Computer Science 340

Software Design & Testing

Design Principles Review

Introduction

- Core principles of software design
  - Fundamental and relatively constant
  - When were our textbooks published? How could they still be relevant?
- Evaluation criteria for designs in this course
- Design evaluation is inherently subjective
- Providing specific evaluation criteria will help make grading as predictable as possible

Goals of Software Design

- Manage COMPLEXITY
- Make and keep software systems ORGANIZED as they evolve
- Combat design ENTROPY
- Create systems that
  - Work
  - Are easy to understand, debug, and maintain
  - Are easy to extend and hold up well under changes
  - Reusable

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

Design Principles

- Abstraction
- Naming
- Cohesion
- Orthogonality
- Decomposition
- Duplication Elimination

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

- 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

Abstraction

- Many domain-specific abstractions can be taken directly from the "domain model"
  - A **domain model** inproblem solving and software engineering is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain.
- Other abstractions do not appear in the domain model, but are needed for internal implementation purposes
  - Mid-level abstractions such as: NotificationManager, PersistenceManager, UserSession, UndoRedoManager, ThreadPool

Abstract All the Way / Avoid Primitive Obsession

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

When is a Primitive Not Right? Domains Don’t Match

- The set of valid values (the domain) doesn’t match.
  - The set of valid values is usually a subset of the set of values in a primitive type
  - Example:
    - The domain of an Age is probably any integer between 0 and 100 (maybe a 1000). That is different than the domain of an integer.
    - The domain of a Name is, at a minimum, a sequence of alphabetic letters. This is different than a String which is a sequence of any character.
- If you create a new class, the domain checks needed during construction and assignment can be isolated to the new class.

When is a Primitive Not Right? Operations Don’t Match

- There are operations you need that are not provided by the primitive type.
  - Example: Formatting a Name as: LastName “, “ SimpleName+
- There are operations you don’t need or want users to have access to but are provided by the primitive type.
  - Example: ~age is not needed
  - Example: -age, don’t want to negate and age

Three Forms of Abstraction

- Aggregation
- Classification
- Generlization/Specialization

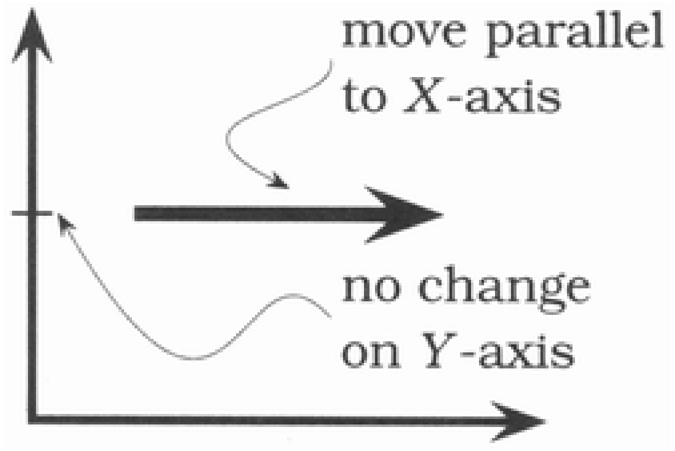
Naming

- 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 represents one, well-defined concept
  - All operations on a class are highly related to the class' concept
  - Cohesive classes are easy to name
- Each method performs one, well-defined task
  - Unrelated or loosely related tasks should be in different methods
  - Cohesive methods are easy to name
  - Using "And" or "Or" in a method name suggest a method with low cohesion

Orthogonality



In computing, two or more things are orthogonal if changes in one do not affect any of the others.

You're on a helicopter tour of the Grand Canyon when the pilot, who made the obvious mistake of eating fish for lunch , suddenly groans and faints. Fortunately, he left you hovering 100 feet above the ground. You rationalize that the collective pitch lever [2] controls overall lift, so lowering it slightly will start a gentle descent to the ground. However, when you try it, you discover that life isn't that simple. The helicopter's nose drops , and you start to spiral down to the left. Suddenly you discover that you're flying a system where every control input has secondary effects. Lower the left-hand lever and you need to add compensating backward movement to the right-hand stick and push the right pedal. But then each of these changes affects all of the other controls again. Suddenly you're juggling an unbelievably complex system, where every change impacts all the other inputs. Your workload is phenomenal: your hands and feet are constantly moving, trying to balance all the interacting forces.

[2] Helicopters have four basic controls. The *cyclic* is the stick you hold in your right hand. Move it, and the helicopter moves in the corresponding direction. Your left hand holds the *collective pitch lever.* Pull up on this and you increase the pitch on all the blades, generating lift. At the end of the pitch lever is the *throttle* . Finally you have two foot *pedals,* which vary the amount of tail rotor thrust and so help turn the helicopter.

Helicopter controls are decidedly not orthogonal.

Cohesion and Orthogonality

An orthogonal approach reduces the risks inherent in any development.

- Diseased sections of code are isolated. If a module is sick, it is less likely to spread the symptoms around the rest of the system. It is also easier to slice it out and transplant in something new and healthy .
- The resulting system is less fragile. Make small changes and fixes to a particular area, and any problems you generate will be restricted to that area.
- An orthogonal system will probably be better tested, because it will be easier to design and run tests on its components.
- You will not be as tightly tied to a particular vendor, product, or platform, because the interfaces to these third-party components will be isolated to smaller parts of the overall development.
- As Cohesion goes up, orthogonality goes up

Decomposition: Generating Aggregates

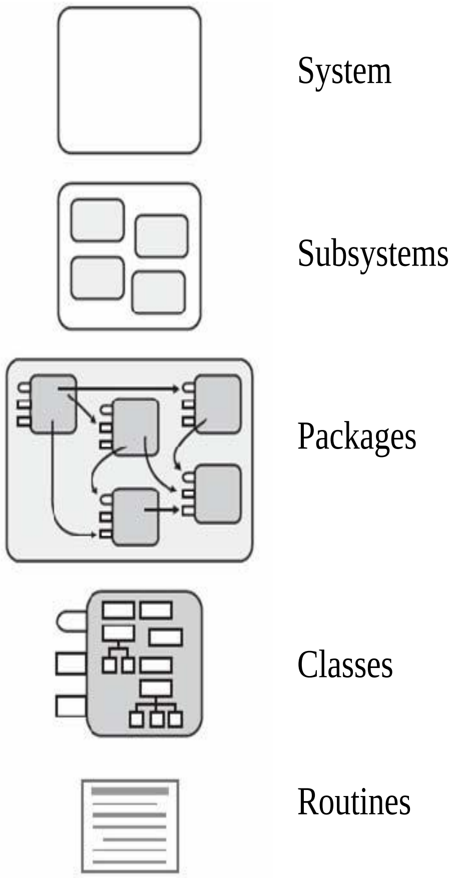
- In addition to Abstraction, Decomposition is the other fundamental technique for taming COMPLEXITY
- Large problems subdivided into smaller sub-problems
- Subdivision continues until leaf-level problems are simple enough to solve directly
- Solutions to sub-problems are recombined into solutions to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



Decomposition

- Hypo- and Hyper-Decomposition

- When have we decomposed far enough?
  - Size metrics
    - Rules of thumb
      - 500 Lines of Code in a method is probably to long
      - 20-50 lines of code?
  - Complexity metrics
    - Rules of thumb
      - 10 parameters for a method may be too many
      - A class with 50 methods may be too many

Algorithm & Data Structure Selection

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

- Remember the lessons of
  - CS 235, CS 236, and CS 312

Minimize Dependencies: Minimizing Coupling

- Dependencies
  - Class A instantiates class B
  - Class A invokes operations on class B
  - Class A accesses class B’s internal state
  - Class A inherits from class B
  - Class A has a method parameter of class B
  - Class A and class B both access the same global data structure or file
  - Class A makes assumptions about class B
  - “And finally, I cannot tell you all the ways whereby classes may depend on each other; for there are divers ways and means, even so many that I cannot number them.” (Mosiah 4:29, with modifications)

Minimize Dependencies

- Minimizing the number of communication channels and 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
- A class' public interface should be as small (or "thin") as possible

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`
- Program to interfaces rather than classes when possible
  - Declare variables/parameters as type List instead of ArrayList, as type Set instead of HashSet, etc.

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”
- The same goes for the “protected” subclass interface
  - Don’ t just blindly expose all internal details to subclasses. Expose only what they must have access to.

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. In these cases it is OK to use an implementation-specific name
  - `HashTable`
  - `TreeSet`

Law of Demeter

- Principle of least knowledge
  - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
  - Each unit should only talk to its friends; don't talk to strangers.
  - Only talk to your immediate friends.

Law of Demeter

- More formally, the Law of Demeter for functions requires that a method m of an object O may only invoke the methods of the following kinds of objects:
  - O itself
  - m's parameters
  - Any objects created/instantiated within m
  - O's direct component objects
  - A global variable, accessible by O, in the scope of m
- In particular, an object should avoid invoking methods of a member object returned by another method. For many modern object oriented languages that use a dot as field identifier, the law can be stated simply as "use only one dot". That is, the code a.b.Method() breaks the law where a.Method() does not. As a simple example, when one wants a dog to walk, one would not command the dog's legs to walk directly; instead one commands the dog which then commands its own legs.

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
    - When you think you’ve fixed a bug, its twins still exist
  - Makes program longer, decreasing maintainability
- Solutions
  - Factor common code into a separate method or class
  - Shared code might be placed in a common superclass

Don’t Repeat Yourself

- Single Responsibility Principle
  - One and only one reason to change
  - Consider Java Date class
    - Used to represent time and date
    - Now there is a Date and GregorianCalendar class
- Isolated Change Principle
  - Ideally the implementation of a responsibility will be isolated to one class or package of classes
  - Not always achievable
  - Consider the opposite: Shotgun Surgery
    - One change alters many classes

Don’t Repeat Yourself

- **Imposed duplication.** Developers feel they have no choice, the environment seems to require duplication.
  - Documentation
  - Code comments
  - Duplication imposed by the language
    - Headers
    - Interfaces

Don’t Repeat Yourself

- **Inadvertent duplication.** Developers don't realize that they are duplicating information.

```
class Line {

public:

    Point  start;
    Point  end;

double  length;

};
```

```
class  Line {

public:

    Point  start;
    Point  end;

double length() {

        return start.distanceTo(end);
}

};
```

Don’t Repeat Yourself

- **Impatient duplication.** Developers get lazy and duplicate because it seems easier.
- **Interdeveloper duplication.** Multiple people on a team (or on different teams) duplicate a piece of information.

Error Reporting

Any operation that can fail should report errors

- Throw exceptions
  - void run() throws BadThingHappenedException;
- Return result objects
  - boolean run(); -- Not very informative
  - Define Result and ValueResult classes
    - Result run();
    - ValueResult<OutputType> run();
  - Documentation should enumerate the possible error modes
    - Enumerated types can be useful here

Example

```
class PhoneBookEntry {
        private String _name;
        private String _address;
        private String _phoneNumber;
}
```

```
class PhoneBookEntry {
        private Name        _name;
        private Address     _address;
        private PhoneNumber _phoneNumber;
}
```

Often, it is best to wrap data in a domain-specific abstraction

Example

```
// No abstraction
List<PhoneBookEntry> phoneBook;
```

```
// Better to wrap it in a class that also contains operations
class PhoneBook {

        private List<PhoneBookEntry> _entries;

}
```

Example

```
// POOR ABSTRACTION
class PhoneBook {

    private List<PhoneBookEntry> _entries;

    // LAZY way to support Add, Update, Delete, Iterate, Search
    List<PhoneBookEntry> getEntries() {
        return _entries;
    }

    // Domain-specific Algorithms
    ...
}
```

Exposes internal implementation

Provides no data integrity enforcement

Lacks operations needed by the program

```
class PhoneBook {

        private List<PhoneBookEntry> _entries;


        public Result addEntry(PhoneBookEntry entry) { ... }

        public Result updateEntry(PhoneBookEntry before, PhoneBookEntry after) { ... }

        public Result deleteEntry(PhoneBookEntry entry) { ... }

        public Iterator<PhoneBookEntry> findAll() { ... }

        public PhoneBookEntry findByPhoneNumber(PhoneNumber value) { ... }

        public Iterator<PhoneBookEntry> findByName(Name value) { ... }

        public Iterator<PhoneBookEntry> findByAddress(Address value) { ... }


        // Domain-specific Algorithms

        …
}
```

```
// Read-Only Search Results
class PhoneBook {

        private List<PhoneBookEntry> _entries;

        …

        //***
        public Iterator<PhoneBookEntry> findAll() {

                return _entries.iterator();

        }

        //*** OR

        public Iterator<PhoneBookEntry> findAll() {

                return java.util.Collections.unmodifiableList(_entries).iterator();

        }

        //***

        …

}
```

```
// Indexing

class PhoneBook {

        private List<PhoneBookEntry> _entries;

        private Map<PhoneNumber, PhoneBookEntry> _indexByPhoneNumber;
        private Map<Name, List<PhoneBookEntry>> _indexByName;
        private Map<Address, List<PhoneBookEntry>> _indexByAddress;


        public Result addEntry(PhoneBookEntry entry) { ... }
        public Result updateEntry(PhoneBookEntry before, PhoneBookEntry after) { ... }
        public Result deleteEntry(PhoneBookEntry entry) { ... }
        public Iterator<PhoneBookEntry> findAll() { ... }
        public PhoneBookEntry findByPhoneNumber(PhoneNumber value) { ... }
        public Iterator<PhoneBookEntry> findByName(Name value) { ... }
        public Iterator<PhoneBookEntry> findByAddress(Address value) { ... }


        // Domain-specific Algorithms

        ...

}
```

```
// Enable/Disable
class PhoneBook {

        private List<PhoneBookEntry> _entries;
        private Map<PhoneNumber, PhoneBookEntry> _indexByPhoneNumber;
        private Map<Name, List<PhoneBookEntry>> _indexByName;
        private Map<Address, List<PhoneBookEntry>> _indexByAddress;

        public Result canAddEntry(PhoneBookEntry entry) { ... }

        public Result addEntry(PhoneBookEntry entry) {
                Result result = canAddEntry(entry);
                if (result.getStatus() == false) {
                        return result;
                }
                else {
                        ...
                }
        }

        public Result canUpdateEntry(PhoneBookEntry before, PhoneBookEntry after) { ... }

        public Result updateEntry(PhoneBookEntry before, PhoneBookEntry after){
                Result result = canUpdateEntry(before, after);
                if (result.getStatus() == false) {
                        return result;
                }
                else {
                        ...
                }
        }
```

can-methods used to enforce constraints

Enforcing Constraints

- isValid and can-methods
- Internal class validation
  - Check whenever data is going to be modified
- Manager classes
  - Some constraints are higher level
  - Constraints on the collection
  - Support with a manager class

Summary

- Abstraction
  - Naming
  - Cohesion
  - Abstracting All the Way
- Decomposition
  - Levels of Design (System, Subsystem, Package, Class, Routine)
  - Hypo- and Hyper- Decomposition
  - Size and Length Metrics
  - Complexity Metrics
- Algorithm & Data Structure Selection
- Minimize Dependencies (or, Low Coupling)
  - Separation of Interface and Implementation
  - Information Hiding
- Avoid Code Duplication
- Error Reporting