Software Refactoring

Mika Mäntylä,
Helsinki University of Technology
Software Business and Engineering Institute
Outline

- Refactoring - Improving the created construction
- When to Refactor: Bad code smells & Anti-Pattern
What is Software Refactoring

- Also know as Restructuring
- Fowler’s book title as definition:
  - Refactoring - Improving the design of existing code
- Refactoring increases
  - Software maintainability (=ease of further development and bug fixing)
    - Evolvability would be a better term
- Refactoring does not
  - Change the observable behavior of the software
    - Performance tuning is not refactoring
- Good programmers have always refactored
Why Refactoring, if we have perfect design?

MY NEW DESIGN WILL MEET ALL OF OUR CUSTOMERS' CURRENT AND FUTURE NEEDS.
Why Refactoring?

- Software Evolution (Lehman’s Laws)
  - Law 1: Change is inevitable
    - Software which is used in a real-world environment must change or become less and less useful in that environment
    - Does not hold for all software, e.g. Algorithms, protocols may remain unchanged for decades
  - Law 2: Each change increases complexity
    - As an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon
  - Consequence: Soon the perfect design is outdated
Why Refactoring cont’d?

- **Learning**
  - Continuous learning (of technology, domain, customer problem, etc) exposes new and better design alternatives
  - Consequence: Perfect design is not possible
  - In some cases even big refactorings are not enough
    - Brooks: *plan to throw one away; you will, anyhow*
    - Managers may not allow complete rewrite

- **Agile software development methods**
  - Accept the consequences of evolution and learning
    - Need of continuous refactoring
  - XP embrace change
Why Refactoring cont’d?

- Refactoring is a part of software development
  - Earlier it was considered as less important maintenance activity
  - Software maintenance starts from day 1
  - Design is not enough

- Benefits of Refactoring according to Fowler and Arnold
  - It improves software design
  - Programs are easier to understand
  - It helps you finding bugs
  - Increased software development speed
  - Easier testing, auditing, documenting
  - Reduced dependency on individuals
  - Greater job satisfaction
  - Extending system’s lifetime
  - Preserving software asset value to organization
void printSalerySheet() {
    printCompanyLogo();
    //Calculate salery
    int hoursWorked = getHours();
    int salary = hoursWorked * getWage();
    System.out.println("Salary: " + salary);
}

void printSalery() {
    printCompanyLogo();
    System.out.println("Salary: " + calculateSalery());
}

int calculateSalery() {
    int hoursWorked = getHours();
    int salary = hoursWorked * getWage();
    return salary;
}
Refactoring Example 2/4 – Null Object

```java
Car car = null;
...
if (car == null)
    engine = Engine.getBasic();
else
    engine = car.getEngine();
```

```java
Car car = new NullCar();
...
engine = car.getEngine();

Class NullCar{
    Engine getEngine(){
        return Engine.getBasic()
    }
}
```
Refactoring Example 3/4 – Switch statement

```java
Class Engine{
    int cylinderCount(){
        switch (type){
            case FAMILY_CAR_ENGINE:
                return getBaseSylinderNumber();
            case EXECUTIVE_CAR_ENGINE:
                return getBaseSylinderNumber() * 2;
            case RACE_CAR_ENGINE:
                return getRaceCarCylinderNumber();
        }
    }
}
```

Mika Mäntylä
Refactoring Example 4/4 – Inheritance

- Engine
  - +cylinderCount()

  - FamilyCarEngine
    - +cylinderCount()

  - ExecutiveCarEngine
    - +cylinderCount()

  - RaceCarEngine
    - +cylinderCount()
Refactoring Example 4/4 – Inheritance

Duplication

Parallel Inheritance Hierarchies

Mika Mäntylä
Refactoring Example 4/4 – Inheritance

Improved Solution

```
+ carType : CarType

0..1
```

```
+ cylinderCount()

1
```

PetrolEngine

DieselEngine

AlcoholEngine

ExecutiveCar

FamilyCar

RaceCar

```
+ cylinderCount()
```

```
+ cylinderCount()
```

```
+ cylinderCount()
```
Practical notes on refactoring

- **When to refactor**
  - Integrate refactoring with normal software development, e.g.
    - Refactor when adding a feature - First refactor then add a feature
  - Make a time box for software refactoring
    - Have a refactoring day / week / iteration, e.g. refactor one day per week
      - Microsoft 20% tax for refactoring

- **What do I tell my manager**
  - Mostly problem when manager is not technically oriented
  - Short answer: Don’t
  - Long answer:
    - Present the benefits of refactoring (Fowler & Arnold slide 7)

- **What should I have before staring refactoring**
  - Refactoring tools
    - Tools enable automated and safe refactorings
    - Has increased considerably during past years
    - A few years back, changing a variable/method/class name was painful
  - Unit tests
    - Manual refactorings can break the system
Terms of “Refactoring domain”

- Refactoring
  - Controlled way to improve the software’s structure without changing its observable behavior
  - Refactoring should be continuous activity performed by all software developers

- Re-engineering
  - Examination and alteration of a system to reconstitute it in a new form
  - Consists of Reverse-engineering and Forward engineering
  - Large system modifications are referred as re-engineering

- Reverse-engineering
  - Analyze system to identify system components and their relationship
  - Create representation of system in higher abstraction level
Summary: Refactoring

- Perfect design is impossible
  - Software evolution
  - Continuous learning and better design alternatives
- Refactoring is needed to keep the code evolvable
- Refactoring is no excuse to omit software design
- Examples of Refactorings
  - Look more from Fowler’s Book on Refactoring
Outline

- Refactoring - Improving the created construction
- When to Refactor: Bad code smells & Anti-Patterns
Anti-Patterns and Bad Code Smells

- Describe sour constructs and other common software development mistakes
Pattern History

- **Design Patterns**
  - Represent reusable designs
    - Strategy, Command, Singleton, etc
  - History
    - Based on pattern languages which focused on traditional architecture (1977)
    - Rediscovered by software people 1987 and published by GoF in 1994
  - Motivation
    - Capsulate years of industry experience on software development

- **Anti-Patterns**
  - Represent frequently occurring undesirable pattern
  - History
    - Have been around longer than Design Patterns
    - Fred Brooks and The Mythical Man-Month 1979
    - Term Anti-Pattern appeared soon after the book by GoF in 1994
  - Motivation
    - Learning from mistakes
Anti-Patterns & Bad Code Smells

- Anti-Pattern cover wider range of topics
  - Development
    - Software Development
  - Architectural
    - Systems Architecture
  - Managerial
    - Organization and Process

- Bad code smells
  - Are in fact development level anti-patterns
Bad Code Smells - Taxonomy

- **Bloaters**
  - When something is too large, e.g.:
    - Long Method
    - Large Class
    - Long Parameter List,
  - These smells likely grow little bit a time
    - Hopefully nobody designs e.g. Long Methods
Bad Code Smells - Taxonomy

- Object-Orientation abusers
  - Object Oriented concept not fully understood and utilized, e.g.:
    - Switch statements
    - Alternative Classes with Different Interfaces
  - C-programmer in a Java context
Switch statement

Class Engine{
    int cylinderCount()
    
    switch (type){
        case FAMILY_CAR_ENGINE:
            return getBaseSylinderNumber();
        case EXECUTIVE_CAR_ENGINE:
            return getBaseSylinderNumber() * 2;
        case RACE_CAR_ENGINE:
            return 8;
    }
}
Alternative Classes with Different Interfaces

**Engine**
- typeCode : int
- raceCar : RaceCar
- familyCar : FamilyCar
- executiveCar : ExecutiveCar

**RaceCar**
- +cylinders()

**FamilyCar**
- +cylinderCount()

**ExecutiveCar**
- +cylinderNumber()
Improved Solution

Engine
- carType : CarType

CarType
+ cylinderCount()

PetrolEngine

DieselEngine

AlcoholEngine

ExecutiveCar
+ cylinderCount()

FamilyCar
+ cylinderCount()

RaceCar
+ cylinderCount()
Bad Code Smells - Taxonomy

- Change Preventers
  - These smells make changing the system unnecessarily difficult, e.g.
    - Shotgun surgery
      - Change the database, e.g., from Oracle to SQLServer requires changes to several classes
  - Violate principle one external change should effect to one class only
Bad Code Smells - Taxonomy

- Dispensables
  - All code needs effort to understand and maintain
  - If code is not used or redundant it needs to be removed, e.g.:
    - Dead code
    - Speculative Generality
    - Duplicate Code
**Duplicate Code**

- **What others say**
  - “Once And Only Once” (Cunningham & Cunningham)
  - “Don’t Repeat Yourself” (Hunt & Thomas)
  - “Number one in the stink parade” (Fowler & Beck)
  - In other words
    - “Ohjelmoi vain sellaista mitä ei ole vielä ohjelmoitu”

- **Syntactic duplication**
  - Fragments of code that are identical or near identical
  - Tools can offer great help, e.g. Simian

- **Semantic duplication**
  - Fragments of code that have identical intent but differ at code level
    - E.g. bubble sort and quick sort
  - No tool help available
Bad Code Smells - Taxonomy

- **Couplers**
  - Low coupling between objects/classes is one the desirable goals of OO software, e.g.
    - Feature Envy
    - Message Chains
      - a.getB().getC().getD().doOperation
  - Too much delegation (= minimizing coupling) can be bad as well, e.g.
    - MiddleMan
Summary: Bad code smells & Anti-Patterns

- Present frequently occurring problems in the code and design
- Anti-patterns can also be found in higher levels, e.g. software processes, management
- Taxonomy of code smells
  - Bloaters
  - Object-Orientation abusers
  - Change Preventers
  - Dispensables
  - Couplers