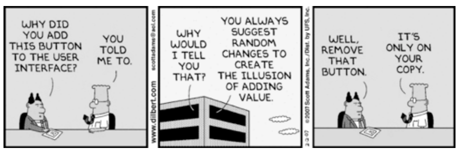
Designing the Module Structure

Designing a module structure
Address Book exercise



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?)

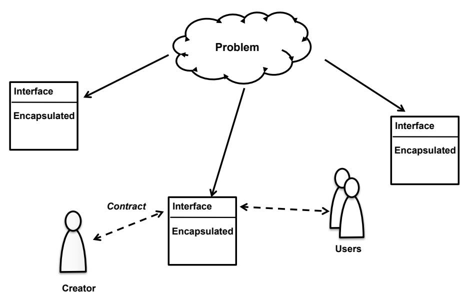
Modularization

- For any large, complex system, must divide the coding into work assignments (WBS)
- Each work assignment is called a “module”
- Properties of a “good” module structure
  - Parts can be designed independently
  - Parts can be tested independently
  - Parts can be changed independently
  - Integration goes smoothly

What is a module?

- Concept due to David Parnas (conceptual basis for objects)
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the systems
  - Its secrets: what the module hides (encapsulates). Design/implementation decisions that other parts of the system *should not depend on*
- Modules are abstract, design-time entities
  - Modules are "black boxes" – specifies the visible properties but not the implementation
  - May, or may not, directly correspond to programming components like classes/objects
    - E.g., one module may be implemented by several objects

Notional Modules



Designing the Module Structure

How do we design to arrive at the desired qualities?

Decomposition Strategies Differ

- How do we develop this structure so that the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Pipes and Filters: each module is a step in a chain of processing
  - Transactional: data transforming components
  - OOD: use case driven development
- Different approaches result in different kinds of dependencies

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
    - CRC Cards (Class-Responsibility-Collaboration)
  - Identify object interactions supporting use cases
    - Sequence or Interaction Diagram for each scenario
  - Identify supporting classes (& associations)
    - Design Class Diagram, relations
- 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
  - Identify transactions
  - Identify persistent information
  - Identify visual elements
  - Identify control elements
  - Execute scenarios

Address Book Design Exercise

- Is this a good design?
  - Walk through the handout to understand how the design is derived
    - Understand how use-case-driven OO design works
  - Walk through the design's class diagram and UML class specifications to understand the structure and function of the design
  - Discuss the good and bad points of the design to arrive a team judgment
  - Justify your answer: what is good about it (or bad) and why? What is the role of the MVC pattern?

General OO Objectives

- Manage complexity
- Improve maintainability
- Improve stakeholder communication
- Improve productivity
- Improve reuse
- Provide unified development model (requirements to code)

General OO Principles

- Principles provided to support goals
- Abstraction and Problem modeling
  - Development in terms of problem domain
  - Supports communication, productivity
- Generalization/Specialization (type of abstraction)
  - Inheritance of shared attributes & Delayed Binding (polymorphism)
  - Support for reuse, productivity
- Modularization and Information Hiding
  - Supports maintainability, reuse
- Independence (abstract interfaces + IH)
  - Classes designed as independent entities
  - Supports readability, reuse, maintainability
- Common underlying model
  - OO model for analysis, design, and programming
  - Supports unified development

Some Design Goals

- Be easy to make the following kinds of change
  - Add additional fields to the entries: for example, fields for someone's email, mobile phone, and business phone
  - Ability to edit the name fields at any time while keeping the associated data
  - As the number of entries gets larger, we will want to be able to search the address book
- Support subsets and extensions
  - Produce a simpler version of the address book with only names and phone #
  - Allow user to keep multiple address books of different kinds (i.e., different fields)
  - Allow the user-defined fields

A Decomposition Approach

Decomposition Strategies Differ

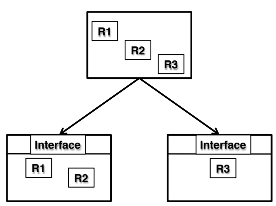
- How do we develop this structure so that *we know* the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Pipes and Filters: each module is a step in a chain of processing
  - Transactional: data transforming components
  - Client/server
  - Use-case driven development
- But, these result in different kinds of dependencies (strong coupling)

Submodule-of Relation

- To define the structure, need the *relation* and the *rule* for constructing the relation
- Relation: sub-module-of
- Rules
  - If a module holds decisions that are likely to change independently, then decompose it into submodules
  - Don't stop until each module contains only things likely to change together
  - 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

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?



Applied Information Hiding

- The rule we just described is called the *information hiding principle*
- 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:
  - System details that are likely to change independently are encapsulated in different modules
  - The interface of a module reveals only those aspects considered unlikely to change

Design Principles

Three Key Design Principles

- Address the basic issue: which constructs are essential to the problem solution vs. which can change
  - "Fundamental assumptions"
  - "Likely changes"
- 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
  - Developing families – create most commonality

Information Hiding

- 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 encapsulated in different modules
  - The interface of a module reveals only those aspects considered unlikely to change

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

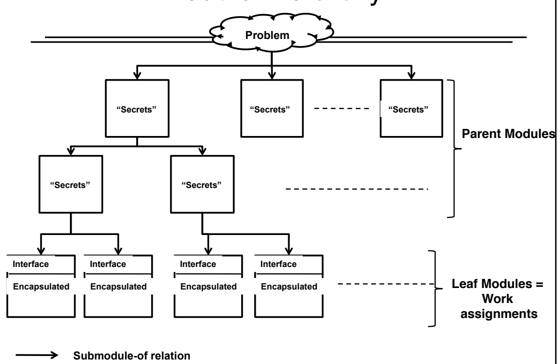
Abstraction

- 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)

Example: Simple Library Model

- What are the abstractions?
- What information is hidden?

Module Hierarchy



Comments

- Applying heuristics does not guarantee qualities
- Applying patterns requires understanding how the pattern works