Software Design Principles and Guidelines

Adam Porter

Overview

• Design Principles
  – Important design concepts
  – Useful design principles
• Design Guidelines
  – Motivation
  – Design Rules of Thumb
Goals of the Design Phase

• Identify the software architecture
• Decompose system into modules. *Modules* are program units that should:
  – be independent,
  – have well-specified interfaces, and
  – have high cohesion and low coupling
• Determine relationships between modules
  – Identify module dependencies
  – Determine the form and protocol for inter-module communication

Goals of the Design Phase (cont'd)

• Specify module interfaces. Interfaces should be well-defined
  – facilitate independent module testing
  – improve group communication
• Describe module functionality
  – Informally
    • *e.g.*, comments or documentation
  – Formally
    • *e.g.*, via module interface specification languages
Notional Design Phases

• Preliminary design
  – External design describes the real-world model
  – Architectural design decomposes the requirement specification into software subsystems
• Detailed design
  – Specify each subsystem
  – Further decompose subsystems, if necessary
• Note: in design phases the orientation moves
  – from customer to developer
  – From what to how

Key Design Concepts and Principles

• Important design concepts and design principles:
  – Decomposition: make big systems from smaller ones
  – Abstraction: suppress irrelevant details
  – Modularity: components should stand alone
  – Information Hiding: modules hide secrets
  – Hierarchy: limit inter-module dependencies
  – Separating Policy and Mechanism: what ≠ how
• Each concept helps manage software system complexity and improve software quality
Decomposition

• Principle: Design decomposes large systems into smaller pieces
• Basic concept is very simple:
  1. Select a piece of the problem (initially, all of it)
  2. Determine its components using some approach
     *e.g.*, operational steps, structured analysis, object-oriented
  3. Determine how the components interact
  4. Repeat 1-3 until some termination criteria is met
     *e.g.*, component can be implemented by 1 person in a few days

Abstraction

• Principle: Decomposition strategy should yield abstractions.
  – Abstraction: Something reduced to its essential characteristics
• Abstraction manages complexity by emphasizing essential characteristics and suppressing implementation details
• Allows postponement of certain design decisions
  – Representations
  – Algorithms
  – Architecture
  – Communications protocols
Abstraction (cont'd)

- Programming languages have continued to raise level of abstraction
  - Procedural abstraction
    - closed subroutines
  - Data abstraction
    - Abstract data types (ADTs)
  - Control abstraction
    - iterators, loops, atomic blocks, etc.

Modularity

- Principle: Decomposition strategy should promote modularity
- Module: A self-contained software component
- Module characteristics:
  - should possess well-specified abstract interfaces
  - should have high cohesion and low coupling
Modularity (cont'd)

- Modularity facilitates software quality factors:
  - Extensibility - well-defined, abstract interfaces
  - Reusability - low-coupling, high-cohesion
  - Compatibility – can create “bridging” interfaces
  - Portability - hide machine dependencies

Coupling

- Coupling is the interdependence of one module on another
- Modules should have low coupling
- Low coupling tends to limit the effect of changes & improves reuse
- Coupling created by:
  - Passing parameters
  - Passing unnecessary data structures
  - Communicating via shared signals
  - Sharing global data
  - Branching into, referencing or modifying each other’s data or statements
Cohesion

- Intuitively cohesion refers to the assoc. of elements within a module
- Elements should be strongly and genuinely related to each other, and should support one well-defined function or purpose
- High cohesion is desirable, but difficult to measure
- Examples of low cohesion
  - Temporal cohesion - e.g., do a bunch of unrelated things at shutdown
  - Procedural cohesion – e.g., unrelated functions done sequentially
  - Sequential cohesion – e.g., output of one function is input to another

More on Modularity

- Tactics for ensuring modular designs:
  - Language Support for Modular Units
  - Few Interfaces
  - Small Interfaces
  - Explicit Interfaces
  - Information Hiding
Modularity Tactics (cont'd)

- Language Support for Modular Units
  - Modules should correspond to syntactic units in the language used
- Few Interfaces
  - Every module should communicate with as few others as possible
    - e.g., Law of Demeter (www.ccs.neu.edu/home/lieber/LoD.html)
- Small Interfaces
  - If any two modules communicate at all, they should exchange as little information as possible

Modularity Tactics (cont'd)

- Explicit Interfaces
  - Whenever two modules A and B communicate, this must be obvious from the text of A or B or both
- Information Hiding
  - A module’s internal details should be invisible outside the module
### Information Hiding

- Principle: Modularization strategy should make change easier
- Information hiding is one criteria for defining modules
  - Details of design decisions that are subject to change should be hidden behind abstract interfaces
- Needs some extra language support
  - Enforce communication only through well-defined interfaces
- Desired outcome
  - Each component exposes as little information as possible.
  - If internal details change, client should be minimally affected
- Shorthand: Modules encapsulate secrets

### Information Hiding (cont'd)

- Some examples of information that can be hidden:
- Data representations
  - Underlying data structures
- Algorithms
  - Multiple variations of generic algorithm
- Input and Output Formats
  - Machine dependencies, byte-ordering, character codes
- Policy/mechanism distinctions
  - Separating when vs. how: access control, caching
- Lower-level module interfaces
  - Ordering of low-level operations
The Open/Closed Principle

A satisfactory module decomposition technique should yield modules that are both open and closed:

- Open Module: is one still available for extension
  - Necessary because requirements and specifications change
- Closed Module: is one available for use by other modules, usually given a well-defined, stable description and packaged in a library
  - Basically, you don’t want to change a module’s source code once it’s released
  - Necessary because otherwise changes ripple through user code

The Open/Closed Principle (cont'd)

- Modularity is not enough to support this principle
- Object-oriented languages use mechanisms like interfaces, inheritance and dynamic binding to solve this problem
Hierarchy

- Principle: reduce inter-module dependencies by restricting the topology of their relationships
- A relation defines a hierarchy if it partitions units into levels
  - Level 0 is the set of all units that use no other units
  - Level i is the set of all units that use at least one unit at level < i and no unit at level ≥ i
- Hierarchical structure is ubiquitous in design
  - Facilitates independent development
  - Isolates ramifications of change
  - Allows rapid prototyping

Hierarchy (cont'd)

- Some relations that define hierarchies:
  - Uses
  - Is-A
  - Has-A
- The first is general to all design methods, the latter two are more particular to object-oriented design and programming
The Uses Relation

- X uses Y if the correct functioning of X depends on the availability of a correct implementation of Y
- Uses is not the same as invokes:
  - Some invocations are not uses: one-way messages
  - Some uses don't involve invocation: external data stores

Uses Relation (cont'd)

- Allow X to use Y when:
- X is simpler because it uses Y
  - Standard C library routines
- Y is not substantially more complex because it is not allowed to use X
  - Hierarchies should be semantically meaningful
- There is a useful subset containing Y and not X
  - Allows sharing and reuse of Y
- There is no conceivably useful subset containing X but not Y
  - Y is necessary for X to function
The Uses Relation, (cont'd)

- A uses relation does not necessarily yield a hierarchy (cycles)
- How should cycles be handled?
  - Group X and Y as a single entity in the uses relation
- A hierarchy in the uses relation is essential for designing non-trivial reusable software systems

The Is-A and Has-A Relations

- Associated with object-oriented design and programming languages that possess inheritance and classes
- Is-A or Descendant relationship
  - class X possesses an Is-A relationship with class Y if instances of class X are specializations of class Y
- Has-A or Containment relationship
  - class X possesses a Has-A relationship with class Y if instances of class X contain one or more instance(s) of class Y
Separating Policy and Mechanism

- Principle: Separate what from how
- Policies (what) are implemented by mechanisms (how)
- Multiple policies can be implemented by single mechanism
  - Access control or CPU scheduling
- Same policy can be implemented by multiple mechanisms
  - FIFO sequencing can be implemented using a queue based on an array, or a linked list, or . . .

A General Design Process

- Given a requirements specification, design involves an iterative decision making process:
  - List the difficult decisions and decisions likely to change
  - Design a module specification to hide each such decision
  - Make decisions that apply to whole program family first
  - Modularize most likely changes first
  - Then modularize remaining difficult decisions and decisions likely to change
  - Design the uses hierarchy as you do this (include reuse decisions)
A General Design Process (cont'd)

- General steps (cont'd)
  - Treat each higher-level module as a specification and apply above process to each
  - Continue refining until all design decisions are:
    - hidden in a module
    - contain easily comprehensible components
    - provide individual, independent, low-level implementation assignments

Design Rules of Thumb


- What comes before how
- Define the service to be performed at every level of abstraction before deciding which structures should be used to realize the services
- Separate orthogonal concerns
  - Don’t connect independent things
Design Rules of Thumb (cont'd)

• Design external functionality before internal functionality
  – First consider the solution as a black-box and decide how it should interact with its environment
  – Then decide how the black-box can be internally organized. Likely it consists of smaller black-boxes that can be refined in a similar fashion

• Keep it simple
  – Fancy designs tend to be buggier than simple ones; they are harder to implement, harder to verify, and often less efficient in practice
  – Problems that appear complex are often just simple problems huddled together
  – Designer’s job is to identify the simpler problems, separate them, and then solve them individually

Design Rules of Thumb (cont'd)

• Design for extensibility
  – A good design is "open-ended," i.e., easily extendible
  – A good design often solves a class of problems rather than a single instance. But don’t go crazy!
    • Do not introduce what is immaterial
    • Do not restrict what is irrelevant
  – Use rapid prototyping when applicable
    • Before implementing a design, build a high-level prototype and verify that the design criteria are met
Design Rules of Thumb (cont'd)

• Details should depend upon abstractions. Abstractions should not depend upon details
• Where possible, use proven patterns to solve design problems
  – Will talk about design patterns in future lectures
• When crossing between two different abstractions, build an interface layer that separates the two
  – Don't pollute one side with the abstraction of the other

Design Rules of Thumb (cont'd)

• Software entities (classes, modules, etc) should be open for extension, but closed for modification
  – The Open/Closed principle -- Bertrand Meyer
• Subclasses must be usable through the superclass interface without the need for the user to know the difference
  – The Liskov Substitution Principle
Design Rules of Thumb (cont'd)

- Make it work correctly, then make it work fast
- Implement the design, measure its performance, and only then, if necessary, optimize it
- Maintain consistency between optimized versions
  - check that the final optimized implementation is equivalent to the high-level design that was verified