7. Refactoring

Java Test Driven Development with JUnit

Module Topics

Defining Refactoring

Refactoring is a controlled technique
for improving the design of an existing
code base. Its essence is applying a series
of small behavior-preserving transformations,
each of which "too small to be worth doing".

However the cumulative effect of each of
these transformations is quite significant.
By doing them in small steps you reduce the
risk of introducing errors. You also avoid
having the system broken while you are
carrying out the restructuring - which allows
you to gradually refactor a system over an
extended period of time.

Martin Fowler

Describing Refactoring

- Refactoring:
  - *Changes the structure of existing code*
  - *Does **not** changes the the functionality of the code*
  - *Uses TDD to ensure code changes do not cause bugs*

- Refactoring is usually done:
  - *After the functionality of the code changes*
  - *As part of a design change*
  - *After a bug has been fixed*
  - *After a code review*

The Importance of Well Structured Code

- Code that is well designed and well structured:
  - *Is easier for programmers to understand*
  - *Is easier to modify*
  - *Is easier to maintain*
  - *Is more elegant and efficient*
- When changes to code take place:
  - *The structure of the code degrades*
  - *The degradation to the code is cumulative*
  - *Eventually the code devolves into an unstructured mess*

Defining Well Structured Code

- Designed using Object Oriented best practices e.g. SOLID:
  - *S: Single Responsibility Principle*
  - *O: Open/Close Principle*
  - *L: Liskov Substitution Principle*
  - *I: Interface Segregation Principle*
  - *D: Dependency Inversion Principle*
- Written using the principles of Clean Code
  - *Often called Software Craftsmanship*
  - *For example: Command Query Segregation*

Refactoring and Code Smells

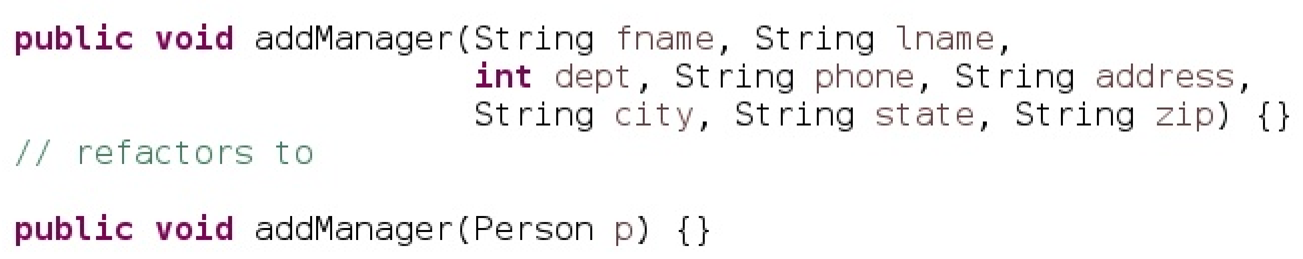
Code Smell

- A code smell is:
  - *"... a surface indication that usually corresponds to a deeper problem in the system" according to Fowler*
  - *Where the code does not support good design practices*
  - *An indication a refactoring needs to be done*

- This is closely related to the idea of an anti-pattern
  - *Anti-patterns are recurring patterns of how developers go from a problem statement to a bad solution*
  - *Anti-patterns can occur at different levels: project management, architecture, etc.*
  - *Code smells are sometimes referred to as code anti-patterns*
  - *Refactorings also exist for anti-patterns*

Code Smell Example - Long Parameter List

- Clean Code principle:
  - *Methods should have arguments of length 0 or 1, with 2 allowed in exceptional circumstances*
- Code smell – Long Parameter List
  - *One of the reasons that this smell can occur is that we have not de-signed our objects correctly*
  - *A refactoring is to encapsulate the parameters in a defined class*

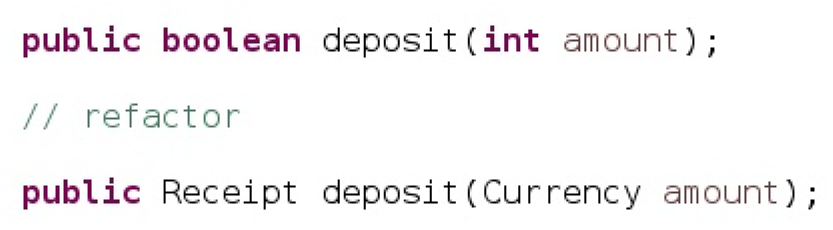


```
public void addManager(String fname, String lname,
                       int dept, String phone, String address,
                       String city, String state, String zip) {}
// refactors to

public void addManager(Person p) {}
```

Code Smell Example - Primitive Obsession

- OOP principle:
  - *Data types should reflect domain types*
- Code smell – Primitive Obsession
  - *This smell occurs because built in data primitives are used instead of meaningful types*
  - *A refactoring is to replace primitive data types with types that are natural for the domain*



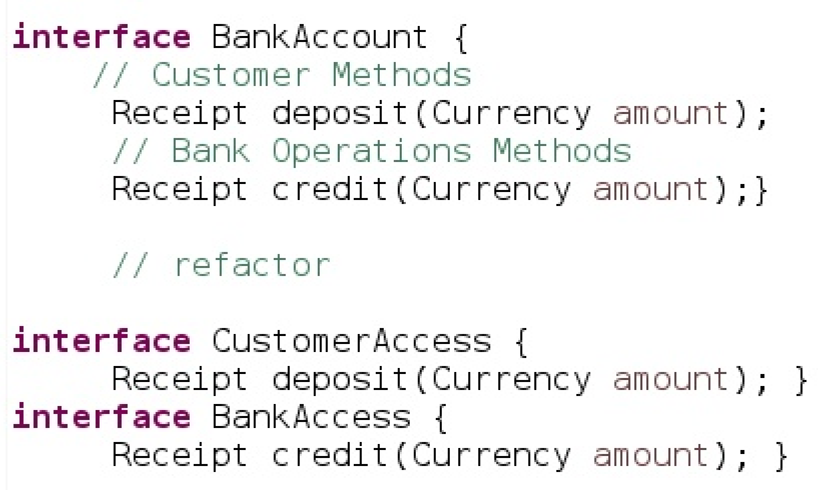
```java
public boolean deposit(int amount);

// refactor

public Receipt deposit(Currency amount);
```

Code Smell Example - Large Interface

- OOP Interface Segregation Principle:
  - *Many client-specific interfaces are better than one general-purpose interface.*
- Code smell – Overly Complex Interface
  - *This smell occurs because several different clients have their specif-ic methods lumped together in a single interface*
  - *A refactoring is to replace the large general interface with smaller client specific interfaces.*



```
interface BankAccount {
    // Customer Methods
     Receipt deposit(Currency amount);
     // Bank Operations Methods
     Receipt credit(Currency amount);}

     // refactor

interface CustomerAccess {
     Receipt deposit(Currency amount); }
interface BankAccess {
     Receipt credit(Currency amount); }
```

Other Code Smells

- Duplicated Code:
  - *The same code occurring in more than one place is difficult to maintain and risky to change.*
  - *The code should be consolidated into a single location.*
- Long Method
  - *The longer a method is the more difficult it is to understand and the less likely the method is to be cohesive.*
  - *The method should be split into smaller and more specialized methods.*
- Large Class
  - *"God" or "Kitchen sink" classes do too much – they usually have low cohesion and tend to be resource intensive*
  - *Large classes can be replaced by a collection of related smaller and more cohesive classes.*

Other Code Smells

- Shotgun Surgery
  - *When a single design change causes a number of small changes to a lot of different classes.*
  - *Some functionality has been spread over a number of classes that should be consolidated in a single location.*

- Comments
  - *Dense sets of comments are often used to explain badly written code.*
  - *Comments should be used to provide context for understanding the code – for example, details of the algorithm implemented*
  - *The code should be rewritten to be self documenting.*

Refactoring After Functional Changes

Refactoring After Functional Changes

- Changes in requirements typically require a change in the functionality of the code
  - *Changing functionality always produces changes in the structure of the code.*
- A mismatch can occur between the old code structure and the new functionality
  - *The code should be refactored to a structure optimal for the changed functionality*

Refactoring After Design Changes

Refactoring After Design Changes

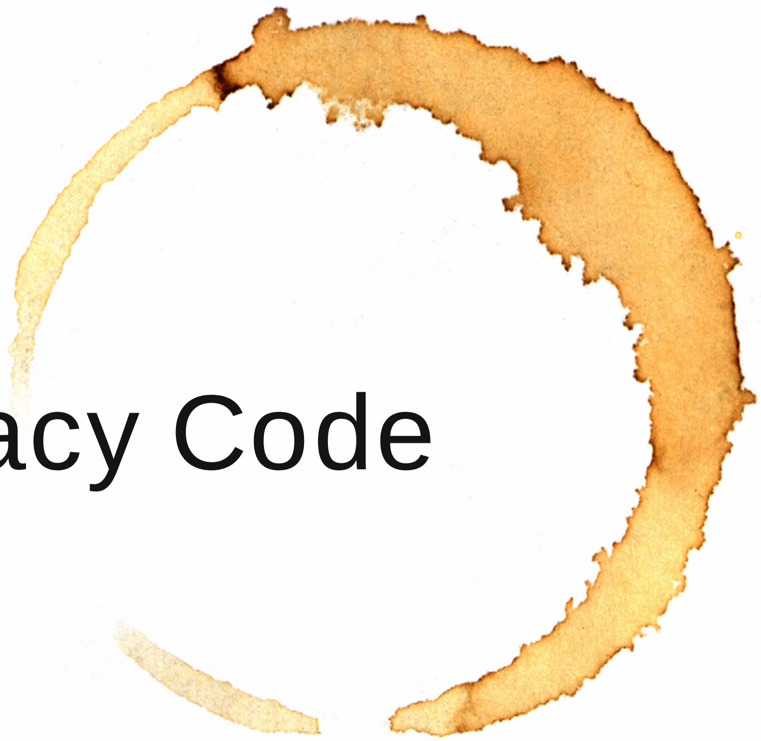
- Changes in design of the code or the application design may require restructuring the code
  - *Design changes can be triggered by a chance in functional or non-functional requirements*
  - *Design changes can occur for other reasons, eg. conformance to changes in software architectural standards*
- Functional and design changes may occur simultaneously
  - *In this case refactoring is critical to reduce project and performance risk*

Refactoring to Design Patterns

Design Patterns

- Design Patterns are standard patterns for structuring sub-systems
- Restructuring is done to meet non-functional requirements
  - *For example: throughput, reliability, resource usage, response time, loading and security*
- Refactoring to design patterns does not change functionality
- There exist a number of standard refactorings to patterns

Refactoring Legacy Code



Legacy Code

- Legacy code may structured or other non-OO code
  - *There are specific refactorings that can be done do convert structured code to an OO design*
- A more common refactoring is improving bad OO code
  - *Most of the standard refactorings deal with this case*
- Deciding to refactor legacy code should be considered carefully
  - *The determining factor is the cost benefit analysis*
  - *If legacy code will be modified a lot, refactoring may be a good choice*

When Not to Refactor

- Refactoring does not replace the design process
  - *Refactoring cannot be used effectively to impose a design where no design exists*
- Sometimes code is badly structured that refactoring is pointless
  - *It is more efficient to redesign and rewrite the code instead*
- The decision to redesign or refactor is GEQ decision
  - *The determining factor, as with legacy code, is the cost benefit analysis*

End of Module 7