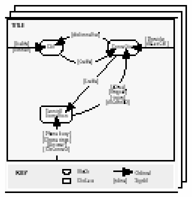
7. Related view packets
   - Pointers to sibling, child, and parent view packets

Summary: Documenting a View



Template for a View Packet

Section 1. Primary Presentation of the View Packet

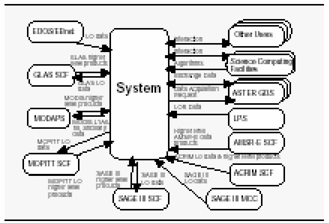


OR

Textual version of the primary presentation

Section 2. Element catalog
Section 2.A Elements and their properties
Section 2.B Relations and their properties
Section 2.C Element interfaces
Section 2.D Element behavior

Section 3. Context diagram



Section 4. Variability Guide

Section 5. Architecture Background
Section 5.A Design rationale
Section 5.B of Analysis Results
Section 5.C Assumptions

Section 6. Other Information

Section 1: Primary presentation

Sections 2-6: Supporting documentation

Documentation beyond views - 1

1. Documentation roadmap
   - How the documentation is organized to serve a stakeholder
   - List of views, with the elements/relations of each, and a statement of what the view is for
   - Scenarios for using the documentation, showing which parts should be consulted

2. View template
   - Explanation of how each view is documented
   - The *standard organization* for each view

Documentation beyond views -2

3. System overview
   - An informal, prose description of the system and its purpose and functionality
   - Goal is to provide context for new member
   - Perfectly OK to point to overview elsewhere if one exists in overall system documentation

4. Mapping between views
   - Establishes useful/insightful correspondence between various views
   - Tabular

Documentation beyond views -3

5. Directory
   - An index showing where every element, relation, and property is defined and used.

6. Architecture glossary and acronym list
   - May be subset of overall system glossary and acronym list. OK to point to larger document if so.

7. Background, design constraints, and rationale
   - As in views, but applied to cross-view design decisions.

Document template available

A Microsoft Word template for a software architecture document based on the Views and Beyond approach is available at

http://www.sei.cmu.edu/architecture/arch_doc.html

or

http://www.sei.cmu.edu/architecture
Click on documentation
Click on download

Topic

Software product lines:
One way to leverage the investment in architecture across an entire family of systems

Two Stories

CARNEGIE MELLON UNIVERSITY
SOFTWARE ENGINEERING INSTITUTE

Cummins, Inc.

World's largest
manufacturer of
large diesel engines.

Complex domain of variation

Today's diesel engines are driven by software

- Micro-control of ignition timing to achieve optimum mix of power, economy, emissions
- Conditions change dynamically as function of road incline, temperature, load, etc.
- Must also respond to statutory regulations that often change
- Reliability is critical! Multi-million dollar fleets can be put out of commission by a single bug
- 130KSLOC -- C, assembler, microcode
- Different sensors, platforms, requirements

In 1993, Cummins had a problem

Six engine projects were underway
Another 12 were planned.

Each project had complete control over its development process, architecture, even choice of language. Two were trying to use O-O methods.

Ron Temple (VP in charge) realized that he would need another 40 engineers to handle the new projects -- out of the question.

This was no way to do business.

What Cummins did

In May, 1994 Temple halted all the projects.

He split the leading project.

- One half built core assets -- generic software, documentation, and other assets that every product could use
- Other half became pilot project for using the core assets to turn out a product

In early 1995, the product was launched on time (relative to re-vamped schedule) with high quality.

Others followed.

Cummins' results

Achieved a product family capability with a breathtaking capacity for variation, or customization

- 9 basic engine types
- 4-18 cylinders
- 3.9 - 164 liter displacement
- 12 kinds of electronic control modules
- 5 kinds of microprocessors
- 10 kinds of fuel systems
- diesel fuel or natural gas

Highly parameterized code. 300 parameters are available for setting by the customer after delivery.

Cummins' results by the numbers -1

- 20 product groups launched, which account for over 1000 separate engine applications
- 75% of all software, on average, comes from core assets
- Product cycle time has plummeted. Time to first engine start went from 250 person-months to a few person-months. One prototype was bulit over a weekend.
- Software quality is at an all-time high, which Cummins attributes to product line approach.

Cummins' results by the numbers -2

- Customer satisfaction is high. Productivity gains enables new features to be developed (more than 200 to date)
- Projects are more successful. Before product line approach, 3 of 10 were on track, 4 were failing, and 3 were on the edge. Now, 15 of 15 are on track.
- Widespread feeling that developers are more portable, and hence more valuable.

Cummins' results by the numbers -3

Supported Components	1992	1993	1994	1995	1996	1997	1998
Electronic control modules (ECMs)	3	3	4	5	5	11	12
Fuel systems	2	2	3	5	5	10	11
Engines	3	3	5	5	12	16	17
Features * ECM	60	80	180	370	1100	2200	2400

Achieving this flexibility without the product line approach would have required 3.6 times the staff Cummins has.

Cummins' results by the numbers -4

Today's largest teams are smaller than yesterday's smallest teams. Two-person teams are not unusual.

Cummins management has a history of embracing change, but carefully targeted change.

- They estimate that process improvement alone has brought a benefit/cost ratio of 2:1 to 3:1.
- They estimate that the product line approach has brought a benefit/cost ratio of 10:1.

Product line approach let them quickly enter and then dominate the *industrial* diesel engine market.

Salion, Inc.

A small organization: 21 people

Maker of software for suppliers who sell complex products via proposals.

- Salion Revenue Process Manager—helps suppliers manage opportunities. It contains a workflow engine and Web-based communication tools to help a supplier organization manage the collaboration of design and pricing. It keeps track of a proposal's status and assists in the assembly of the final document.
- Salion Knowledge Manager—helps triage and analyze requests for proposals (RFPs), with decision support capabilities and analysis of bid performance, win/loss rates, and pricing.   Helps choose the best candidates from among all the available opportunities. The Knowledge Manager uses historical information to prioritize opportunities and improve response rates.
- Salion Business Link—extends collaboration between the supplier and the customer, and between the supplier and subsuppliers.

A specialized but important market

“It should take us one day or less to turn a quote around. For some reason, it takes five weeks. This process is out of control.” —Director of Engineering, Tier 1 automotive supplier

“We recently rushed a late quote out the door that we thought we had priced with a ‘nice margin.’ In reality, the quote was for a part that we had been selling at twice the price we quoted. Luckily, our customer only asked for one year of retroactive rebates.” —Director of Sales, Tier 1 automotive supplier

A specialized but important market

"We just spent $100,000 on an opportunity that we had no chance of winning. We bid on the same business two years ago and our price was 50% too high. We have no way to capture or analyze our historical sales and bidding performance, so we make the same mistakes over and over."—Tier 2 automotive supplier

"We spent $600,000 in overnight shipping costs last year."—Tier 1 automotive supplier

Variabilities

Customers run different combinations of products

Installation options:

- Run on customer’s hardware (installed)
- Run on Salion’s dedicated hardware (hosted)
- Run on Salion’s shared hardware

Each customer will have a unique workflow, a unique set of input screens and other user-interface concerns, and a unique set of reports they want to generate.

Each customer will have unique “bulk load” requirements, involving the transformation of existing data and databases into forms compatible with Salion’s products.

An automotive industry trade group has defined a business-to-business transaction framework encompassing some 120 standard objects to be used to transfer information from organization to organization. Not every customer will want to make use of all 120 objects.

How Salion builds its product line

First produced a “standard” product as its entry into the market.

That product formed the basis for Salion’s software product line and the basis for each new customer-specific product it fielded.

The standard product was more than an engineering model from which “real” systems were produced; it was also sold.

Typical product: 40 modules, 150K SLOC

Customization vs. configuration

Salion builds subsequent products by

- *customizing* elements of the “standard” product.
- *configuring* elements of the “standard” product.

Early on, Salion tried to make many elements configurable

- Forms manager
- Customized reports manager

Results were wasted effort, wrong guesses, and bloated software.   Now, Salion customizes these aspects.

Tool support plays an important role in managing these variations:

- 3,333 files for 3 products
- 88 files represent variations

Salion’s product line benefits

Seven developers produce and support sophisticated, highly secure, high-availability, COTS-intensive systems

As of report time, Salion had produced its 12th 30-day release, all of which were on schedule.

Building the standard product took 190 person-months.

- Building the first customer product took just 15 person-months with 97% reuse.
- Building the second product took 30% less effort.

Salion’s approach gives it superb position to answer investors question “How are you going to scale?”

- Normal answser: Re-write product to make robust, increase development staff, bring on QA staff
- Salion’s answer: Nothing. We can scale right now, as we are.

CelsiusTech: Ship System 2000

A family of 55 ship systems

Integration test of 1-1.5 million SLOC requires 1-2 people

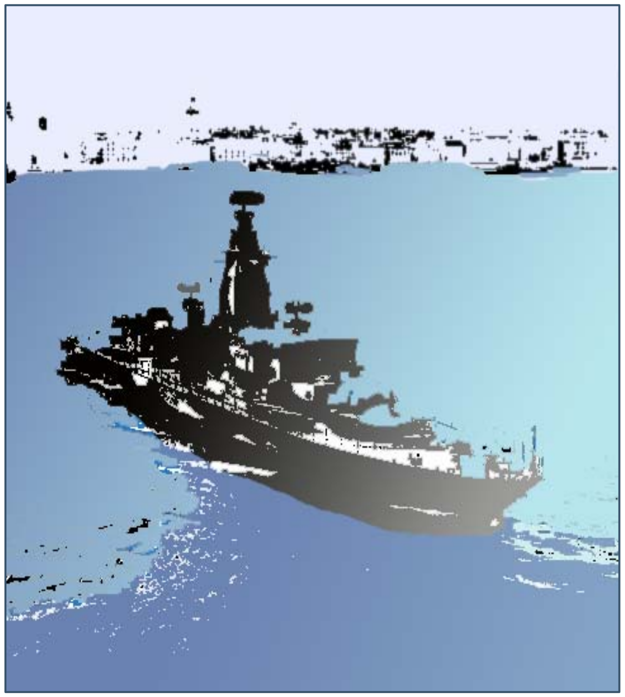
Rehosting to a new platform/OS takes 3 months

Cost and schedule targets are predictably met

Performance/distribution behavior are known in advance

Customer satisfaction is high

Hardware-to-software cost ratio changed from 35:65 to 80:20



National Reconnaissance Office / Raytheon: Control Channel Toolkit

Ground-based spacecraft command and control systems

Increased quality by 10X
Incremental build time reduced from months to weeks
Software productivity increased by 7X
Development time and costs decreased by 50%
Decreased product risk

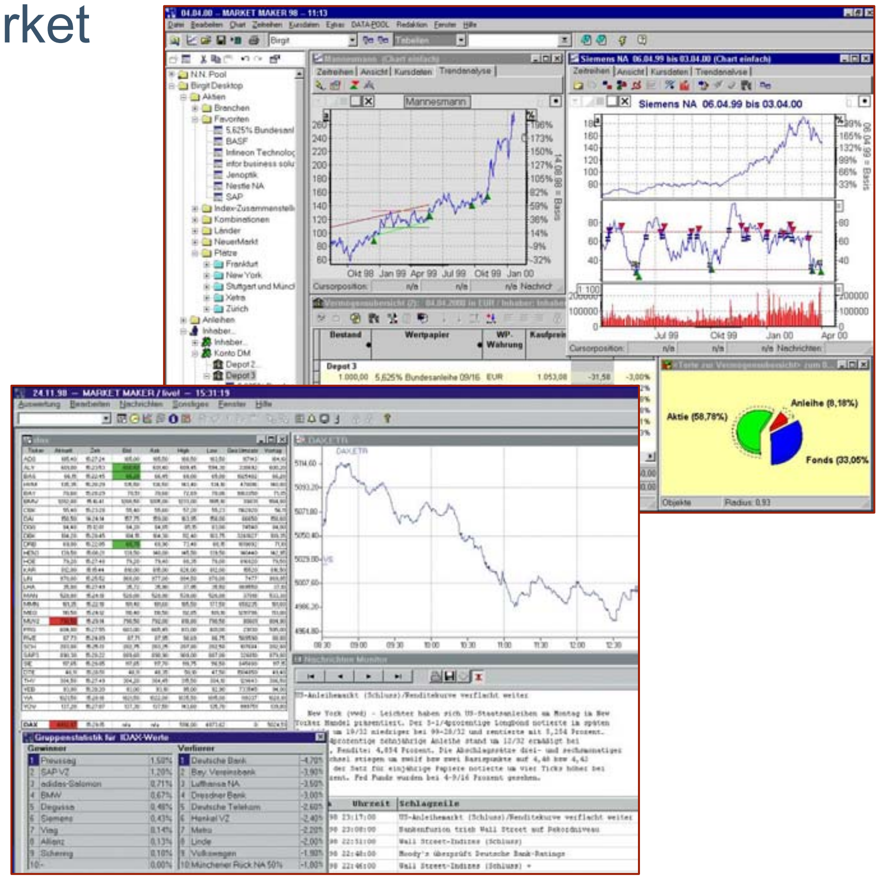
Carnegie Mellon
Software Engineering Institute

Market Maker GmbH: MERGER

Internet-based stock market software

Each product "uniquely" configured

Three days to put up a customized system


04.04.00 – MARKET MAKER 98 – 11:13
Mannesmann
Siemens NA 06.04.99 bis 03.04.00
24.11.98 – MARKET MAKER / live! – 15:31:19
DAX-ETR
Gruppenstatistik für DAX-Werte
Gewinner
Verlierer
Aktie (58,78%)
Anleihe (8,18%)
Fonds (33,05%)
Uhrzeit
Schlagzeile

Hewlett Packard

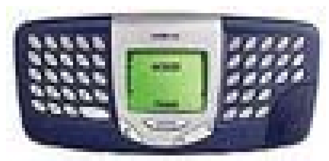
Printer systems

- 2-7x cycle time improvement (some 10x)
- Sample Project
  - –shipped 5x number of products
  - –that were 4x as complex
  - –and had 3x the number of features
  - –with 4x products shipped/person

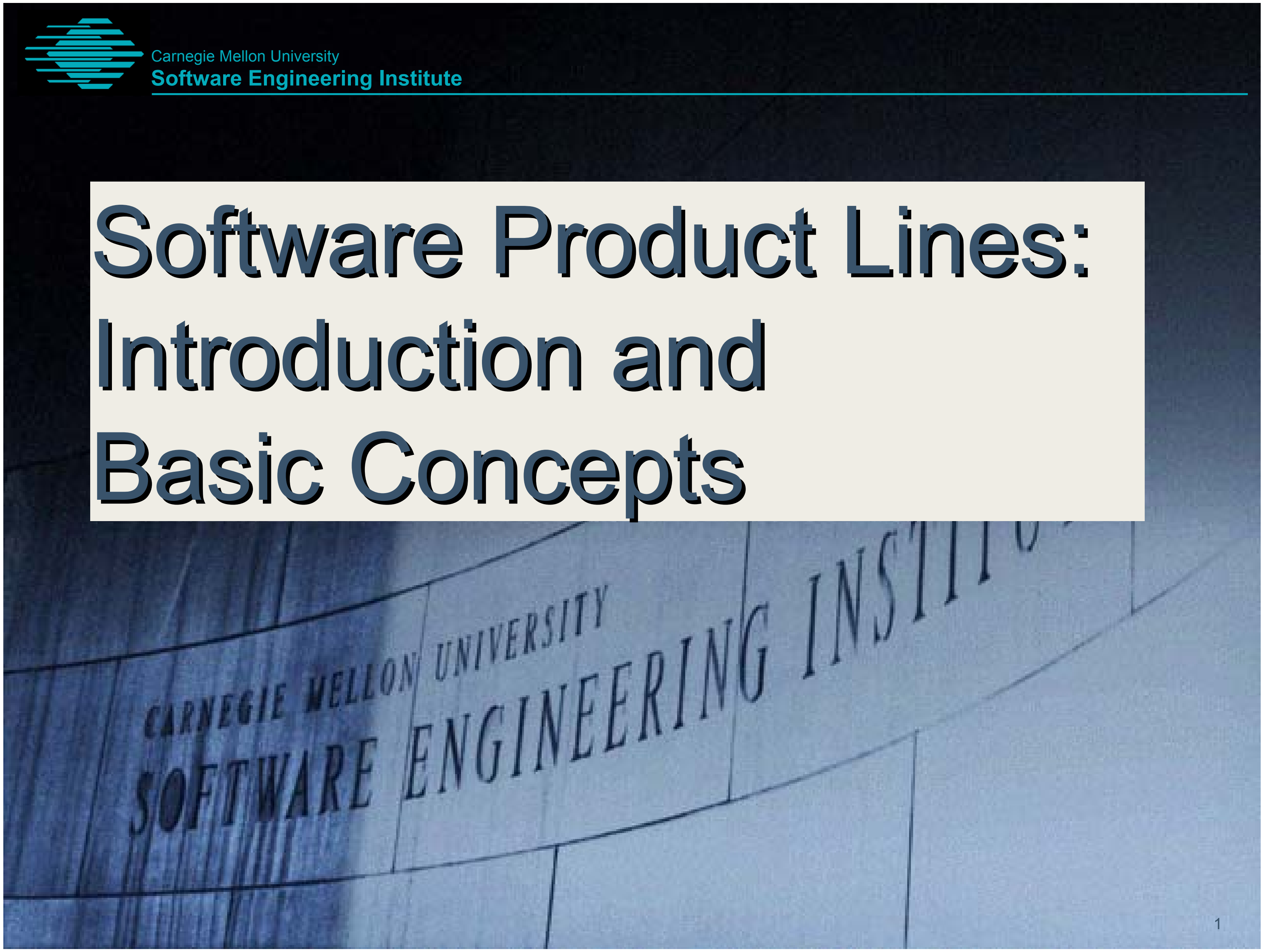
Nokia Mobile Phones

Product lines with 25-30 new products per year

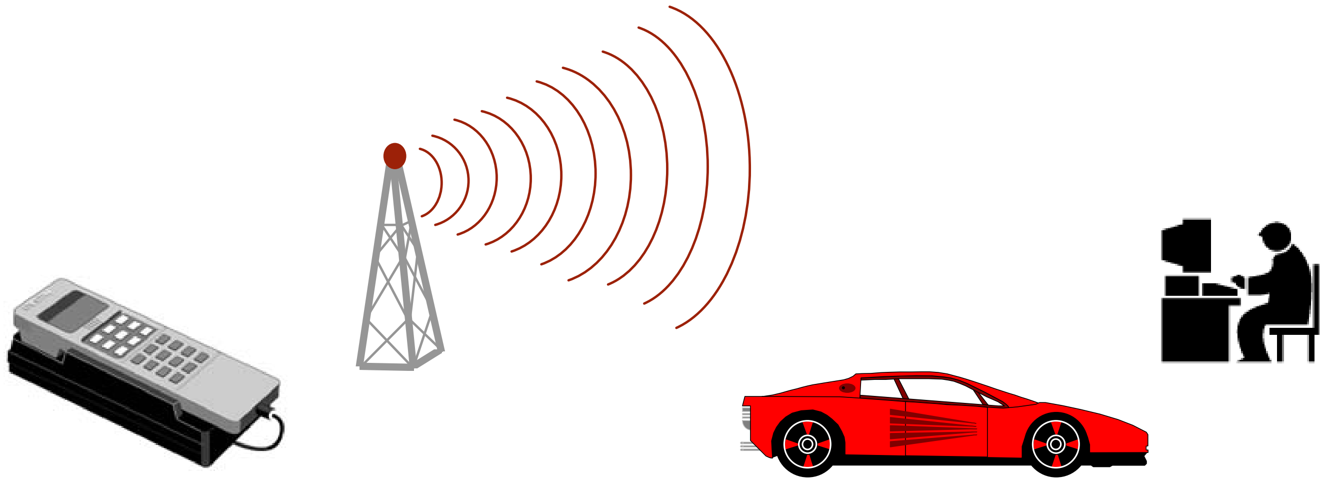
Across products there are
• varying number of keys
• varying display sizes
• varying sets of features
• 58 languages supported
• 130 countries served
• multiple protocols
• needs for backwards compatibility
• configurable features
• needs for product behavior change after release



Software Product Lines: Introduction and Basic Concepts



Business Success Requires Software Prowess



Software pervades every sector.
Software has become the bottom line for many organizations who never envisioned themselves in the software business.

Universal Business Goals

High quality

Quick time to market

Effective use of limited resources

Product alignment

Low cost production

Low cost maintenance

Mass customization

Mind share

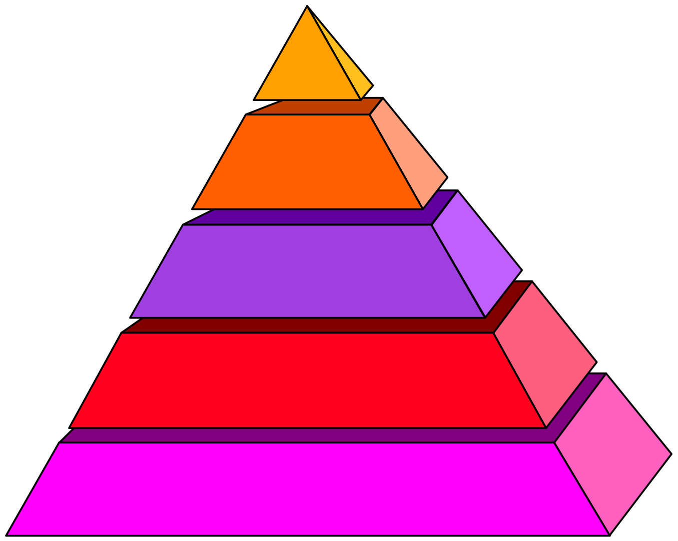
$\Bigg\}$ **improved efficiency and productivity**

Software (System) Strategies

Process Improvement

Technology Innovation

Reuse

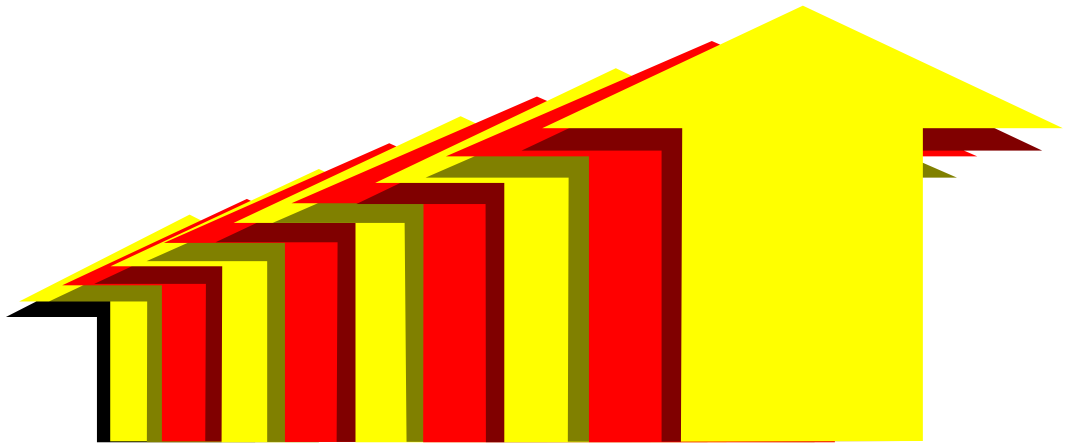


Few Systems Are Unique

Most organizations produce families of similar systems, differentiated by features.

Carnegie Mellon
Software Engineering Institute

Reuse History



1960's
Subroutines

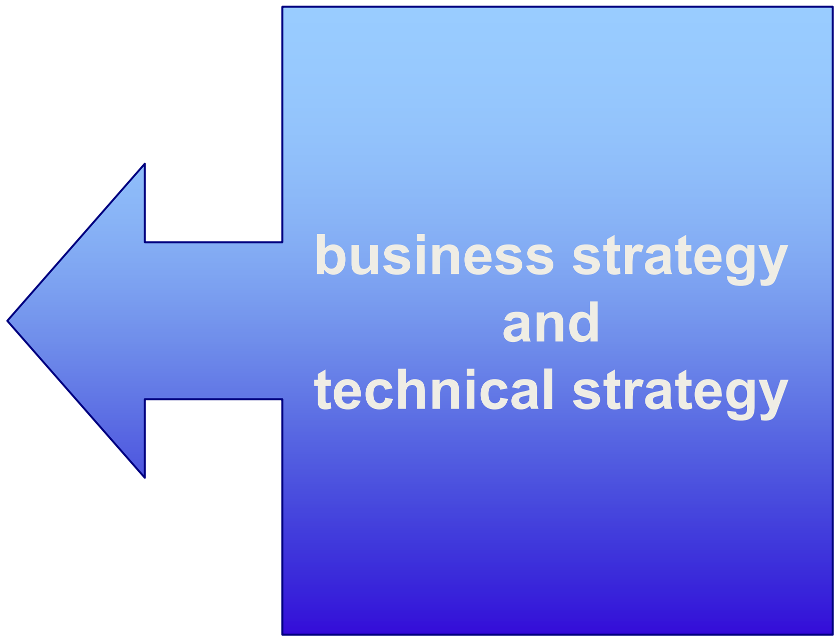
1970's
Modules

1980's
Objects

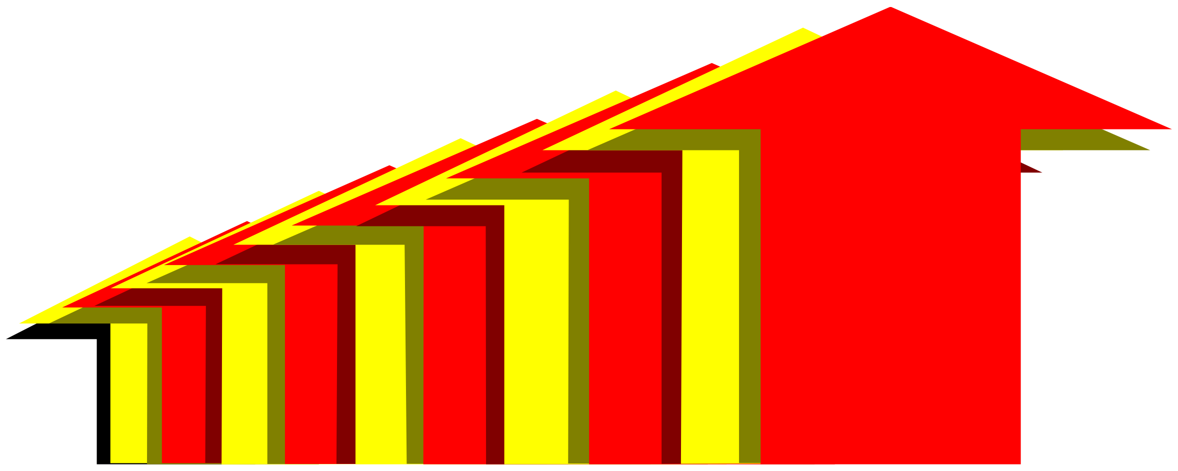
1990's
Components

Focus was small-grained and opportunistic.
Results fell short of expectations.

Imagine Strategic Reuse



Reuse History: From Ad-Hoc to Systematic



1960's
Subroutines

1970's
Modules

1980's
Objects

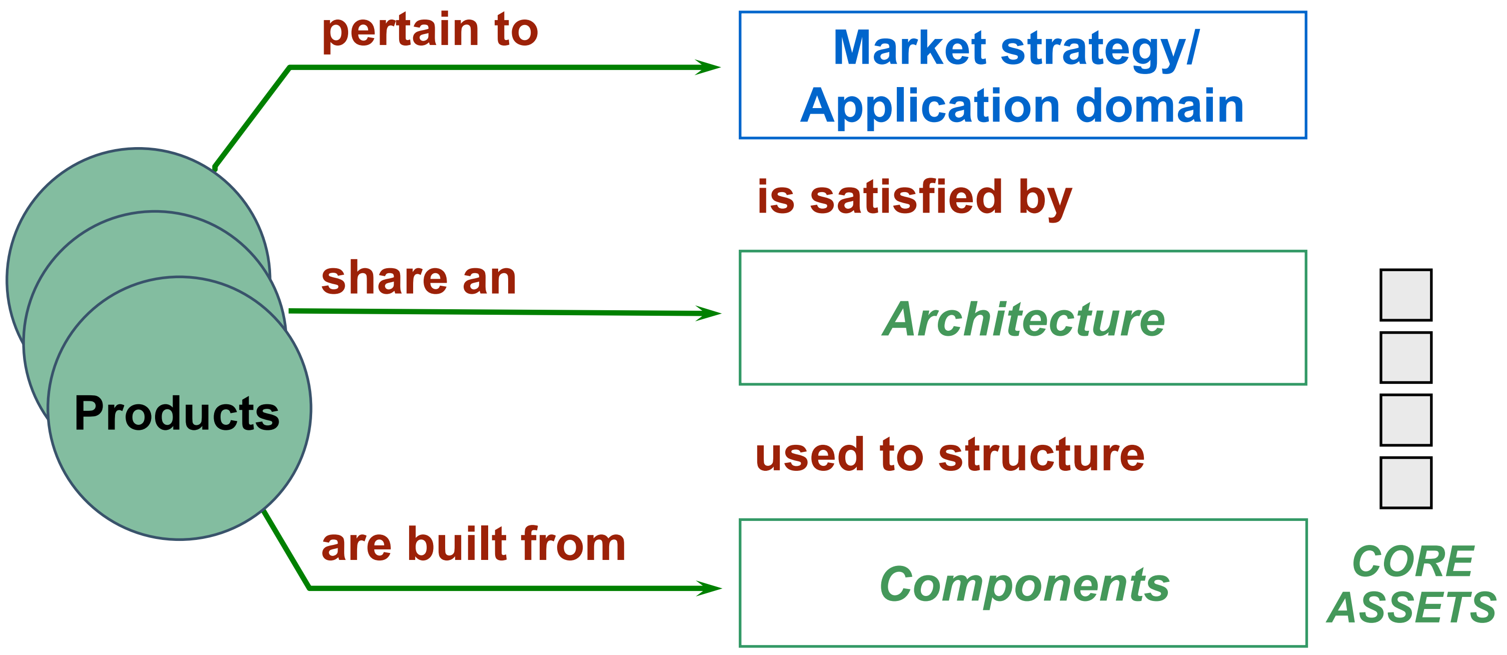
1990's
Components

2000's
Software
Product Lines

What is a Software Product Line?

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Software Product Lines

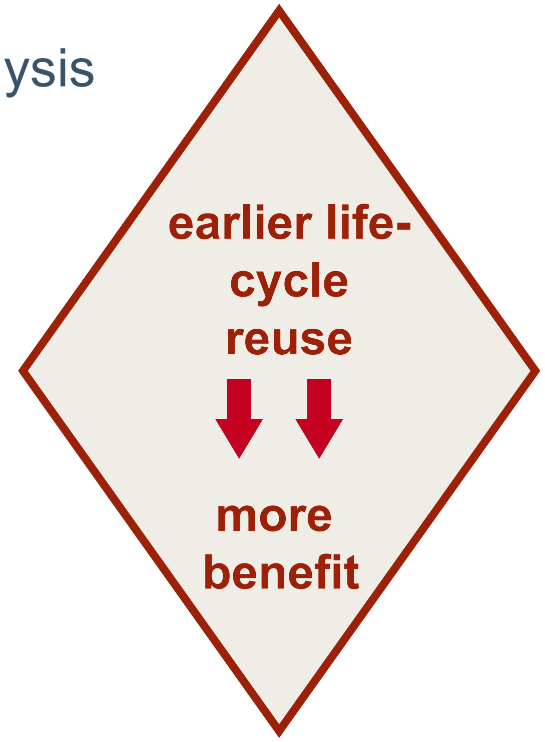


Product lines
- take economic advantage of commonality
- bound variability

How Do Product Lines Help?

Product lines amortize the investment in these and other *core assets*:

- requirements and requirements analysis
- domain model
- software architecture and design
- performance engineering
- documentation
- test plans, test cases, and data
- people: their knowledge and skills
- processes, methods, and tools
- budgets, schedules, and work plans
- components

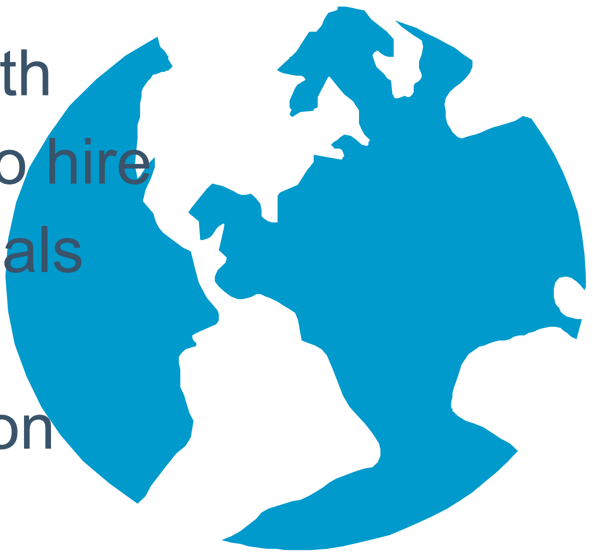


product lines = strategic reuse

Real World Motivation

Organizations use product line practices to:

- achieve large scale productivity gains
- improve time to market
- maintain market presence
- sustain unprecedented growth
- compensate for an inability to hire
- achieve systematic reuse goals
- improve product quality
- increase customer satisfaction
- enable mass customization
- get control of diverse product configurations

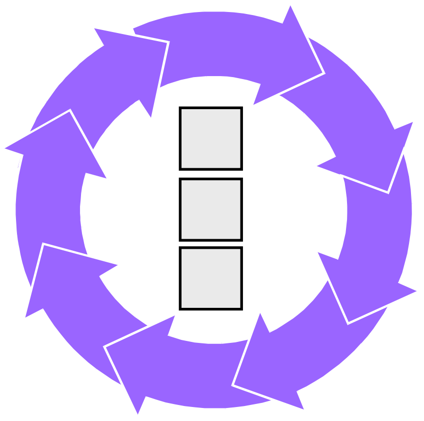


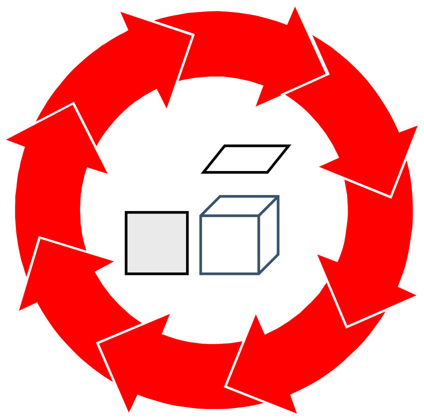
The Key Concepts

Use of a common asset base

in production

of a related set of products



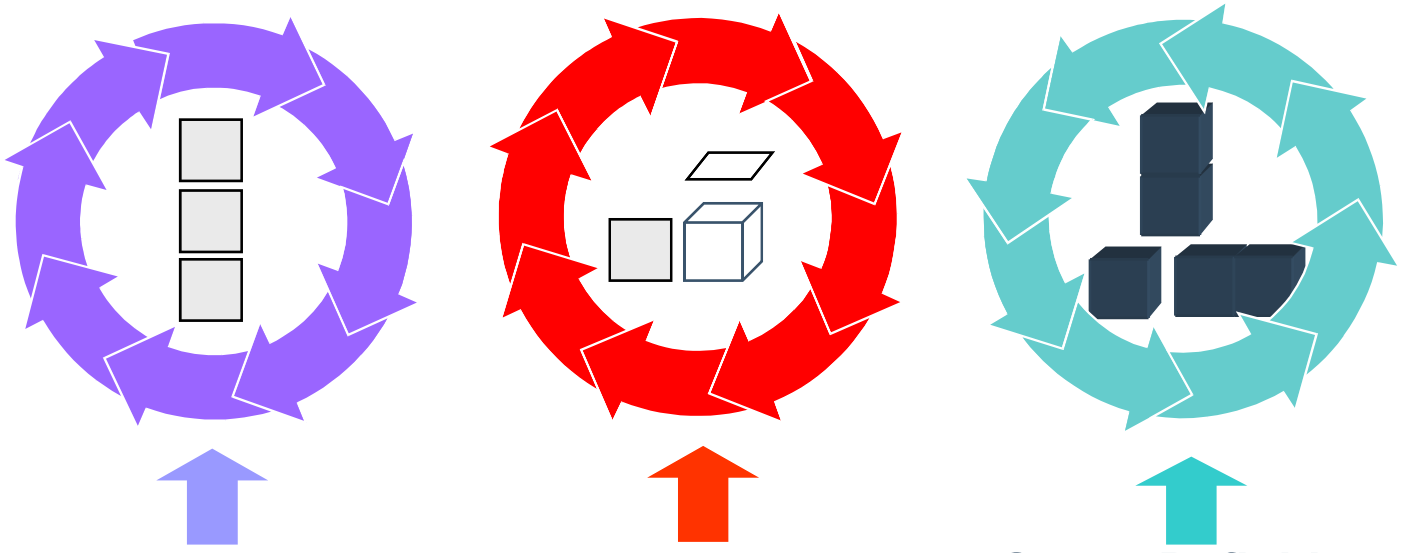


The Key Concepts

Use of a common asset base

in production

of a related set of products



Architecture

Production Plan

Scope Definition
Business Case

Commercial Examples

Successful software product lines have been built for families of

- Mobile phones
- Command and control ship systems
- Ground-based spacecraft systems
- Avionics systems
- Pagers
- Engine control systems
- Billing systems
- Web-based retail systems
- Printers
- Consumer electronic products
- Acquisition management enterprise systems

Organizational Benefits

Improved productivity
by as much as 10x

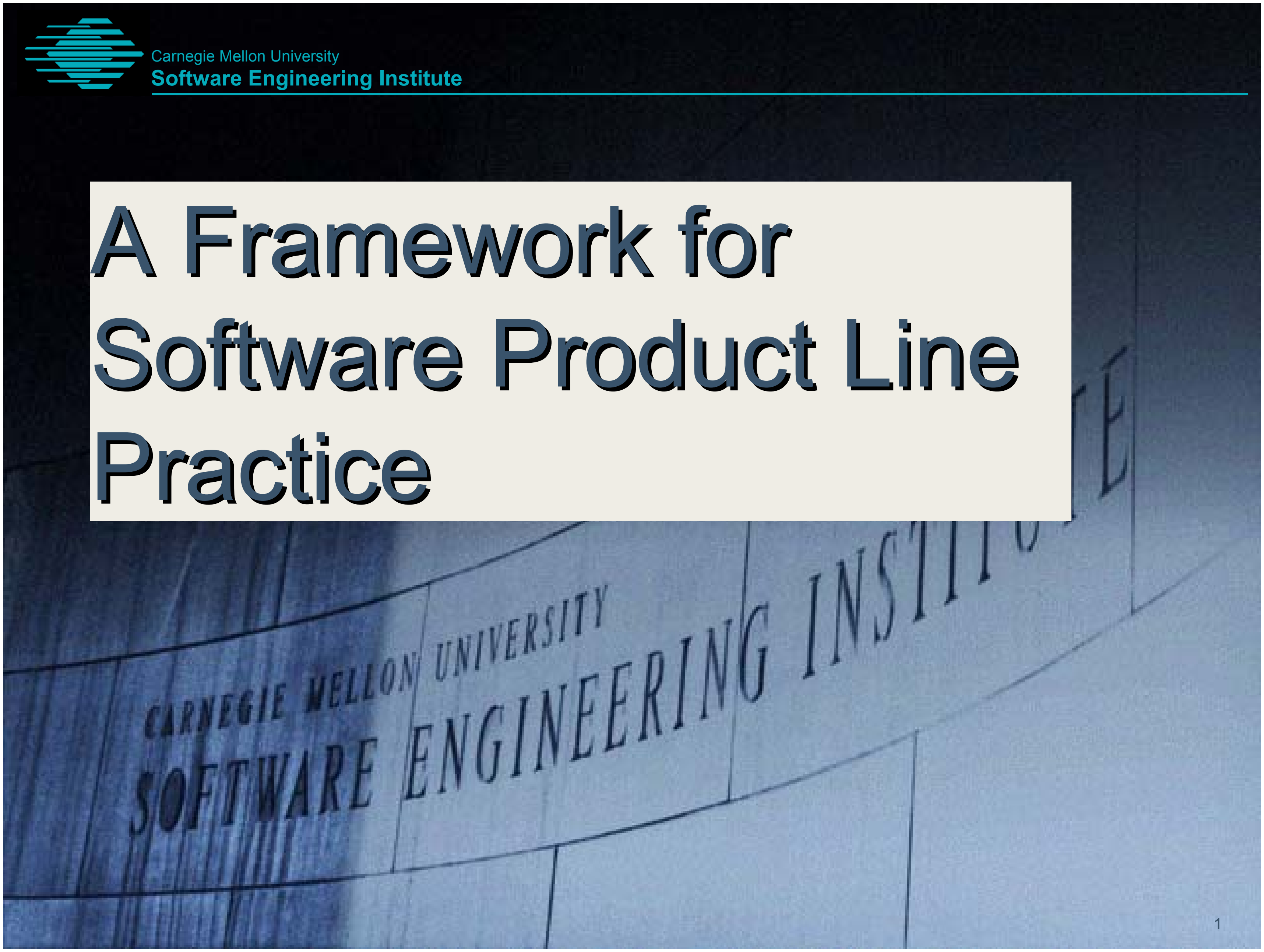
Decreased time to market (to field, to launch...)
by as much as 10x

Decreased cost
by as much as 60%

Decreased labor needs
by as much as 10X fewer software developers

Increased quality
by as much as 10X fewer defects

Carnegie Mellon University
Software Engineering Institute



A Framework for Software Product Line Practice

Product Line Practice

Contexts for product lines vary widely

- nature of products
- nature of market or mission
- business goals
- organizational infrastructure
- workforce distribution
- process discipline
- artifact maturity

But there are universal essential activities and practices.

A Framework for Software Product Line Practice

Three essential activities…
- core asset development
- product development
- management

…and the descriptions of the product line practice areas form a conceptual framework for software product line practice.

This framework is evolving based on the experience and information provided by the community.

Version 4.0: *Software Product Lines: Practices and Patterns*

Version 4.2: http://www.sei.cmu.edu/plp/framework.html

Information Sources

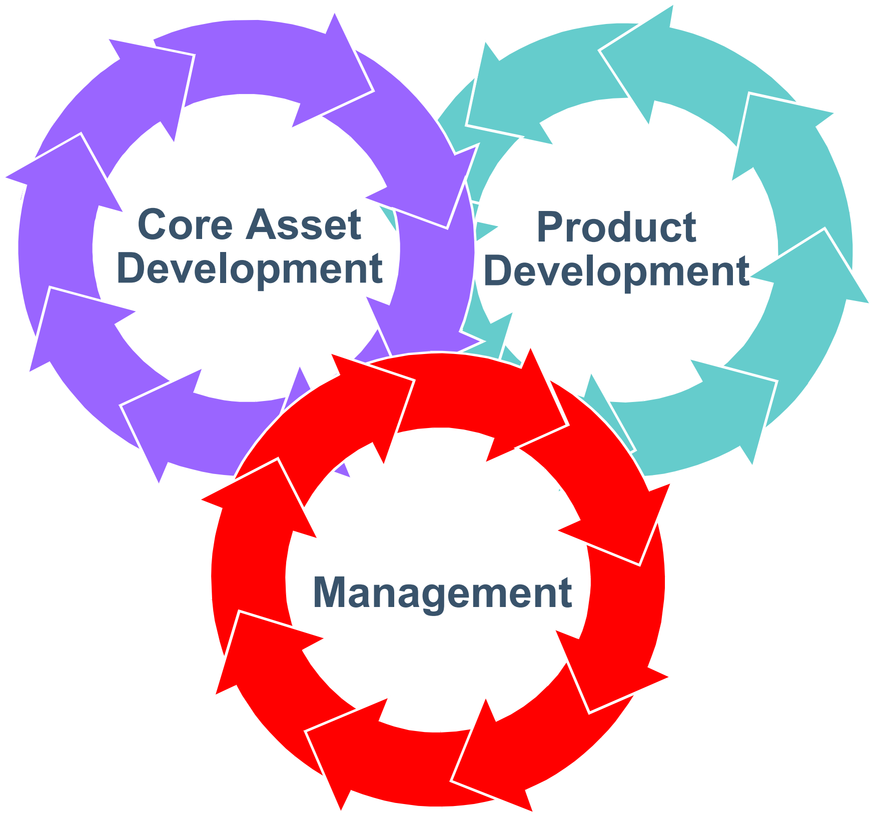
Case studies,
experience reports,
and surveys

Workshops,
and
conferences

Applied research

Collaborations
with customers
on actual product lines

Three Essential Activities



The Nature of the Essential Activities

All three activities are interrelated and highly iterative.

There is no “first” activity.

- In some contexts, existing products are mined for core assets.
- In others, core assets may be developed or procured for future use.

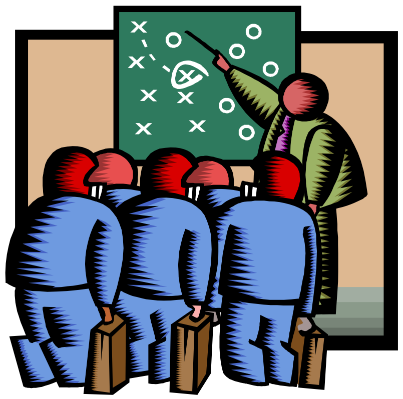
There is a strong feedback loop between the core assets and the products.

Strong management at multiple levels is needed throughout.

Management

Management at multiple levels plays a critical role in the successful product line practice by

- achieving the right organizational structure
- allocating resources
- coordinating and supervising
- providing training
- rewarding employees appropriately
- developing and communicating an acquisition strategy
- managing external interfaces
- creating and implementing a product line adoption plan



Managing a Software Product Line Requires Leadership

A key role for a software product line manager is that of champion.

The champion must

- set and maintain the vision
- ensure appropriate goals and measures are in place
- “sell” the product line up and down the chain
- sustain morale
- deflect potential derailments
- solicit feedback and continuously improve the approach

Different Approaches - 1

Proactive: Develop the core assets first

- Develop the scope first and use it as a “mission” statement.
- Products come to market quickly with minimum code-writing.
- Requires upfront investment and predictive knowledge.

Reactive: Start with one or more products

- From these generate the product line core assets and then future products; the scope evolves more dramatically.
- Much lower cost of entry
- Architecture and other core assets must be robust, extensible, and appropriate to future product line needs

Different Approaches - 2

Incremental: Develop in stages with the plan from the beginning to develop a product line.

- Develop part of the core asset base, including the architecture and some of the components.
- Develop one or more products.
- Develop part of the rest of the core asset base.
- Develop more products.
- Evolve more of the core asset base.
- …..

Driving the Essential Activities

Beneath the level of the essential activities are essential practices that fall into practice areas.

A practice area is a body of work or a collection of activities that an organization must master to successfully carry out the essential work of a product line.

Product line experience yields important lessons

Lessons in software engineering

- architectures for product lines
- testing variable architectures and components
- importance of having and capturing domain knowledge
- managing variations
- important of large, pre-integrated chunks

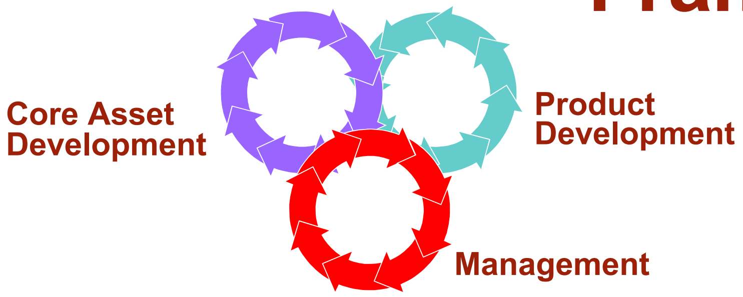
Lessons in technical/project management

- importance of configuration management, and why it's harder for product lines
- product line scoping: What's in? What's out?
- Tool support for product lines

Lessons in organizational management.

- People issues: how to bring about change, how to launch the effort
- Organizational structure: Who builds the core assets?
- Funding: How are the core assets paid for?
- Interacting with the customer has whole new dimension

Framework

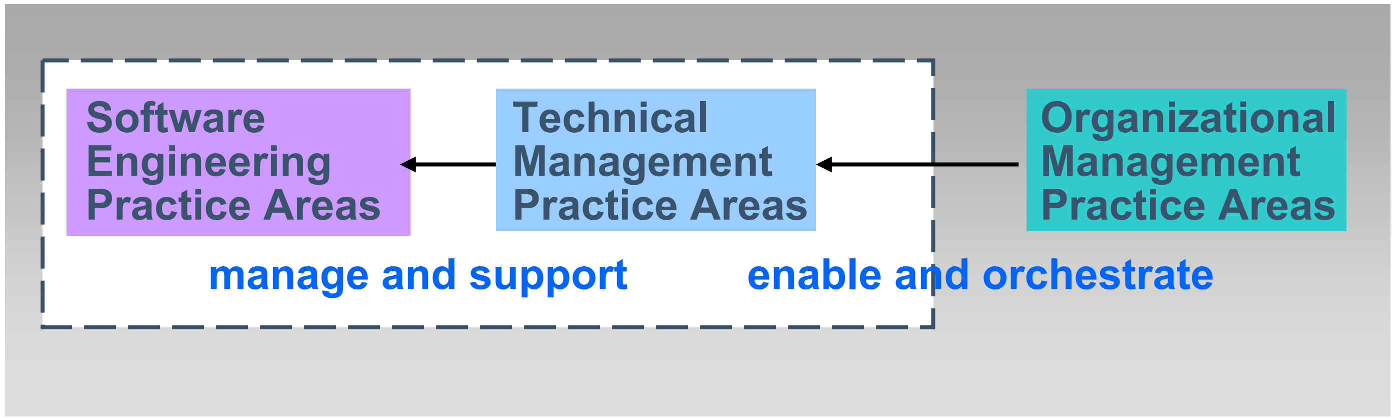


Essential Activities

Software Engineering	Technical Management	Organizational Management
Architecture Definition	Configuration Management	Building a Business Case
Architecture Evaluation	Data Collection, Metrics, and Tracking	Customer Interface Management
Component Development	Make/Buy/Mine/Commission Analysis	Implementing an Acquisition Strategy
COTS Utilization	Process Definition	Funding
Mining Existing Assets	Scoping	Launching and Institutionalizing
Requirements Engineering	Technical Planning	Market Analysis
Software System Integration	Technical Risk Management	Operations
Testing	Tool Support	Organizational Planning
Understanding Relevant Domains		Organizational Risk Management
		Structuring the Organization
		Technology Forecasting
		Training

Practice Areas

Relationships among Categories of Practice Areas


Software Engineering Practice Areas
Technical Management Practice Areas
Organizational Management Practice Areas
manage and support
enable and orchestrate

Scoping

Scoping bounds a system or set of systems by defining those behaviors or aspects that are in and those that are out.

All system development involves scoping; there is no system for which everything is in.

In conventional development, scoping is usually done informally (if at all), as a prelude to the requirements engineering activity.

Scoping: Aspects Peculiar to Product Lines - 1

The overall goal is to define what's in and what's out.

Scope definition lets you determine if a proposed new product can be reasonably developed as part of the existing (or planned) product line.

We want to draw the boundary so the product line is profitable.

Scoping: Aspects Peculiar to Product Lines - 2

Proper scoping is critical to a successful product line:

- If the scope is too limited, there will be too few products to justify the investment in the core assets.
- If the scope is too large, the core assets will need to be impossibly general.
- If the scope encompasses the wrong products, the product line will not succeed.

Scoping: Aspects Peculiar to Product Lines - 3

The scope starts out broad and very general.

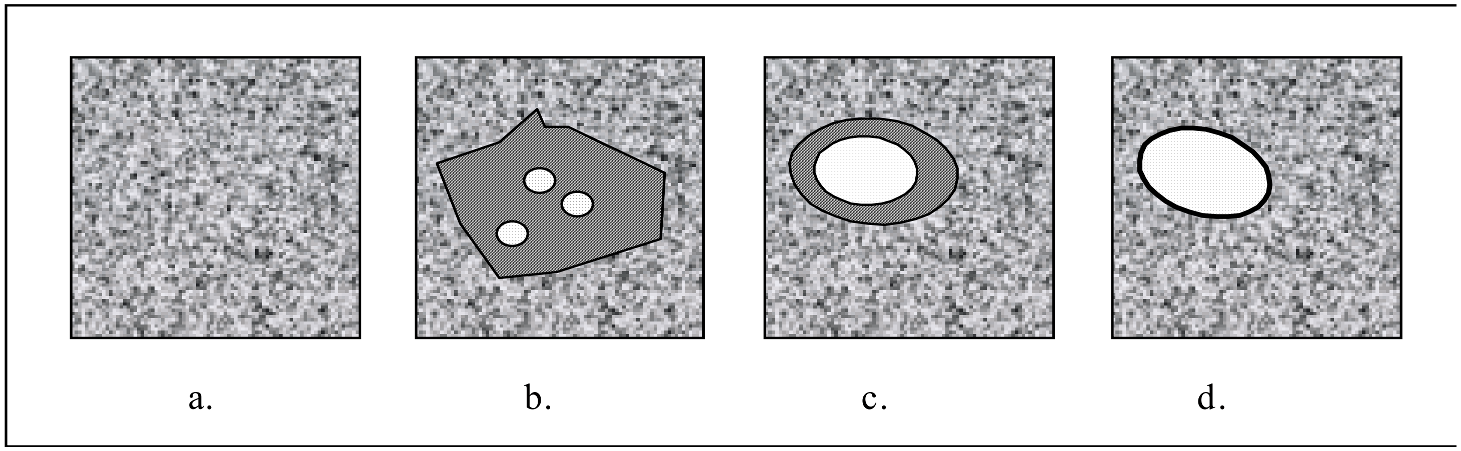
In a product line of Web software

- Browsers are definitely in.
- Aircraft flight simulators are definitely out.
- Email handlers are… well, we aren’t sure yet.

The scope grows more detailed as our knowledge increases and the product line matures.

Initially, many possible systems will be “on the cusp,” meaning their “in/out” decision must made on a case-by-case basis. That’s healthy.

Scope precision increases as we learn more…up to a point.



a: space of all possible products
b: early, coarse-grained “in/out” decisions
c: product line scope with a healthy area of indecision
d: product line scope = product line requirements

If many products appear on the cusp over time, you need to reactively adjust the scope.

Proactively Adjusting the Scope -1

Companies highly skilled at product line engineering routinely adjust their scope to take advantage of opportunities that are in the market.

CelsiusTech

- saw that air defense systems were just a short distance away in the product space from ship systems
- was able to enter the air defense market quickly and effectively. Forty percent of its air defense system was complete on day one.

Pro-actively Adjusting the Scope -2

Cummins, Inc.

- developed a software product line for automotive diesel engines
- saw a lucrative underutilized market nearby in industrial diesel engines
- was able to quickly enter and dominate the industrial diesel engine market

Motorola

- developed software product line for one-way pagers
- saw nearby market for two-way pagers and was able to use the same product line architecture for both

Key Themes Among Successful Product Lines

Sophistication in the domain

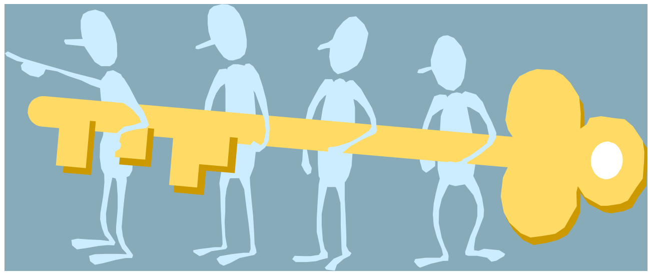
A legacy base from which to build

Architectural excellence

Process maturity

Management commitment

Capacity for introspection



Based on Our Experience

1. Product line business practices cannot be effected without management commitment and involvement.
2. Organization size doesn't matter.
3. Organizational culture plays a major role in adoption success.
4. Organizations need support: guidance, diagnostics, methods, and tools.
5. The lack of an architecture focus and/or talent can kill an otherwise promising product line effort.
6. Process discipline is critical.
7. The community needs more quantitative data to support product line adoption.
8. The cultural barriers and cost of adoption are major impediments to widespread transition.
9. Software product line practice is at the "chasm." (in Geoffrey Moore's terms: *Crossing the Chasm)*

The Time is Right

Rapidly maturing, increasingly sophisticated software development technologies including *object technology, component technology, standardization of commercial middleware.*

A global realization of the *importance of architecture*

A universal recognition of the need for *process discipline.*

Role models and case studies that are emerging in the literature and trade journals.

Conferences, workshops, and education programs that are now including product lines in the agenda.

Company and inter-company *product line initiatives.*

Rising recognition of the *amazing cost/performance savings* that are possible.

Final Examination

1. Produce four six-part quality attribute scenarios that express (one each):
   - modifiability
   - performance
   - security
   - availability

2. Produce a list of tactics to support usability.

3. Product a quality attribute utility tree for a system you are familiar with. Try to include at least five quality attributes. For each one, decompose it into quality attribute concerns, and one or two scenarios each.