Introduction to Test-Driven Development

Christopher Bartling

Why Test-Driven Development

- Assessing code quality
  - Tests
- Code design
  - Tests communicate soundness of design decisions
- Testing as a design activity
  - Clarify our ideas about the intent of our code
- Executable documentation

Feedback

- We apply feedback loops to every level of our development.
- Unit testing is a feedback loop that occurs within seconds.
  - Quick execution of a unit test provides us with information.
- Constant testing will catch regression errors.
- Test-driven development gives us feedback on...
  - Quality of the implementation (does it work)
  - Quality of the code design (is it well structured)

Test-driven development mantra

- Red: Write a test that doesn't work or perhaps doesn't even compile
- Green: Make the test pass--don't worry about code duplication at this point
- Refactor: Eliminate code duplication and make source code more communicative, all the while making sure all tests continue to pass successfully

Code smells

- Any characteristic in the source code that possibly indicates a deeper problem
- Structures in the code that indicate violation of fundamental design principles and negatively impact design quality
- Also known as *anti-patterns*

Code smell examples: Application level

- Mysterious Name: Naming of functions, modules, variables or classes that does not communicate their intent
- Duplicated code: identical or very similar code that exists in more than one location (*aka* copy pasta)
- Contrived complexity: forced usage of overcomplicated design patterns where simpler design patterns would suffice
- Uncontrolled side effects: side effects of coding that commonly cause runtime exceptions, with unit tests unable to capture the exact cause of the problem

Code smell examples: Class level

- Large class: Class has grown too large and has too many responsibilities (*aka* God object)
- God objects: a class that has lots of responsibilities and is low cohesive.
- Feature envy: a class that uses methods of another class excessively and has no responsibility of its own
- Inappropriate intimacy: a class that has dependencies on implementation details of another class
- Refused bequest: a class that overrides a method of a base

Code smell examples: Class level

- Excessive use of literals: Literals should be coded as named constants, to improve readability and to avoid programming errors
- Cyclomatic complexity: Too many branches or loops, indicating that a function needs to be broken up into smaller functions, or that it has potential for simplification/refactoring

Code smell examples: Class level

- Downcasting: Typecasting to a specific derived type which breaks the abstraction model; the abstraction may have to be refactored or eliminated
- Orphan variable or constant class: a class that typically has a collection of constants which belong elsewhere where those constants should be owned by one of the other member classes
- Data clump: Occurs when a group of variables are passed around together in various parts of the program. In general, this suggests that it would be more appropriate to formally group the different variables together into a single object, and pass

Code smell examples: Method level

- Too many parameters: a long list of parameters is hard to read, and makes calling and testing the function complicated. It may indicate that the purpose of the function is ill-conceived and that the code should be refactored so responsibility is assigned in a more clean-cut way
- Long method: a method, function, or procedure that has grown too large
- Excessively long identifiers: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture

Code smell examples: Method level

- Excessively short identifiers: the name of a variable should reflect its function unless the function is obvious
- Excessive return of data: a function or method that returns more than what each of its callers needs
- Excessively long line of code (or God Line): A line of code which is too long, making the code difficult to read, understand, debug, refactor, or even identify possibilities of software reuse

SOLID principles

- SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.
- The principles are a subset of many principles promoted by American software engineer and instructor Robert C. Martin (*aka* Uncle Bob), first introduced in his 2000 paper Design Principles and Design Patterns.
- Single-responsibility principle
- Open–closed principle

Single Responsibility Principle (SRP)

- There should never be more than one reason for a class to change.
- In other words, every class should have only one responsibility.

Open–Closed Principle (OCP)

- Software entities should be open for extension, but closed for modification.

Liskov Substitution Principle (LSP)

- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
- Design by contract.

Interface Segregation Principle (ISP)

- Many client-specific interfaces are better than one general-purpose interface.

Dependency Inversion Principle (DIP)

- Depend upon abstractions, [not] concretions.

GRASP

- General Responsibility Assignment Software Patterns (or Principles), abbreviated **GRASP**.
- Set of "nine fundamental principles in object design and responsibility assignment"
- First published by Craig Larman in his 1997 book **Applying UML and Patterns**.

GoF design patterns

Title

Further reading

- Test-Driven Development: By Example by Kent Beck
- Working Effectively with Legacy Code by Michael C. Feathers
- Growing Object-Oriented Software, Guided by Tests by Steve Freeman and Nat Pryce