Responsibility Driven Design

Mikael Svahnberg*

2023-09-13

Contents

*Mikael.Svahnberg@bth.se

1 Introduction

- Barnes & Kölling Chapter 8, Designing Classes
- Barnes & Kölling Chapter 9, Well-Behaved Objects
- Book tip: Robert C. Martin, *Clean Code*, Pearson Education, 2009.
- Book tip: Steve McConnell, *Code Complete*, Microsoft Press, 2004.
- Design Principle: *Low Coupling*
- Design Principle: *Encapsulation*
- Design Principle: *Localising Change (High Cohesion)*

2 Tight Coupling

Examples

- We pass around an object that only contains data but no methods
- We use `switch/case` statements to figure out which method to call in another object
- We use similar `switch/case` statements in many places
- We have long call chains:

```
myCustomers->findCustomer(theCustomerName)
->getTickets()->filterByTime("today")->first()->getDetails();
```

- These are all examples of hard coded dependencies
- Require knowledge about
  - Which objects there are
  - The type of each object
  - Which method to call for every type of object and situation
- Broken Design Principle: *Low Coupling*
  - More couplings than is necessary
  - Tight couplings
- Broken Design Principle: *High Cohesion*
  - We need to know and manage the structure of *other* objects to do our job
  - We need to know the design to do our job

3 Code Duplication

- Similar code in many methods
  - Often "branching logic" (`if/else` or `switch/case`)
  - Input/Output
  - Deriving a calculated value from existing data
- $\sum$ Difficult to extend and maintain the system
  - Many places need edit for the same change
  - Many places contain the same bug
  - More code to read and understand
  - Methods become longer and difficult to overview
- Code duplication may imply tighter coupling to other classes
- Code duplication may imply tighter coupling to an *idea* (same branching logic in many places)
- Code duplication can either directly mean tighter coupling to other classes
- or, code duplication means tighter coupling to an *idea* (same switch-case in many places)
- Broken Design Principle: *Encapsulation*
  - The same logic is exposed in many places
- Broken Design Principle: *High Cohesion*
  - Methods contain several responsibilities
  - The responsibility they *should* have
  - The responsibility to know how the rest of the system works

4 Responsibility-Driven Design

- There should only be one reason for modifying a class
  - Also known as *Single Responsibility* principle.
  - Results in *High Cohesion*
  - Implies that *Each class has access to all the data it needs to fulfill its responsibilities*
  - Implies that *Each class has all the methods required to fulfill its responsibilities*
- Each class should be responsible for handling its own data
  - Also known as *Information Expert*
  - An object represents a specific real world entity by expressing all its data attributes

  - An object should not need the help of other objects to operate on those attributes
  - Implies that *Each class has all the methods required to fulfill its responsibilities*
- Also works in the small: Each *method* should have a single responsibility

5 Doing or Knowing

- Responsibilities can involve *doing* or *knowing*
  - *Doing*, e.g.
    * create objects
    * calculate or process some data
    * coordinate the actions of other objects
  - *Knowing*, e.g.
    * know some data
    * know a workflow
    * know which objects are needed for a workflow
    * know how to process some data

6 Localise Change → Encapsulation

- As few places as possible required to fix one issue
- As "close together" as possible
  - within one method
  - within one class
  - within one package
  - within one group of classes that collaborate on similar issues
- We *encapsulate* the change into a single place
- We may have to introduce a new method or a new class

7 Indirection/Implicit Coupling

- Group similar concepts into
  - same class (different values of attributes)
  - same data structure (using inheritance if necessary)
  - a single switch/case (encapsulate logic into a single method)
  - same design pattern (e.g. *Stategy*)
- $\sum$ We call some other class/object/method to perform the task.
  - Examples of *indirection*
  - Ideally we let the compiler decide (e.g. with inheritance and polymorphism)

8 Cohesion

- A unit of code should always be responsible for one and only one task.
- Applies to methods as well as classes
- Methods
  - "not my responsibility"? → call another method.
  - Similar to something done in another method? → create a new method and call it.
  - Many steps in methods' responsibility? → create new methods and call them.
- Classes
  - One single well-defined entity in the problem domain
  - new concept → new class
  - No longer "just an attribute" → new class
  - Several attributes for specific part of the responsibility → new class
  - new responsibility → new class
  - Different ways to solve the responsibility → refactor e.g. with Strategy Design Pattern
- High Cohesion helps
  - Readability : Fewer things to keep in mind at the same time
  - Reuse : Easier to extract independent pieces of code

9 Refactoring

- Refactoring changes the structure of the code without changing the functionality
- Automated unit testing ensures that functionality remains the same

Refactoring Checklist

☐ You leave the code cleaner than you found it.

☐ There is no new functionality added together with a refactoring

☐ All existing tests still pass after refactoring

Clean Code

☐ Obvious to other programmers

- Reasonable size of each component/class/method
- Good names for classes attributes, methods, variables

☐ Does not contain duplicated code

☐ Does not contain unnecessary classes

☐ Passes all tests

10 Code Smells

- Originally by Martin Fowler.
- Can be summarised as *"You know all the bad things your OO design teacher told you not to do? Well, don't!"*

Examples

Bloaters Code that grows too long → low cohesion

- long methods
- long parameter lists
- large classes
- using primitives instead of small objects

Object Orientation Abusers incorrect use of object oriented programming principles

- Classes that do the same things but with different method names
- Classes that only partially implement the interface from a superclass
- Long and elaborate switch statements rather than relying on polymorphism

Change Preventers Responsibilities are scattered through code so you need to change several places at once

- When many methods need to be edited for a single change (e.g. adding a new product type)
- Parallel inheritance hierarchies: Adding a class in one hierarchy means you also need to add a class in another hierarchy

Dispensables Pointless code or text that does not contribute anything

- Too many comments
- Duplicate code
- Dead Code
- Classes that no longer do anything meaningful
- Adding classes or inheritance hierarchies for future needs

Couplers Things that tie classes too closely together

- Using data in other classes (more than your own data)
- Message chains `myFancyObject->getFrobnicator()->createFluxCapacitor()->initiate()`

11 When to Refactor

Rule of Three

1. First time, just get it done
2. Second time you do something similar, recognise that it is similar but do it anyway
3. Third time – refactor!

Adding a Feature

- Refactor while reading existing code – as a part of understanding it

When fixing a bug

- Clean up the code while looking for the bug

Code Reviews

- Regular activity with the purpose of cleaning up the code

A Little Bit All The Time

- Make this part of your normal velocity
- Easier than trying to motivate code reviews to the powers that be

12 Refactoring Techniques

(A selection that has a design impact: There are many more techniques for how to write clearer code *within* methods)

- Break out code into new methods to simplify the code
- Move methods and attributes to the class that should be responsible for them
- Remove classes that do not have any responsibilities
- Hide delegates to avoid method chains. If you are just "object hopping" to reach the right object, then you know too much about the design.
- Use wrapper classes to add functionality to libraries.
- Use getters and setters to access data
- Keep code from different layers/components separate. Duplicate data that should pass between components.
- Introduce classes to maintain collections (xxxManager, xxxContainer, . . .)
- Use Design Patterns instead of if-then-else chains.
- Create methods for complex if-then-else statements: `if condition() then trueCondition() else falseCondition()`

- Always return a meaningful object (e.g. a `null` Object)
- Rename classes/methods/parameters/attributes/variables to meaningful and readable names
- Separate queries from modifiers → avoid side effects in code
- Parameterise method (from `frobnicateA(), frobnicateB(), \dots` to `frobnicate(type)`
- Replace complex constructor with a factory method.
- Apply Design Patterns
- Apply Fundamental Object Oriented Design Principles

13 Testing

- *Syntax Errors* , the compiler tells you what you have written wrong
- *Logical Errors* , the compiler does not know this is wrong

Logical Errors can be *found* by:

- Manual Code Inspections
- Ad-hoc testing
- (Automated) Unit Testing

$\sum$ the test framework tells you that your code is doing the wrong thing.
Logical Errors are *understood* by:

- Manual Code Inspection
- Debugging

The risk of introducing Logical Errors is lowered by:

- Writing for maintainability
- Object Oriented Design Principles
- Up-front design
- Unit testing

14 Testing during Coding

Positive Testing Does the program work as expected?

Negative Testing Does the program fail when it is supposed to?

When coding, we use both positive and negative testing

When debugging, we follow a specific test and see where we go wrong

Keep the test to make sure we never make that particular mistake again!

15 Interactive Testing

- Debugger
  - *Inspect* values of attributes and objects
  - *step through* code as it is executing
  - The debugger is *extremely* useful but rarely used. Become friends with it!
- Console Output
  - `System.out.println()`
  - `myDebugPrint(debugLevel, message)`
  - a logging framework
- Manual code inspection together with debugger or console output
- What is the *initial state* of a variable?
- Where is it changed? What is the *expected value* after it should have been modified?
- Are object references copied and objects cloned when they should be?

16 Unit Testing Principles

- What is the *unit*? `component`, `package`, `class`, `method`
- What do we test? All values of all parameters? All branches?
- Typically:
  - a few "normal" cases that should pass
  - a few cases that should fail
  - edges and extremes (too big values, too small values, null, zero, ...)

Key principles:

- *Repeatable* and *Independent* tests
- *Organised* to reflect code structure, `test suites`
- *Portable* and *Reusable*: reuse code, reuse test.
- *Helpful* information when a test fails
- Let developers focus on *content* rather than "test framework overhead"
- *Fast*, make tests an extension of the compiler
- *Meaningful*: The test should evaluate your program, not whether the compiler is able to store a value.

```
// Bad Test
@test
public void TestingTheCompiler() {
  String hw = "hello world";
  Message aMess = new Message(hw);
  assertEquals(hw, aMess.getMessage());
}
```

17 Unit Testing Frameworks

- Java: junit
- C++: GoogleTest
- C++, Java, and Other languages: Microsoft Unit Test
- cppunit, cxxunit, mocha, ...
- Often integrated into your development environment

18 Unit Testing fundamentals

Each test framework has their own flavour, but a few things are common:

Fixture setup of objects to be used in the same way in all tests

- `setUp()` is run before every test
- `tearDown()` is run after every test

Test Suite a collection of tests that should be run together

Test Runner executes the test cases. has a `main()` method

Assert Used inside a test for the actual evaluation

19 Summary

- Well-behaved classes and methods:
  - are *commented*
  - are *testable*
  - are *debugable*
  - have a *single area of responsibility*
  - *encapsulates* behaviour in easy-to-find and localised places.

20 Next Lecture: Design Patterns: Observer and Decorator

- Freeman & Robson, Chapter 2: The Observer Pattern
- Freeman & Robson, Chapter 3: The Decorator Pattern
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*
- Design Principle: *Low Coupling*
- Design Principle: *Composition over Inheritance*
- Design Principle: *Open-Closed Principle*
- Design Pattern: *Observer*
- Design Pattern: *Decorator*