Principles of Software Architecture Design

Len Bass
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pa USA

Sponsored by the U.S. Department of Defense
© 2009 by Carnegie Mellon University
Outline

Background
- Responsibilities as design description
- Importance of quality attributes to the design process
- Specification of quality attributes
Architecturally significant requirements and utility tree
Design as generate and test
- Creating initial hypothesis
- Testing the hypothesis
- Generating alternatives for next hypothesis
Summary
Responsibilities as Design Description

A responsibility is a general statement about an architectural element and include the actions an element performs, the knowledge an element maintains, and major decisions an element makes that affect others.

Responsibilities are widely used in O-O design

Responsibility Examples

An Automated Teller Machine (ATM) provides money to authorized users who have sufficient funds on deposit.

Responsibilities can be refined
• An ATM requires a user to provide a PIN as an authorization
• Currency can be provided in multiple denominations
• No currency is distributed until the ATM has received confirmation from the bank’s computer system.
• etc
Types of Requirements

Constraints – pre-specified design decisions
Responsibilities that are visible to the user (e.g. what the system does). Also called “functional requirements”
Quality Attribute – how well the system does by various measures (e.g., how timely, secure, modifiable, deployable it is). Also called “non-functional requirements”
Constraints

Constraints – pre-specified design decisions (e.g. what the system does) given as a portion of the requirements.

Software Architectures

Constraints reduce the space of architectures in which to search for a solution.
Must live within constraints

Very little software design is “greenfield”

Frameworks, large-grained components are frequently required.

Disciplined design must accommodate constraints.

Designer does not make design decisions to achieve constraints – constraints are given.

Designer makes design decisions to achieve other requirements within given constraints.
Functional and quality attribute requirements

Functional requirements have related quality attribute requirements. May be explicit or implicit.

For example: A bank client can use an automated teller machine (ATM) to withdraw money, transfer money, or check the balance of an account.

Implicit quality attribute requirements:
- Performance – how long does a transaction take?
- Availability – what are the hours of operation?
- Security – how is the identity of the client determined?
- Usability – is the client able to cancel the operation?
- Modifiability – how long will it take to change the authentication mechanism?
Do functional or quality attribute requirements drive architectural design?

Quality requirements – not functional requirements – determine most architectural design decisions

If the only concern is functionality then a monolithic system would suffice.

However is it quite common to see:
- Redundancy structures for reliability
- Concurrency structures for performance
- Layers for modifiability
Quality Attribute Requirement Elicitation

One problem resulting from the conjunction of
• the importance of quality attribute requirements in design and
• the current requirement practice of not paying much attention to quality attribute requirements

is that the architect is not initially given sufficient information for the design process and must determine (or guess) the quality attribute requirements.

This problem is out of scope for this tutorial.
Specifying Quality Attribute Requirements

Two problems:
1. Quality attribute requirements are not appropriately specific
2. A common approach to specifying quality attributes is through taxonomies
Quality Attribute requirements are not appropriately specific

Requirements such as “the system shall be modifiable”, are meaningless.

What does it mean to say “the system shall be modifiable”? Must state “modifiable with respect to what?”
Taxonomies do not help

Common approach is to say a quality is divided into (ISO9126)
  • Functionality
  • Reliability
  • Usability
  • Efficiency
  • Maintainability
  • Portability

In order to use a taxonomy, a specific requirement must be placed into a category.
What category are the following?

Denial of service attack

Response time for user request

System A must be fielded within six months.
Broad categories must be used carefully

They are useful in establishing a vocabulary and frame of reference

They are useful in generating ideas during requirements elicitation

They are not useful if requirements must be placed into exactly one category
We have a common form for specification of quality requirements

We use **quality attribute general scenarios**, which are system independent, to guide the specification of quality attribute requirements.

We characterize quality attribute requirements for a specific system by a collection of **concrete quality attribute scenarios**. These are instances of general scenarios.

We use **general scenario generation tables** to construct well-formed general scenarios for each attribute.
General Scenarios

General scenarios have six parts. The “values” for each part define a vocabulary for articulating quality attribute requirements. The parts are:

- Stimulus
- Source of stimulus
- Environment in which the stimulus arrives
- Artifact influenced by the stimulus
- Response of the system to the stimulus
- Response measures

General scenario generation tables are a codification of possible general scenarios.
### Availability Scenario Generation Table

<table>
<thead>
<tr>
<th>Source of stimulus:</th>
<th>Stimulus:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Internal to the system</td>
<td>✓ Unanticipated event</td>
</tr>
<tr>
<td>✓ External to the system</td>
<td>• Update to a data store</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment:</th>
<th>Artifact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Normal operation</td>
<td>✓ Process</td>
</tr>
<tr>
<td>• Degraded mode</td>
<td>• Persistent storage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response:</th>
<th>Response measures:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ record it</td>
<td>✓ Availability percentage</td>
</tr>
<tr>
<td>✓ notify parties</td>
<td>• Time range in which the system can be in degraded mode</td>
</tr>
<tr>
<td>• operate in normal or degraded mode</td>
<td></td>
</tr>
</tbody>
</table>

**Example Scenario:**

"An unanticipated message is received by a system process during normal operation. The process has to record it, inform the appropriate parties and continue to operate in normal mode without any downtime."
Constructing Quality Requirements from General Scenarios

Generate a possible general scenario by choosing one or more entries from each list and combining them. Not all:
- general scenarios are relevant to specific system
- generated scenarios make sense

Make each scenario system specific (concrete scenario)

May be multiple concrete scenarios for each general scenario. e.g., modify function.

Eliminate duplicates
Which attributes?

We have lists for six quality attributes:
- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability

EXERCISE – we will choose an attribute and create a general scenario generation table
Outline

Background
- Responsibilities as design description
- Importance of quality attributes to the design process
- Specification of quality attributes

Architecturally significant requirements and utility tree

Design as generate and test
- Creating initial hypothesis
- Testing the hypothesis
- Generating alternatives for next hypothesis

Summary
Architecturally Significant Requirements - 1

Architecturally significant requirements (ASRs) are the requirements that impact the structure of the design and should be the primary focus when doing architectural analysis.

The ASR concept derives from our experience with ATAM (Architecture Tradeoff Analysis Method). ATAM uses architecture description from “30,000 ft” level. This perspective enables an understanding of what drove the architect to create the design being evaluated.
Architecturally Significant Requirements – 2

When creating an architecture, the goal is to determine what those “driving” requirements are.

RUP refers to Architecturally Significant Use Cases (same concept)

Recall that quality attribute requirements are the ones that drive the design => Architecturally significant requirements are quality attribute requirements.
Utility Tree - 1

Quality attribute utility trees provide a mechanism for translating the business drivers of a system into concrete quality attribute scenarios.

A utility tree lists
- The quality attributes for the particular system being designed as one level of the tree.
- The quality attribute “concerns” as the next level.
- Quality attribute scenarios are the leaves of the tree

The utility tree at the leaves serves to make concrete the quality attribute requirements, forcing architect and customer representatives to define relevant quality attributes precisely.
Utility Tree - 2

The leaves are prioritized in two dimensions
• The importance to the business of the scenario (H, M, L)
• The pervasiveness within the architecture of the requirements (H, M, L)

Those scenarios rated high importance and high difficulty provide the most critical context against which the architecture can be analyzed. These scenarios are candidates for the ASRs.
Utility Tree - 3

Performance
- Data latency
- Transaction throughput
- New products
- Change COTS

Modifiability
- Change Web user interface
- Add CORBA middleware in < 20 person-months
- Change COTS SAV failures

Availability
- HAV failure
- COTS SAV failures
- Network failure detected and recovered in < 1.5 minutes

Security
- Data confidentiality
- Data integrity
- Credit card transactions are secure 99.999% of the time
- Customer DB authorization works 99.999% of the time

\[ L = \text{Low}, M = \text{Medium}, H = \text{High} \]
<table>
<thead>
<tr>
<th>QA</th>
<th>Dist.</th>
<th>QA Concern</th>
<th>Dist.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifiability</td>
<td>14.1%</td>
<td>new/revised functionality/components</td>
<td>6.4%</td>
</tr>
<tr>
<td>Performance</td>
<td>13.6%</td>
<td>operability (e.g., can do)</td>
<td>4.1%</td>
</tr>
<tr>
<td>Usability</td>
<td>11.4%</td>
<td>upgrade/add hardware components</td>
<td>3.9%</td>
</tr>
<tr>
<td>Maintainability</td>
<td>8.5%</td>
<td>response time/deadline</td>
<td>3.6%</td>
</tr>
<tr>
<td>Interoperability</td>
<td>7.8%</td>
<td>latency</td>
<td>3.2%</td>
</tr>
<tr>
<td>Security</td>
<td>7.3%</td>
<td>portable to other platforms</td>
<td>3.1%</td>
</tr>
<tr>
<td>Configurability</td>
<td>6.9%</td>
<td>operate intra-service (e.g., ship-to-ship)</td>
<td>2.8%</td>
</tr>
<tr>
<td>Availability</td>
<td>6.8%</td>
<td>ease of operation: can do within a time limit</td>
<td>2.7%</td>
</tr>
<tr>
<td>Reliability</td>
<td>5.7%</td>
<td>throughput</td>
<td>2.1%</td>
</tr>
<tr>
<td>Scalability</td>
<td>3.2%</td>
<td>resource utilization</td>
<td>1.9%</td>
</tr>
<tr>
<td>Testability</td>
<td>2.6%</td>
<td>failure recovery/containment</td>
<td>1.9%</td>
</tr>
<tr>
<td>Affordability</td>
<td>2.0%</td>
<td>flexibility (range of operation scenarios)</td>
<td>1.7%</td>
</tr>
<tr>
<td>Reusability</td>
<td>1.9%</td>
<td>graceful degradation</td>
<td>1.6%</td>
</tr>
<tr>
<td>Integrability</td>
<td>1.9%</td>
<td>compliance to standards/protocols</td>
<td>1.5%</td>
</tr>
<tr>
<td>Safety</td>
<td>1.1%</td>
<td>affordability of various decisions (e.g., opennes)</td>
<td>1.5%</td>
</tr>
<tr>
<td>User data management</td>
<td>1.0%</td>
<td>replace COTS</td>
<td>1.4%</td>
</tr>
<tr>
<td>Portability</td>
<td>0.8%</td>
<td>real time</td>
<td>1.4%</td>
</tr>
<tr>
<td>Assurance</td>
<td>0.8%</td>
<td>fault tolerance</td>
<td>1.3%</td>
</tr>
<tr>
<td>Product line</td>
<td>0.8%</td>
<td>discovery (new configuration)</td>
<td>1.3%</td>
</tr>
<tr>
<td>Net-centric operation</td>
<td>0.5%</td>
<td>authentication</td>
<td>1.3%</td>
</tr>
</tbody>
</table>

Outline

Background
• Responsibilities as design description
• Importance of quality attributes to the design process
• Specification of quality attributes
Architecturally significant requirements and utility tree

Design as generate and test
• Creating initial hypothesis
• Testing the hypothesis
• Generating alternatives for next hypothesis

Summary
Design as Generate and Test

Design is the process of
• generating a hypothesis,
• testing that hypothesis,
• generating a new hypothesis, and
• repeating until hypothesized design passes the tests

Several questions result from this view of the design process
• Where does the initial hypothesis come from?
• What does it mean to test a hypothesis?
• Where does the new hypothesis come from?
Initial Design Hypothesis - 1

The initial design hypothesis comes from one of four sources (in order of preference):

1. From similar successful systems to that being built. Successful systems similar to the one being constructed have dealt with most of the issues facing the current system.
2. From a legacy system. If the current system is an extension to a legacy system, then the initial hypothesis comes from the legacy system and the next hypothesis will deal with problems raised through the testing process.
Initial Design Hypothesis - 2

3. A collection of frameworks and pre-existing components. If the system is going to be largely created from frameworks and pre-existing components, then the initial hypothesis consists of these frameworks and components connected with empty connectors. The testing process will determine how the connectors get filled in.

4. A pattern. Multiple patterns exist both in books and on the web. These patterns present solutions to recurring problems. If a pattern exists that can satisfy one of the architecturally significant requirements, then this provides a starting place. Such patterns typically, however, will not address more than one of the architecturally significant requirements. Different patterns can be chosen to address each architecturally significant requirement but then the patterns must be composed to get an overall pattern.
5. From first principles of quality attributes—we will discuss tactics later in this tutorial.
6. From functional/logical view. In this case, the testing will disclose missing quality attribute requirements that needs to be addressed.
Exercise – pattern as a basis for initial hypothesis

Choose a pattern you are familiar with.

What does the pattern tell you?

What does the pattern leave out?

How do you resolve the things that are left out?
Outline

Background
• Responsibilities as design description
• Importance of quality attributes to the design process
• Specification of quality attributes

Architecturally significant requirements and utility tree

Design as generate and test
• Creating initial hypothesis
• Testing the hypothesis
• Generating alternatives for next hypothesis

Summary
Output of test stage

We will test the hypothesis with a collection of test cases.

The output of the tests will be
• Additional responsibilities that need to be addressed
• List of quality attribute problems

Source of test cases
• Architecturally significant requirements
• Quality attribute specific use cases

Testing technique
• Understanding of quality attributes
Architecturally significant requirements as test cases

Architecturally significant requirements are the ones that the architecture design must satisfy.

As such, they are obvious test cases for any design hypothesis.
Quality Attribute Specific Use Cases - 1

There are a collection of quality attribute use cases that should be used as test cases in addition to the architecturally significant requirements.

Based on consideration of functionality:
- expected operation exercising major capabilities
- exceptions
- growth and exploratory scenarios
- deferred binding time
- version upgrades
- modification scenarios

Look for:
- Allocation of responsibilities to modules
- Responsibilities associated with exception management
- Responsibilities associated with deferred binding time.
Quality Attribute Specific Use Cases - 2

Based on consideration of parallelism:
- two users doing similar tasks simultaneously
- one user performing multiple activities simultaneously
- start-up (creating threads that must be in waiting mode, initializing connected devices, etc.)
- shutdown (cleaning up similar finishing activities, storing data, etc.)

Look for
- Points of resource contention (synchronization),
- Opportunities for parallelism (creation of new threads)
- Necessity for killing processes (deleting threads)
- Additional responsibilities to manage points of contention and clean up
- Management of user state
Quality Attribute Specific Use Cases - 3

Based on consideration of multiple processors

- installation
- initialization
- processing across processors
- messaging over the network
- disconnected operation
- failure of an element (e.g., process, processor, network)

For each use case

- Determine desired policy
- Determine mechanisms to achieve desired policy
- Determine responsibilities to implement chosen mechanisms
Exercise

Consider the pattern that we used for the initial hypothesis

Exercise that pattern with the use case of disconnected operation.

What new responsibilities are discovered?
Output of test stage

We will test the hypothesis with a collection of test cases.

The output of the tests will be
- Additional responsibilities
- List of quality attribute problems

Source of test cases
- Architecturally significant requirements
- Quality attribute specific use cases

Testing techniques
- Understanding of quality attributes
Understanding Performance - 1

Typical performance measures are the
- Time from the arrival of an event to the end of the processing of that event (latency)
- The number of events that can be processed in a unit time (throughput)

The time to process an event is key in both measures

Two basic contributors to the time to process an event are
- Time spent consuming resources and
- Time spent blocked.
Understanding Performance - 2

Resources include
- central processing unit (CPU),
- data stores,
- network communication bandwidth, and
- memory.

A computation can be blocked from using a resource because
- there is contention for the resource,
- the resource is unavailable, or
- the computation depends on the result of other computations that are not yet available.
Testing for Performance

- Consider event that enters system
- Identify computation requirements
- Identify potential blockages
- Consider event that enters system and travels between processors
- Identify computation requirements
- Identify potential blockages

Use architecturally significant performance requirements to identify source of events, sequence of responsibilities, and compare to desired latency or throughput.
Understanding Modifiability - 1

Modifiability has to do with the cost of change to the software.

There must be a human involved in the modification.

Cost gets complicated. There is
• The cost of preparing for a change to be made
• The cost of making an individual change
• The probability that a set of changes will be made
• Opportunity cost

We will focus on the cost of making an individual change
Understanding Modifiability - 2

Any modification involves
• Changing a responsibility
• Adding a new responsibility
• Deleting a responsibility
• Moving a responsibility from one module to another

A modification may affect data flow

A modification may affect additional responsibilities that have dependencies on the responsibility being modified.

The cost of a modification becomes the cost of modifying a responsibility plus the cost of modifying any dependent responsibilities.
Testing for Modifiability

Consider architecturally significant modifiability requirements.

Consider responsibilities and dependencies among responsibilities
• Determine the direct cost of modifying responsibilities as specified by the architecturally significant modifiability requirement
• Determine which other responsibilities must be modified because they are dependent on the directly modified responsibilities. This is called impact analysis.

Compare cost of modification to requirement given by the architecturally significant modifiability requirement.
**Exercise**

Choose a quality attribute other than performance or modifiability.

What are the fundamental concepts associated with that quality attribute?

How would you measure the quality attribute?
Summary of hypothesis testing

Use architecturally significant requirements to identify specific portions of the current hypothesis that are relevant to the satisfaction of these requirements.

Use understanding of quality attributes to determine whether the architecturally significant requirements are satisfied by the current hypothesis.

Use quality attribute specific use cases and architecturally significant requirements to determine additional responsibilities that should be included in the next hypothesis.
Outline

Background
- Responsibilities as design description
- Importance of quality attributes to the design process
- Specification of quality attributes

Architecturally significant requirements and utility tree

Design as generate and test
- Creating initial hypothesis
- Testing the hypothesis
- Generating alternatives for next hypothesis

Summary
Generating alternatives for next hypothesis

Early design decisions

Tactics

Tactics -> responsibilities

Generating new hypothesis from tactics and responsibilities
Exercise

What system are you working on?

What is your favorite quality attribute?

What questions would you ask to determine how the design impacts the quality attribute?

What techniques would you apply to achieve the quality attribute?
What is Architectural Design?

Architectural design is making the decisions that lead to the creation of architecture.

• The design for a system consists of a collection of decisions.
• Decisions made early constrain ones made later.
• Make decisions early that have the farthest reaching impact.
Early Architectural Design Decisions

The coordination model

The data model

Allocation of functionality

Management of run time resources

Mapping among architectural elements

Binding time of the decisions in the other categories
The Coordination Model

What are the communication mechanisms between the system and external entities?

What are the inter-element communication mechanisms and what are their properties (e.g., synchronous, asynchronous, hybrid coupling)?

What are the intra-element communication mechanisms?
The Data Model

What is the *structure*---elements and relations, element attributes---in the data model?

Which portions of the data model are *used by* which software elements in which *order*?

What are the *access rules* for the data items?

Where are data items *created*, *modified*, and *destroyed*?
Allocation of Functionality

What are the *major processing steps* necessary to carry out the work of the system?

What is the *division and assignment of functionality* to software elements?

What are the *key abstractions* that can be used to provide the services of the system?

Are the elements *stateful or stateless*?

What are the *activation and deactivation dependencies* among software elements?
Management of Run Time Resources

What *scheduling strategies* will be employed?

How much do system elements know about *time*?

What *process/thread models* will be employed?

What *resources* need to be *managed* and what are their *limits*?
Mapping Among Architectural Elements

How are modules mapped to runtime elements?

How are runtime elements mapped to processors?

How are units of development mapped to configuration items in the development environment?
Binding Time Decisions

The decisions made above can be bound at a variety of times.

- Design time (built in)
- Compile time (e.g., compiler switches)
- Build time (e.g., replace modules, pick from library)
- Load time (e.g., dynamic link libraries [DLLs])
- Initialization time (e.g., resource files)
- Run time (e.g., load balancing)
## Management of Run Time Resources in the Context of Performance

<table>
<thead>
<tr>
<th>Coordination model</th>
<th>Availability</th>
<th>Modifiability</th>
<th>Performance</th>
<th>Security</th>
<th>Testability</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation of functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Management of run time resources</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping among architectural elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Questions (to understand impact):
- what is overhead of creating/deleting threads?
- how much activation/deactivation will be done?
- how mobile are threads?
- can threads be pre-allocated?
- can threads be dynamically allocated?
- how many task switches are anticipated? overhead?

### Techniques:
- thread pool
- database connection pool
- time frame skipping
- compression
- buffering
- scheduling
Exercise

For the system and quality attribute you chose in the previous exercise, what does this quality attribute mean in terms of each of the early design decisions?

<table>
<thead>
<tr>
<th></th>
<th>Availability</th>
<th>Modifiability</th>
<th>Performance</th>
<th>Security</th>
<th>Testability</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Management of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>run time resources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping among</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>architectural</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Choose one of the existing quality attributes or insert your favorite here.

Go down the column and for each design decision consider:
• How does this decision impact the quality attribute?
• What techniques can I use?
Architectural Tactic

An architectural tactic is a design decision that improves the performance of the system with respect to a single quality attribute.

Ideally, tactics are based on models of the quality attribute. Recall:

- performance has to do with execution time and blocked time
- modifiability has to do with dependency among responsibilities
Queuing Model for Performance

Parameters:
- Arrival rate
- Queuing discipline
- Scheduling algorithm
- Service time
- Topology
- Network bandwidth
- Routing algorithm

Latency can be affected only by changing one of the parameters.
Controlling Performance Parameters

Architectural means for controlling the parameters of a performance model

- **Arrival rate** – restrict access, differential rate/charging structure, constrain message size
- **Queuing discipline** – first-come first served (FCFS), priority queues, etc.
- **Service time**
  - Increase efficiency of algorithms.
  - Cut down on overhead (reduce inter-process communication, use thread pools, use pool of DB connections, etc.).
  - Use faster processor.
- **Scheduling algorithm** – round robin, service last interrupt first, etc.
- **Topology** – add/delete processors
- **Network bandwidth** – faster networks
- **Routing algorithm** – load balancing
Performance Tactic Categories

Performance tactic categories and their goals:

- *Resource demand* – Reduce or manage the demand for resources.
- *Resource management* – Manage resources even though the demand for resources is not controllable.
- *Resource arbitration* – Control contention for resources through scheduling.
Performance Tactics

Performance

Resource Demand
- Increase Computation Efficiency
- Reduce Computational Overhead
- Manage Event Rate
- Control Frequency of Sampling

Resource Management
- Introduce Concurrency
- Maintain Multiple Copies
- Increase Available Resources

Resource Arbitration
- Scheduling Policy
- Synchronization Policy
Dependency Analysis - 1

The Prevent Ripple Effects set of modifiability tactics is fundamentally about *dependency*:

- If module A depends on module B, modifications to module B may require modifications to module A.
- Dependencies can be either code or data related.
- The tactics for achieving modifiability are based on examining dependency among responsibilities and either
  - Co-locating in a module responsibilities with high dependency (increase cohestion)
  - Reduce dependencies among responsibilities in different modules
  - deferring binding time
## Dependency Analysis - 3

<table>
<thead>
<tr>
<th>Type of Change</th>
<th>Data Syntax</th>
<th>Data Semantics</th>
<th>Service Syntax</th>
<th>Service Semantics</th>
<th>Sequencing (Data)</th>
<th>Sequencing (Control)</th>
<th>Identity of an Interface</th>
<th>Existence</th>
<th>Location at Runtime</th>
<th>Quality of Service</th>
<th>Resource Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>B changes propagate to A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+ propagation might occur; - propagation will not occur.
Cost Model for Modifiability

Parameters:
- average cost of modifying a single responsibility
- coupling
- cohesion
- life cycle time of modification

Cost can be affected only by changing one of the parameters.
Controlling Modifiability Parameters

Architectural means for controlling the parameters of a modifiability model

- **average cost of modifying a single responsibility** – reducing the cost of modifying a single responsibility by splitting the responsibility into two portions based on the specific change made.
- **coupling** – reducing coupling by breaking or weakening dependencies.
- **cohesion** – increasing cohesion by moving responsibilities from one module to another.
- **life cycle time of modification** – reducing the cost by deferring bindings.
Modifiability Tactic Categories

Modifiability tactic categories and their goals:

- *Localize changes* – Reduce the number of modules directly affected by a change.
- *Prevention of ripple effect* – Limit modifications to localized modules.
- *Defer binding time* – Control deployment time and cost.
Modifiability Tactics

- Localize Changes
  - Semantic Coherence
  - Anticipate Expected Change
  - Generalize Module
  - Limit Possible Options
  - Abstract Common Services

- Prevent Ripple Effects
  - Hide Information
  - Maintain Existing Interfaces
  - Restrict Communication Paths
  - Use an Intermediary

- Defer Binding Time
  - Runtime Registration
  - Configuration Files
  - Polymorphism
  - Component Replacement
  - Adhere to Defined Protocols
A tactic is one (or more) of the following types of transformations:

- **modify responsibility.** The tactic increase message content can be achieved by modifying the responsibilities that construct messages to construct larger messages.
- **introduce new responsibilities.** The tactic introduce concurrency requires that responsibilities for forking the concurrent threads and joining those threads together be introduced.
- **Introduce new structural elements.** The tactic maintain multiple copies requires elements to store the new copy and maintain consistency among the copies.
Tactics are transformations on responsibilities and structure-2.

- modify the properties of a responsibility. The tactic reduce execution time will result in a modification of a property (execution time) of the responsibility that is being made more efficient.
- decompose responsibilities. The tactic maintain multiple copies will result in the responsibility of storing information into one location being decomposed (and augmented) into responsibilities that store the information and synchronize the information with other locations.
- reallocate responsibilities. The tactic reduce computational overhead may result in responsibilities being reallocated from one process into another to reduce interprocess communication.
Generate next hypothesis

At this point we have as a result of the test phase:

- Additional responsibilities as a result of responsibilities discovered through test cases or through tactics.
- Revised responsibilities as a result of decomposition.
- Constraints on the allocation of responsibilities to modules as a result of tactics.
- Other constraints on responsibilities such as budgeted execution time.

These responsibilities and their constraints are merged with the current hypothesis to generate the next hypothesis.
Exercise

Consider pattern we used as initial hypothesis earlier.

Consider responsibilities generated as a result of considering the test case of failure.

How would these responsibilities be allocated to create a new hypothesis?
Summary

Design is the process of generate and test.

The initial design is generated from existing or similar systems, frameworks and components, or patterns.

The design is tested against the architecturally significant requirements and a collection of quality attribute use cases to derive additional responsibilities and constraints on responsibilities.

The design is analyzed against quality attribute models to discover shortcomings.

Tactics are used to propose alternatives for improving the design.

The next hypothesis is generated based on additional responsibilities and constraints discovered during test and analysis.