Architectural Design IV

Designing the Module Structure
Design Principles

Architecture Design Process

Building architecture to address business goals:
1. Understand the goals for the system
2. Define the quality requirements
3. Design the architecture
   1. Views: which architectural structures should we use? (goals<->architectural structures<->representation)
   2. Documentation: how do we communicate design decisions?
   3. Design: how do we decompose the system?
4. Evaluate the architecture (is it a good design?)
Module Decomposition Strategies

• How do we develop this structure so that the leaf modules make independent work assignments?
  – Dependencies are few
  – Decisions that might change are encapsulated
  – Interfaces are simple and well defined
• I.e. low *coupling*, high *cohesion*
  – Coupling: degree of interdependence *between* software modules
  – Cohesion: degree of relation or interdependence of elements within a module

Modular Decomposition

• Design goals: modifiability, work assignments, maintainability, reusability, understandability, etc.
• Observed strategies only partially sucessful
  – Use-case driven OOD, heuristics
  – MVC Pattern
• What should be done differently?
  – Why did these approaches fail?
Use Case Driven OO Process

- Address book design: in-class exercise
- Requirements
- Problem Analysis
  - Identify use cases from requirements
  - Identify domain classes operationalizing use cases (apply heuristics)
- OO Design (refinement)
  - Allocate responsibilities among classes
  - Identify object interactions supporting use cases
  - Identify supporting classes (& associations)
- Detailed Design
  - Design class interfaces (class attributes and services)

Decomposition Heuristics

- Heuristics: suppose we create objects by …
  - Underline the nouns
  - Identify causal agents
  - Identify coherent services
  - Identify real-world items
  - Identify physical devices
  - Identify essential abstractions
  - Etc.
- Do the propertiers we want follow?
  - Conversely, is it possible to satisfy the heuristic but not get the properties we want (e.g. few dependencies)?
Design Principles

Modular Structure

- Architecture = components, relations, and interfaces
- Components
  - Called modules
  - Leaf modules are work assignments
  - Non-leaf modules are the union of their submodules
- Relations (connectors)
  - submodule-of => implements-secrets-of
    - Module is an aggregate of its submodules
    - Constrained to be acyclic tree (hierarchy)
- Interfaces (externally visible component behavior)
  - Defined in terms of access procedures (services or method)
  - Services provide only access to module internals
Module Hierarchy

Design Principles

- Principle (n): a comprehensive and fundamental rule, doctrine, or assumption
- Design Principles – rules that guide developers in making design decisions consistent with overall design goals and constraints
  - Guide the decision making process of design by helping choose between alternatives
  - Embodied in methods and techniques (e.g., for decompositions)
Three Key Design Principles

• Most solid first
• Information hiding
• Abstraction

Principle: Most Solid First

• View design as a sequence of decisions
  – Later decisions depend on earlier
  – Early decisions harder to change
• Most solid first: in a sequence of decisions, those that are least likely to change should be made first
• Goal: reduce rework by limiting the impact of changes
• Application: used to order a sequence of design decisions
  – Generally applicable to design decisions
  – Module decomposition – ease of change
Information Hiding

- Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following the design principles that (Parnas):
  - System details that are likely to change independently are put in different modules
  - The interface of a module reveals only those aspects considered unlikely to change
  - Details other modules should not depend on are encapsulated

Decomposition Strategy

- Decompose recursively
  - If a module holds decisions that are likely to change independently, then decompose it into submodules
  - Decisions that are likely to change together are allocated to the same submodule
  - Decisions that change independently should be allocated to different submodules
- Stopping criteria
  - Each module contains only things likely to change together
  - Each module is simple enough to be understood fully, small enough that it makes sense to throw it away rather than re-do
- Define the Interfaces
  - Anything that other modules should not depend on become secrets of the module (e.g., implementation details)
  - If the module has an interface, only things not likely to change can be part of the interface
Module Hierarchy

Problem

"Secrets"  "Secrets"  "Secrets"

"Secrets"  "Secrets"  "Secrets"

Interface  Interface  Interface
Encapsulated  Encapsulated  Encapsulated

Submodule-of relation

Leaf Modules = Work assignments

Effects of Changes

- Consider what happens to communication among module developers
- Suppose we have groups of requirements R1 – R3:
  - R1 and R3 are related and likely to change together
  - R2 is likely to change independently
- Suppose we put R1 and R2 in the same module and assign to different teams
  - What happens when R1 changes?
  - R2?
- Suppose R1 and R3 are put in the same module?
Abstraction

- General: disassociating from specific instances to represent what the instances have in common
  - Abstraction defines a *one-to-many relationship*
    E.g., one type, many possible implementations
- Modular decomposition: Interface design principle of providing only essential information and suppressing unnecessary detail

Abstraction

- Two primary uses
- Reduce Complexity
  - Goal: manage complexity by reducing the amount of information that must be considered at one time
  - Approach: Separate information important to the problem at hand from that which is not
    - Abstraction suppresses or hides “irrelevant detail”
    - Examples: stacks, queues, abstract device
- Model the problem domain
  - Goal: leverage domain knowledge to simplify understanding, creating, checking designs
  - Approach: Provide components that make it easier to model a class of problems
    - May be quite general (e.g., type real, type float)
    - May be very problem specific (e.g., class automobile, book object)
Address Book Reconsidered

- Consider address book design based on principles

Summary

- Heuristics and patterns are guidelines
  - Do not guarantee qualities
  - Must understand how and why they work to apply effectively
- Principles are more direct – achieve qualities by construction
- Good design requires careful thinking
  - Which goals are we trying to achieve
  - How design decisions address those goals
Lessons on Patterns

• Patterns are often misused
• Using a pattern correctly requires understanding it
  – “Correctly” – such that the pattern’s design goals are realized in your design
  – “Understanding” – you understand what the pattern is supposed to accomplish, how it works, and how to apply it in your context
Lessons on Patterns (2)

- A pattern is a three part rule that expresses a relation between [Schmidt]:
  1. A particular problem context
  2. A set of competing forces (goals and constraints) in that context
  3. A software configuration that resolves the set of forces
     - Configuration == objects, interfaces, relations
     - Resolves == concurrently addresses the goals and constraints