



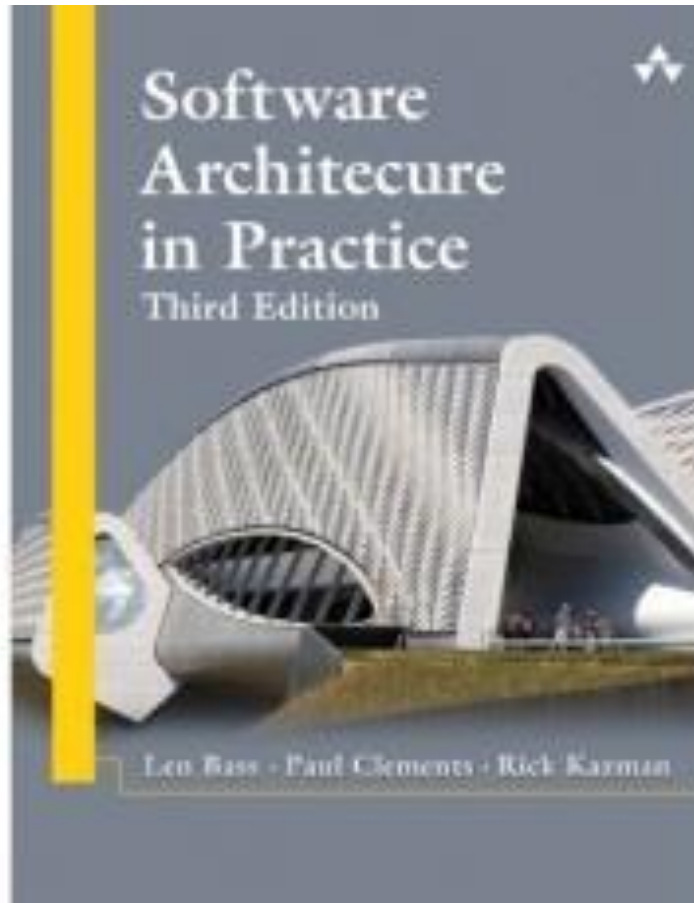
SE 4352

Software Architecture and Design

Fall 2018

Module 4

Chapters 2 & 4





Why Software Architecture Is Important

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

Inhibiting or Enabling a System's Quality Attributes

- Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.
- **This is the most important message of this course!**
 - **Performance:** You must manage the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
 - **Modifiability:** Assign responsibilities to elements so that the majority of changes to the system will affect a small number of those elements.
 - **Security:** Manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism).
 - **Scalability:** Localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hardcoding in resource assumptions or limits.
 - **Incremental subset delivery:** Manage inter-component usage.
 - **Reusability:** Restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment.

Reasoning About and Managing Change

- About 80 percent of a typical software system's total cost occurs after initial deployment
 - accommodate new features
 - adapt to new environments,
 - fix bugs, and so forth.
- Every architecture partitions possible changes into three categories
 - A *local* change can be accomplished by modifying a single element.
 - A *nonlocal* change requires multiple element modifications but leaves the underlying architectural approach intact.
 - An *architectural* change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system.
- Obviously, local changes are the most desirable
- A good architecture is one in which the most common changes are local, and hence easy to make.

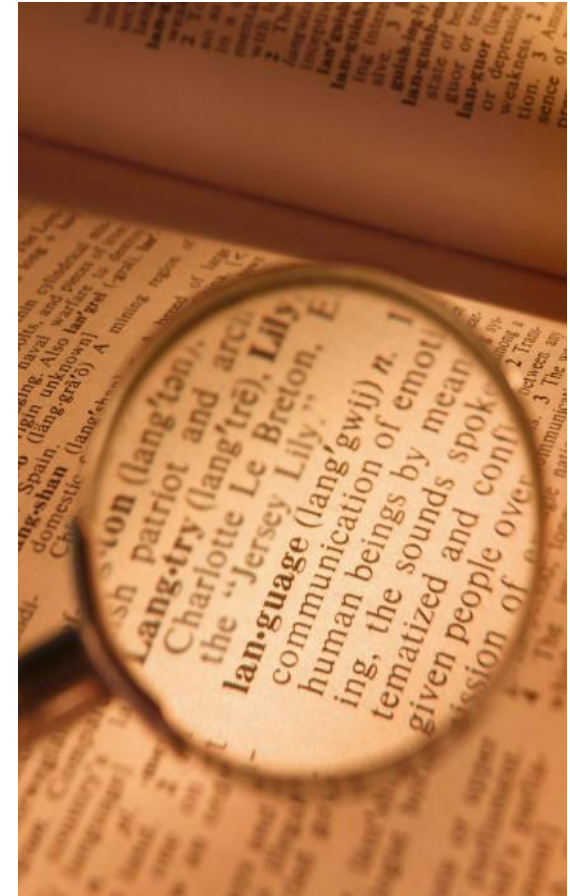
Predicting System Qualities

- If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, we can make those decisions and rightly expect to be rewarded with the associated quality attributes.
- When we examine an architecture we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.
- The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.



Enhancing Communication Among Stakeholders

- Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems.



Earliest Design Decisions

- Architecture is more than just the early decisions
- But they are critical
- Imagine the nightmare of having to change any of these decisions.



Defining Constraints on an Implementation

- e.g. Software Developers may not be aware of the architectural tradeoffs—the architecture (or architect) simply constrains them in such a way as to meet the tradeoffs.



Influencing the Organizational Structure

- Architecture prescribes the structure of the system being developed.
- That structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization).

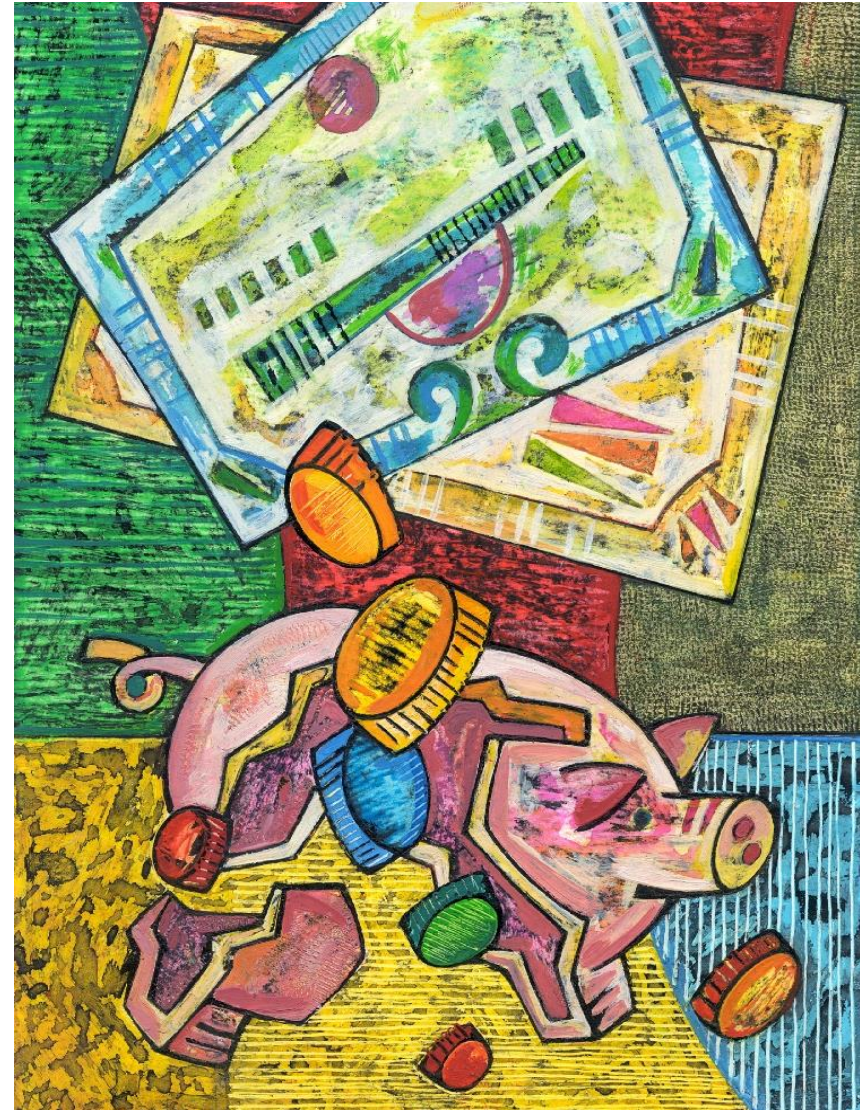


Enabling Evolutionary Prototyping



Improving Cost and Schedule Estimates

- One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle.



Transferable, Reusable Model

- Reuse of architectures provides tremendous leverage for systems with similar requirements.
 - Not only can code be reused, but so can the requirements that led to the architecture in the first place, as well as the experience and infrastructure gained in building the reused architecture.
 - When architectural decisions can be reused across multiple systems, all of the early-decision consequences are also transferred.



Using Independently Developed Components

- Architecture-based development often focuses on components that are likely to have been developed separately, even independently, from each other.
- The architecture defines the elements that can be incorporated into the system.
- Commercial off-the-shelf components, open source software, publicly available apps, and networked services are example of interchangeable software components.



Restricting Design Vocabulary

- As useful architectural patterns are collected, we see the benefit in voluntarily restricting ourselves to a relatively small number of choices of elements and their interactions.
 - We minimize the design complexity of the system we are building.
 - Enhanced reuse
 - More regular and simpler designs that are more easily understood and communicated
 - More capable analysis
 - Shorter selection time
 - Greater interoperability.
- Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design.
 - Properties of software design follow from the choice of an architectural pattern.

Basis for Training

- The architecture can serve as the first introduction to the system for new project members.




So...13 great reasons!!!





Why Software Architecture Is Important

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

- 
- Are these 13 reasons that determine the importance of architecture already in the order of priority?

Architecture and Requirements

- System requirements can be categorized as:
 - **Functional requirements.** These requirements state what the system must do, how it must behave or react to run-time stimuli.
 - **Quality attribute requirements.** These requirements annotate (qualify) functional requirements. Qualification might be how fast the function must be performed, how resilient it must be to erroneous input, how easy the function is to learn, etc.
 - **Constraints.** A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that has already been made for you.



Functionality

- Functionality is the ability of the system to do the work for which it was intended.
- Functionality has a strange relationship to architecture:
 - functionality does not determine architecture; given a set of required functionality, there is no end to the architectures you could create to satisfy that functionality
 - functionality and quality attributes are orthogonal

Quality Attribute Considerations

- If a functional requirement is "when the user presses the green button the Options dialog appears":
 - a **performance** QA annotation might describe how quickly the dialog will appear;
 - an **availability** QA annotation might describe how often this function will fail, and how quickly it will be repaired;
 - a **usability** QA annotation might describe how easy it is to learn this function.



Quality Attribute Considerations

- There are three problems with previous discussions of quality attributes:
 1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be “modifiable”.
 2. Endless time is wasted on arguing over which quality a concern belongs to. Is a system failure due to a denial of service attack an aspect of availability, performance, security, or usability?
 3. Each attribute community has developed its own vocabulary.



Quality Attribute Considerations

- A solution to the first two of these problems is to use *quality attribute scenarios* as a means of characterizing quality attributes.
- A solution to the third problem is to provide a discussion of each attribute—concentrating on its underlying concerns—to illustrate the concepts that are fundamental to that attribute community.





Specifying Quality Attribute Requirements

- We use a common form to specify all quality attribute requirements as scenarios.
- Our representation of quality attribute scenarios has these parts:
 1. **Stimulus**
 2. **Stimulus source**
 3. **Response**
 4. **Response measure**
 5. **Environment**
 6. **Artifact**

Specifying Quality Attribute Requirements

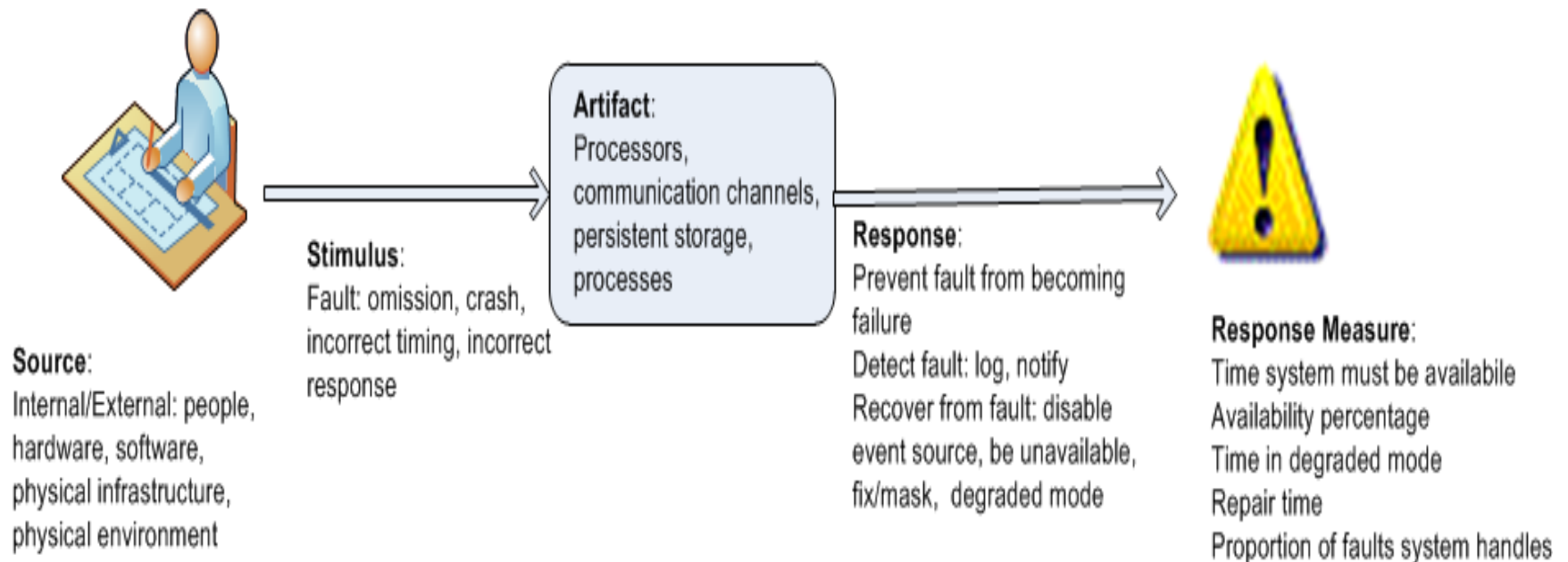
1. **Source of stimulus.** This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. **Stimulus.** The stimulus is a condition that requires a response when it arrives at a system.
3. **Environment.** The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, “normal” operation can refer to one of a number of modes.
4. **Artifact.** Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. **Response.** The response is the activity undertaken as the result of the arrival of the stimulus.
6. **Response measure.** When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Specifying Quality Attribute Requirements

- *general* quality attribute scenarios (“general scenarios”)
 - those that are system independent and can, potentially, pertain to any system
- *concrete* quality attribute scenarios (concrete scenarios)
 - those that are specific to the particular system under consideration.

Specifying Quality Attribute Requirements

- Example general scenario for availability:





Achieving Quality Attributes Through Tactics

- There are a collection of primitive design techniques that an architect can use to achieve a quality attribute response.
- We call these architectural design primitives *tactics*.
- Tactics, like design patterns, are techniques that architects have been using for years. We do not *invent* tactics, we simply capture what architects do in practice.



Achieving Quality Attributes Through Tactics

- Why do we do this? There are three reasons:
 1. Design patterns are complex; they are a bundle of design decisions. But patterns are often difficult to apply as is; architects need to modify and adapt them. By understanding tactics, an architect can assess the options for augmenting an existing pattern to achieve a quality attribute goal.
 2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles".
 3. By cataloguing tactics, we make design more systematic. You frequently will have a choice of multiple tactics to improve a particular quality attribute. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement.

So...how do we get started making architectural decisions?



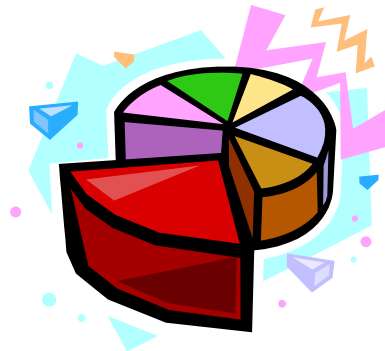
Guiding Quality Design Decisions

- Architecture design is a systematic approach to making design decisions.
- We categorize the design decisions that an architect needs to make as follows:
 1. Allocation of responsibilities
 2. Coordination model
 3. Data model
 4. Management of resources
 5. Mapping among architectural elements
 6. Binding time decisions
 7. Choice of technology



Allocation of Responsibilities

- Decisions involving allocation of responsibilities include:
 - identifying the important responsibilities including basic system functions, architectural infrastructure, and satisfaction of quality attributes.
 - determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).



Coordination Model

- Decisions about the coordination model include:
 - identify the elements of the system that must coordinate, or are prohibited from coordinating
 - determine the properties of the coordination
 - choose the communication mechanisms
 - stateful vs. stateless,
 - synchronous vs. asynchronous,
 - guaranteed vs. non-guaranteed delivery,
 - performance-related properties such as throughput and latency



Data Model



- Decisions about the data model include:
 - choosing the major data abstractions, their operations, and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed.
 - metadata needed for consistent interpretation of the data
 - organization of the data. This includes determining whether the data is going to be kept in a relational data base, a collection of objects or both

Management of Resources



- Decisions for management of resources include:
 - identifying the resources that must be managed and determining the limits for each
 - determining which system element(s) manage each resource
 - determining how resources are shared and the arbitration strategies employed when there is contention
 - determining the impact of saturation on different resources.

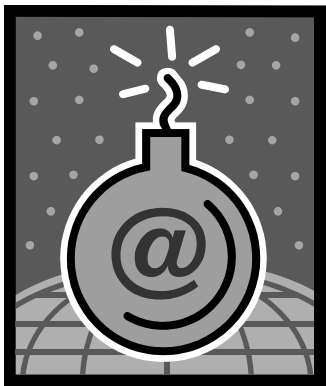
Mapping Among Architectural Elements



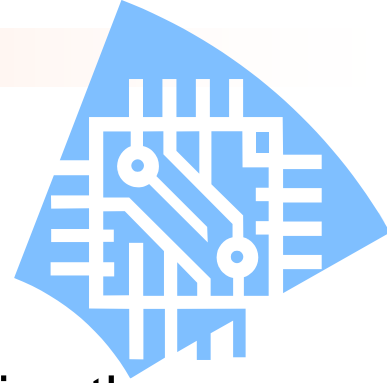
- Useful mappings include:
 - the mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element
 - the assignment of runtime elements to processors
 - the assignment of items in the data model to data stores
 - the mapping of modules and runtime elements to units of delivery

Binding Time

- The decisions in the other categories have an associated binding time decision. Examples of such binding time decisions include:
 - For allocation of responsibilities you can have build-time selection of modules via a parameterized build script.
 - For choice of coordination model you can design run-time negotiation of protocols.
 - For resource management you can design a system to accept new peripheral devices plugged in at run-time.
 - For choice of technology you can build an app-store for a smart phone that automatically downloads the appropriate version of the app.



Choice of Technology



- Choice of technology decisions involve:
 - deciding which technologies are available to realize the decisions made in the other categories
 - determining whether the tools to support this technology (IDEs, simulators, testing tools, etc.) are adequate
 - determining the extent of internal familiarity and external support for the technology (such as courses, tutorials, examples, availability of contractors)
 - determining the side effects of choosing a technology such as a required coordination model or constrained resource management opportunities
 - determining whether a new technology is compatible with the existing technology stack



Example

- Consider a programming language say **Java**
- What are some binding time decisions?
- What are some allocation structures?
- How are resources managed?



InClass Group Project

- Teams of 4 to 5
- Consider the choice between synchronous and asynchronous medium of instruction at a university. What qualities might lead you to choose one over another?
- Write down team answer
- Turn in for display to all students

