Design Patterns and Anti-Patterns
Overview

- Recently covered topics
  - Object-oriented programming concepts
  - Object-oriented modeling tools (UML)
  - SOLID design principles
  - Design patterns
  - Anti-patterns
Overview

• Recently covered topics
  • Object-oriented programming concepts
  • Object-oriented modeling tools (UML)
  • SOLID design principles
  • Design patterns
  • Anti-patterns

In terms of coding, the last three differentiate software engineers from regular programmers.
“An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive” — D. Budgen

Expressed in other words…

*Bad* development practices frequently seen in attempts to solve a problem (design smells)

- Listed for processes in business and software engineering. Some SE examples:
  - Big ball of mud
  - Gold plating
  - Input kludge
  - Magic pushbutton
  - Race hazard
  - Basebean
  - God object
  - Object orgy
  - Poltergeists
  - Sequential coupling
  - Magic numbers
  - Improvability factor
  - Premature optimization
  - Data clump
  - and many more…

Read about all of them and get very familiar with at least 3
Some Key Underlying Quality Factors

- Cohesion
  - Measure of relatedness
- Coupling
  - Measure of interdependence
- Connascent components
  - Change in one requires modification in the other
- Cyclomatic complexity
  - The number of linearly independent paths within a code
- Halstead complexity
  - Measurable properties of software (length, volume, difficulty, effort, time required to program, number of delivered bugs)
- Maintainability index
  - Measurement of code entropy/degraded integrity
Cohesion

• **Definition**
  - Degree to which elements of a single entity are *related*
  - Two methods are related *if* they access same the class-level variable, or if one calls the other

• **Advantages of high cohesion**
  - Increased module reusability
  - Increased maintainability
  - Lower module complexity
A Measure of Cohesion

• Lack of Cohesion of Methods (LCOM4)
  • Measures the number of connected components, i.e. sets of related methods, in a class
  • LCOM4 = 1 (“good” cohesive class)
  • LCOM4 >= 2 (“bad” class should be split)
Coupling

• **Definition**
  • Degree of interdependence between software modules
  • Measure of how closely connected 2 routines/modules are

• **Low coupled system**
  • Components have little knowledge of the definition of other separate components

• **Advantages**
  • Low coupling is associated with good design
  • Combined with high cohesion leads to high readability and maintainability
Coupling refers to the interdependencies between modules, while cohesion describes how related the functions within a single module are.
Some design smells and what to do about them
Related to Cohesion

- **Data clump**
  - Sign that a class is evolving towards multiple responsibilities

- **Low cohesion**
  - high LCOM4 value means class has multiple personalities

*violations of the SRP*

- **Solution**
  - Should remove and encapsulate additional responsibilities in separate class — extract class refactoring
Case statement smell

• **Situation**
  - Code shows multiple switch-cases based on type of argument decided at runtime

*Violation of the OCP*

• **Solutions**
  - Apply the **abstract factory, factory method** or **strategy** patterns
Example: Problem 1

- Problems
  - Report is dependent on concrete subclasses
  - output() tries to be “polymorphic”
  - Adding more formats can lead to lots of changes in client class
    Report (shotgun surgery)
Example: Solution 1.1 - Factory method

- **Benefits**
  - Client class Report does not depend on concrete classes
  - New formats require minimal changes to client class
  - Program to the interface not the implementation
Related to Coupling

- **Situation**
  - Tight dependencies between client and service classes

  *violations of the OCP*

- **Solutions**
  - E.g. Dependency injection pattern
Example: Problem 2

- **Problems**
  - Report is directly dependent on HTMLformatter
    - Hard to change one without affecting the other
    - E.g. if a new formatter wants to be used
  - Inflexible design can affect maintainability down the line
Example: Solution 2.1 - Dependency injection

```java
method main() {
    Formatter formatter = new HTMLformatter()
    Report report = new Report(formatter);
    report.output();
}
```

**Benefits**

- Report is now loosely coupled with HTMLformatter
- Adding new formatters don’t affect Report

**Who should own the interface Formatter, the client side or the service side?**
Dependency Injection

• Some potential issues
  • Issue 1: Injecting a dependency may expose different APIs for the concrete classes

• Solution
  • Adapter pattern - standardizes the API
  • Facade pattern

• Issue 2: What if do-nothing behavior is needed?
• Solution
  • Null object pattern - add a concrete class with same API but no action
Refused bequest

- A special case implemented through inheritance
  - E.g. Square is a special case of Rectangle
    - Rectangle method: makeTwiceAsWideAsHigh()
    - Inherited but cannot be used (should raise exception or handled in some way)

Violation of the LSP

- Solutions
  - Use composition and delegation rather than inheritance
Other Smells

• Inappropriate intimacy:
  • One class uses the internal fields and methods of another class.
  • Too much of it is called feature envy

• Mock train wreck
  • Difficulty of mocking a deeply nested model structure
  • Mocking is the creation of mock objects which can be used to mimic the behavior of real objects, often because it is hard to test with the real objects
  • A trainwreck is multiple levels of method calls (called a chain), which each return objects upon which new methods can be called

• Solutions
  • Dependency injection pattern
  • Delegation
  • Visitor pattern / Iterator pattern
  • Observer pattern