Design for Modifiability

主讲教师： 王灿

Email: wcan@zju.edu.cn

TA: 李奇平 liqiping1991@gmail.com

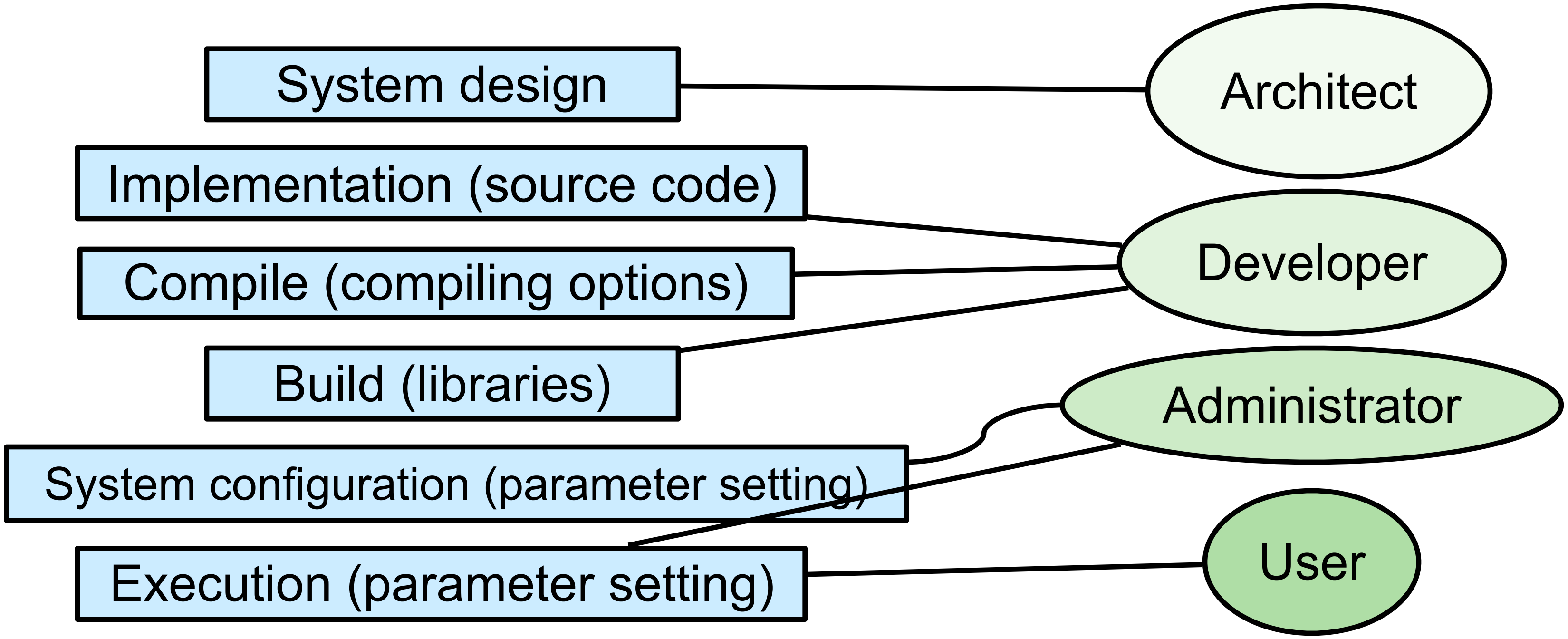
Course FTP: ftp://sa:sa@10.214.51.13

Modifiability (1)

- Software change is constant and ubiquitous
- Modifiability is about change
- Four major concerns:
    - What can change?
        - The function the system computes
        - The platform it exists on
            - Portability
        - Qualities of the system
        - Capacity of the system

Modifiability (2)

- When is the change made and who makes it?

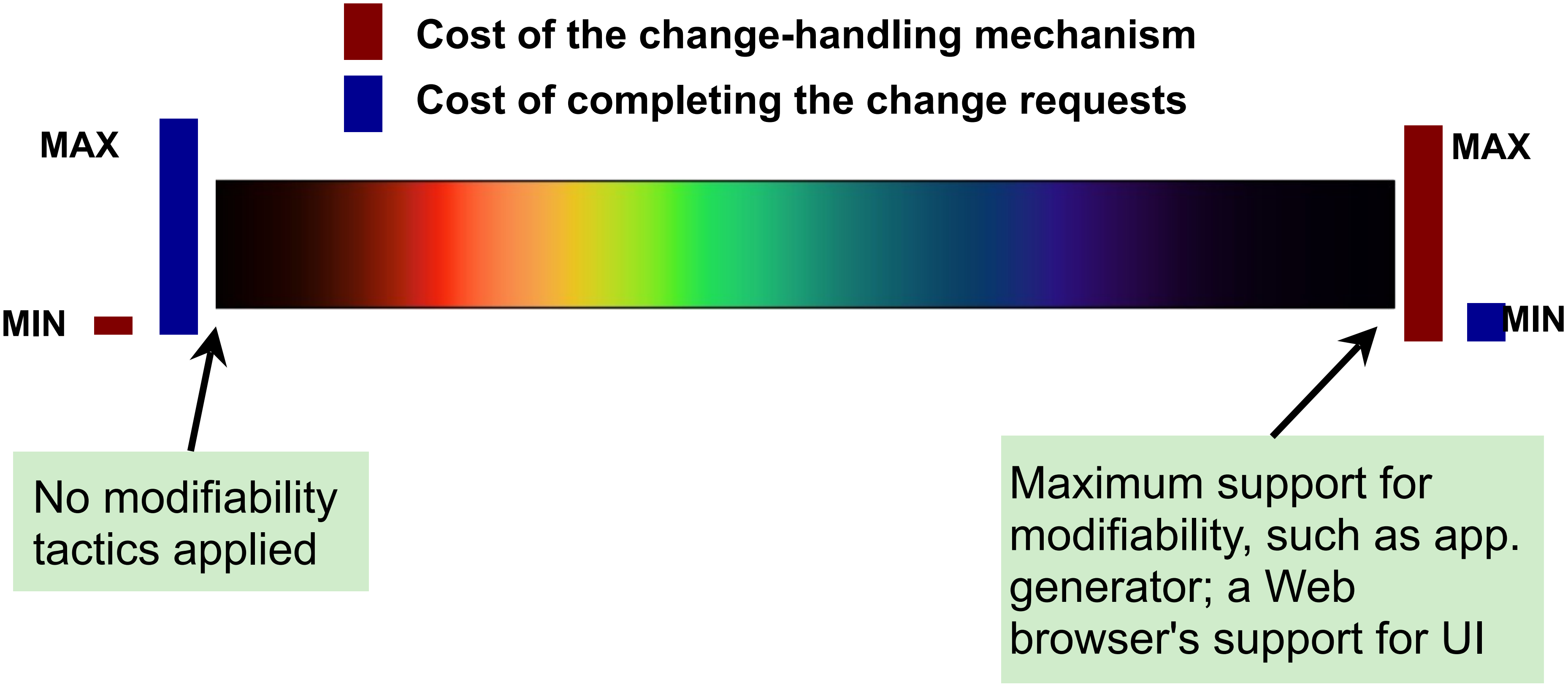

System design
Implementation (source code)
Compile (compiling options)
Build (libraries)
System configuration (parameter setting)
Execution (parameter setting)
Architect
Developer
Administrator
User

  - Once a change has been specified, the new implementation must be designed, implemented, tested, and deployed.

Modifiability (3)

- What is the likelihood of the change?
  - Architectural supports shall go to the parts most likely to change, or more precisely, supports the parts whose changes will incur the highest costs.
- What is the cost of the change?
  - The cost of introducing the change-handling mechanism(s)
  - The cost of making the modification using the change-handling mechanism(s)

Two Ends of the Modifiability Spectrum



🟥 Cost of the change-handling mechanism

🟦 Cost of completing the change requests

No modifiability tactics applied

Maximum support for modifiability, such as app. generator; a Web browser's support for UI

A Simple Equation for Planning Change-handling Mechanism

- For N similar modifications, a simplified justification for a change-handling mechanism:
  - *N* x Cost of making the change without the mechanism < Cost of installing the mechanism + (N x Cost of making the change using the mechanism)
  - N is the expected number of modifications

Change-planning equation

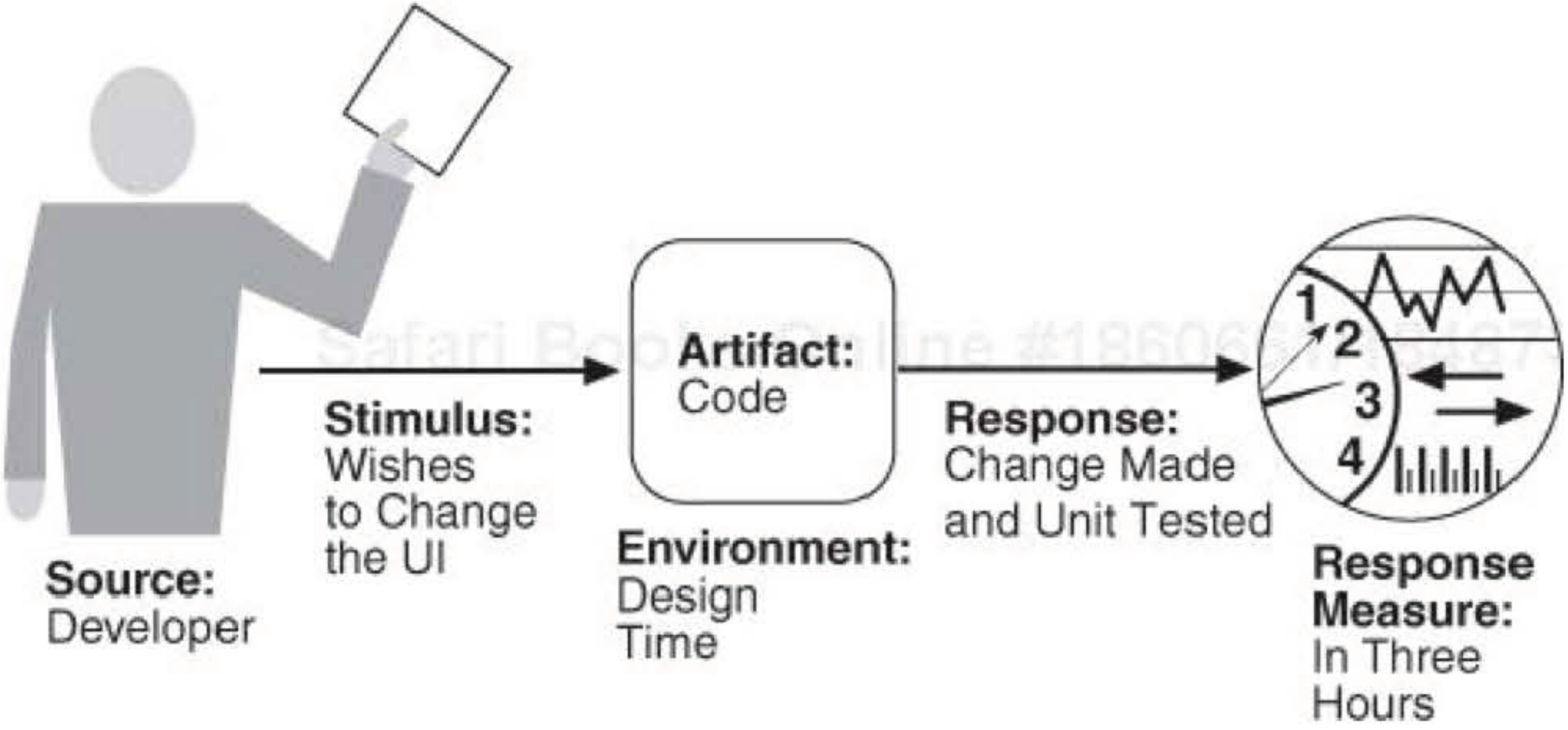
Modifiability General Scenario (1)

- Stimulus: changes to be made
  - Functions
  - Quality attributes
  - Capacities
  - …
- Source of stimulus
  - End user
  - System administrator
  - Developer
  - …

Modifiability General Scenario (2)

- Response
  - Make, test and deploy the change
- Response
  - Time and money
  - Number of elements (modules, defects) affected,
- Artifact
  - Any aspect of a system
- Environment
  - Design, compile, build time
  - Initiation time
  - Runtime

A Sample Concrete Modifiability Scenario



Source: Developer

Stimulus: Wishes to Change the UI

Artifact: Code

Environment: Design Time

Response: Change Made and Unit Tested

Response Measure: In Three Hours

Coupling

- Coupling measures the overlap of two modules by measuring the probability that a modification to one module will propagate to the other
  - If two modules' responsibilities overlap in some way, then a single change may well affect them both
  - Tight coupling is an enemy of modifiability

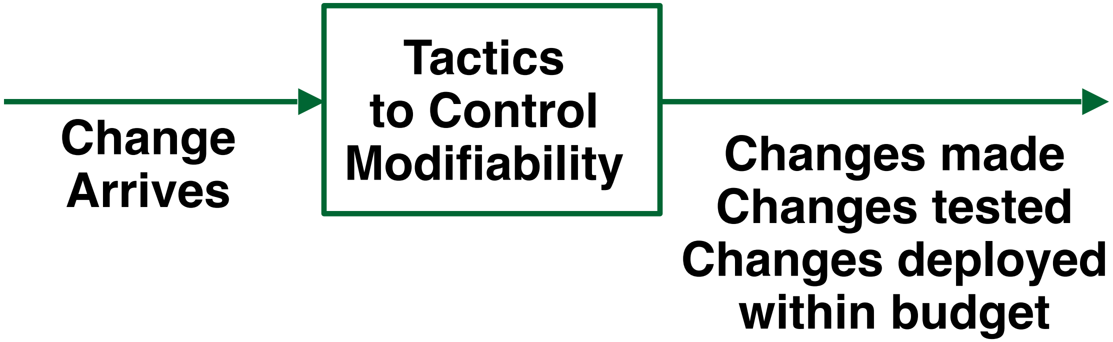
Cohesion

- Cohesion measures how strongly the responsibilities of a module are related
  - Cohesion is the probability that a change scenario that affects a responsibility will also affect other (different) responsibilities
  - “Unity of purpose”

Parameters for Controlling Modifiability

- Size of a module
- Coupling
- Cohesion

Modifiability Tactics (1)



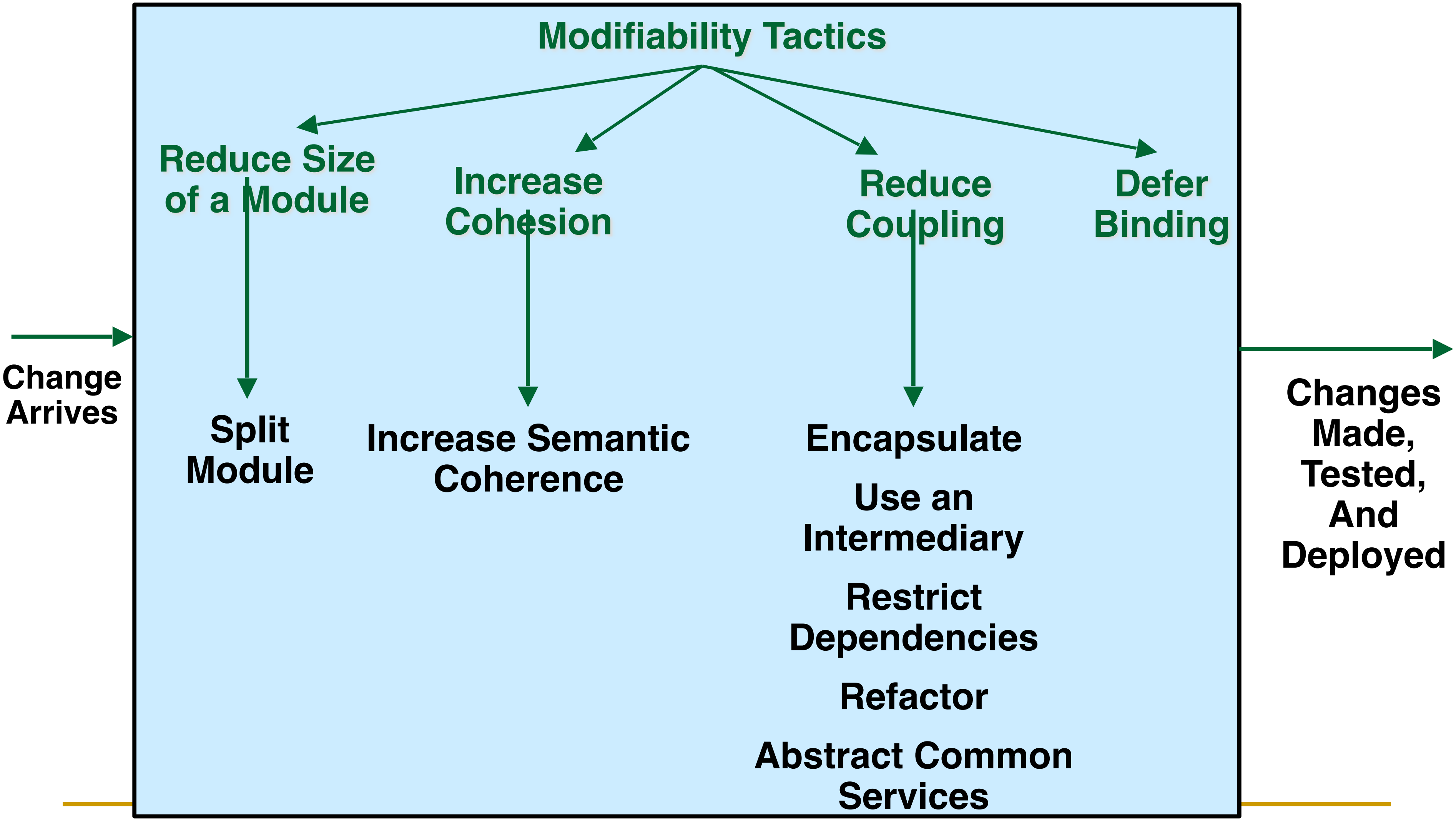
Change Arrives

Tactics to Control Modifiability

Changes made
Changes tested
Changes deployed within budget

- Goals: controlling the time and cost to implement, test, and deploy changes.

Modifiability Tactics (2)



Change Arrives

Modifiability Tactics

Reduce Size of a Module

Split Module

Increase Cohesion

Increase Semantic Coherence

Reduce Coupling

Encapsulate

Use an Intermediary

Restrict Dependencies

Refactor

Abstract Common Services

Defer Binding

Changes Made, Tested, And Deployed

Reduce the Size of a Module: Split Module

- If the module being modified includes a great deal of capability, the modification costs will likely be high
  - Refining the module into several smaller modules should reduce the average cost of future changes

Increase Cohesion: Increase Semantic Coherence

- Semantic coherence: the relationships among responsibilities in a module
  - Keep things that are related together (serve the same purpose)
- If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules
  - Creating a new module
  - Moving a responsibility to an existing module

Reduce Coupling: Encapsulate

- Encapsulate: visible API + transparent implementation

- Interface limits the ways in which external responsibilities can interact with the module
  - The external responsibilities can now only directly interact with the module through the exposed interface

Reduce Coupling: Use an Intermediary

- Intermediaries are used to break a dependency between responsibility A and responsibility B, e.g.:
  - Directory service in service invocation
  - Data repository separates readers of a piece of data from writers of that data
  - Memory handles for runtime memory location
  - Resource manager for resource contention
  - Publish-subscribe pattern removes the data producer's knowledge of its consumers

Reduce Coupling: Restrict Dependencies

- Restricts the modules that a given module interacts with or depends on
  - Restricting a module's visibility
  - Restricting access to only authorized modules
- Examples
  - In layered pattern, a layer is only allowed to use lower layers
  - In the use of wrappers, external entities can only see (and hence depend on) the wrapper and not the internal functionality that it wraps

Reduce Coupling: Refactor

- Refactoring disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior
- Code refactoring avoids duplicative or overly complex code using a series techniques
  - Extract method, *generalize type*, encapsulate field, *pull up*...

Reduce Coupling: Abstract Common Services

- Keep a service required by multiple clients in one place
  - Modifications need not be made separately
- Parameterizing module's activities to introduce abstraction
  - Simple parameters VS. specialized language

Defer Binding (1)

- The later in the life cycle we can bind values, the more flexibility we have to handle modification
  - Late binding usually are more expensive to implement
- Defer binding tactics at compile time or build time:
  - Component replacement (for example, in a build script or make file)
  - Compile-time parameterization

Defer Binding (2)

- Tactics to bind values at deployment time include:
  - Configuration-time binding
- Tactics to bind values at startup or initialization time include:
  - Resource files

Defer Binding (3)

- Tactics to bind values at runtime include these:
  - Runtime registration
  - *Dynamic lookup* (e.g., for services)
  - Interpret parameters
  - Startup time binding
  - *Name servers*
  - *Plug-ins*
  - *Publish-subscribe*
  - Shared repositories
  - *Polymorphism*

Architectural Design Support for Modifiability

- We check the architectural design and analysis process for modifiability from the following 7 aspects:
  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

- Determine the changes that are likely to occur
  - The responsibilities affected by the change
  - The corresponding responsibilities to be added/ modified/deleted

  - Responsibility allocation shall put responsibilities likely to be changed together in one place

Coordination Model

- Similar to responsibility considerations, we determined coordination that are likely to be affected by the change, *esp*. those occur at runtime
- Consider coordination models that reduce coupling for these parts whose modifiability is a concern

Data Model

- Determine likely changes to the data model (abstractions/operations/properties)
- Determine the changes to data abstractions that will involve their creation, initialization, persistence, manipulation, translation, or destruction.
- Determine who will make these changes and check whether they have been granted proper privileges

Mapping among Architectural Elements

- Determine how the mapping between functionality and computational elements (e.g., processes, threads, processors) supports modifiability (at runtime, compile time, design time, or build time)

Resources Management

- Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage

Binding Time Decisions

- Determine the late binding mechanisms that can accommodate the change request
- Choose a defer-binding mechanism using the change-planning equation

Choice of Technology

- Determine how the technology helps to make, test, and deploy modifications
- Determine the cost of switching between alternative technologies

Reading Assignment

- Read Chapter 8 of the textbook.