Computer Science 240

Principles of Software Design

Goals of Software Design

- Create systems that
  - Work
  - Are easy to understand, debug, and maintain
  - Are easy to extend and hold up well under changes
  - Have reusable components

Design is inherently iterative

- Design, implement, test, Design, implement, test, …
- Feedback loop from implementation back into design provides valuable knowledge
- Designing everything before beginning implementation doesn’t work
- Beginning implementation without doing any design also doesn’t work
- The appropriate balance is achieved by interleaving design and implementation activities in relatively short iterations

Abstraction

- Abstraction is one of the software designer's primary tools for coping with COMPLEXITY
- Programming languages and OSes provide abstractions that model the underlying machine
- Programs written solely in terms of these low-level abstractions are very difficult to understand
- Software designers must create higher-level, domain-specific abstractions, and write their software in terms of those
  - High-level abstractions implemented in terms of low-level abstractions

Abstraction

- Some abstractions correspond to “real world” concepts in the application domain
  - Examples: Bank, Customer, Account, Loan, Broker, …
- Other abstractions do not correspond to “real world” domain concepts, but are needed for internal implementation
  - Examples: HttpServer, Database, HashTable, …

Abstraction

- Each abstraction is represented as a class
- Each class has a carefully designed public interface that defines how the rest of the system interacts with it
- A client can invoke operations on an object without understanding how it works internally
- This is a powerful technique for reducing the cognitive burden of building complex systems

Naming

- A central part of abstraction is giving things names (or identifiers)
- Selecting good names for things is critical
- Class, method, and variable names should clearly convey their function or purpose
- Class and variable names are usually nouns
- Method names are usually verbs
  - Exceptions
    - Object properties (ID, Name, Parent, etc.)
    - Event handlers (MouseMoved, UserLoggedIn)

Cohesion / Single Responsibility

- Each abstraction should have a single responsibility
- Each class should represent one, well-defined concept
  - All operations on a class are highly related to the class' concept
- Each method should perform one, well-defined task
  - Unrelated or loosely related tasks should be in different methods
- Cohesive classes and methods are easy to name

Abstracting All the Way

- Some abstractions are simple enough to store directly using the language's built-in data types
  - Name => string
  - Pay Grade => int
  - Credit Card => string
- Often it is best to create classes for such simple abstractions for the following reasons:
  - Data validation
  - Related operations
  - Code readability

Decomposition

- In addition to Abstraction, Decomposition is the other fundamental technique for taming COMPLEXITY
- Large problems subdivided into smaller sub-problems
- Subdivision continues until leaf-level problems are simple enough to solve directly
- Solutions to sub-problems are recombined into solutions to larger problems

Decomposition

- Decomposition is strongly related to Abstraction
- The solution to each sub-problem is encapsulated in its own abstraction (class or subroutine)
- Solutions to larger problems are concise because they’re expressed in terms of sub-problem solutions, the details of which can be ignored
- The decomposition process helps us discover (or invent) the abstractions that we need

Decomposition

- Levels of decomposition
  - System
  - Subsystem
  - Packages
  - Classes
  - Routines
- Hypo- and Hyper-Decomposition
- When have we decomposed far enough?
  - Size metrics
  - Complexity metrics

Algorithm & Data Structure Selection

- No amount of decomposition or abstraction will hide a fundamentally flawed selection of algorithm or data structure.

Minimize Dependencies

- Dependencies
  - Class A CALLS Class B
  - Class A HAS MEMBER OF Class B
  - Class A INHERITS FROM Class B

Minimize Dependencies

- Minimizing the number of interactions between different classes has several benefits:
  - A class with few dependencies is easier to understand
  - A class with few dependencies is less prone to ripple effects
  - A class with few dependencies is easier to reuse

Minimize Dependencies

- When classes must interact, if possible they should do so through simple method calls
  - This kind of dependency is clear in the code and relatively easy to understand

Separation of Interface and Implementation

- Maintain a strict separation between a class' interface and its implementation
- This allows internal details to change without affecting clients
- `interface Stack` + `class StackImpl`

Information Hiding

- Many languages provide “public”, “private”, and “protected” access levels
- All internal implementation is “private” unless there’s a good reason to make it “protected” or “public”
- A class’ public interface should be as simple as possible

Information Hiding

- Don’t let internal details “leak out” of a class
  - `search` instead of `binarySearch`
  - `ClassRoll`instead of `StudentLinkedList`
- Some classes or methods are inherently tied to a particular implementation. For these it is OK to use an implementation-specific name
  - `HashTable`
  - `TreeSet`

Code Duplication

- Code duplication should be strenuously avoided
  - Identical or similar sections of code
- Disadvantages of duplication:
  - N copies to maintain
  - Bugs are duplicated N times
  - Makes program longer, decreasing maintainability
- Solutions
  - Factor common code into a separate method or class
  - Shared code might be placed in a common superclass