Design Patterns

Engr. Abdul-Rahman Mahmood

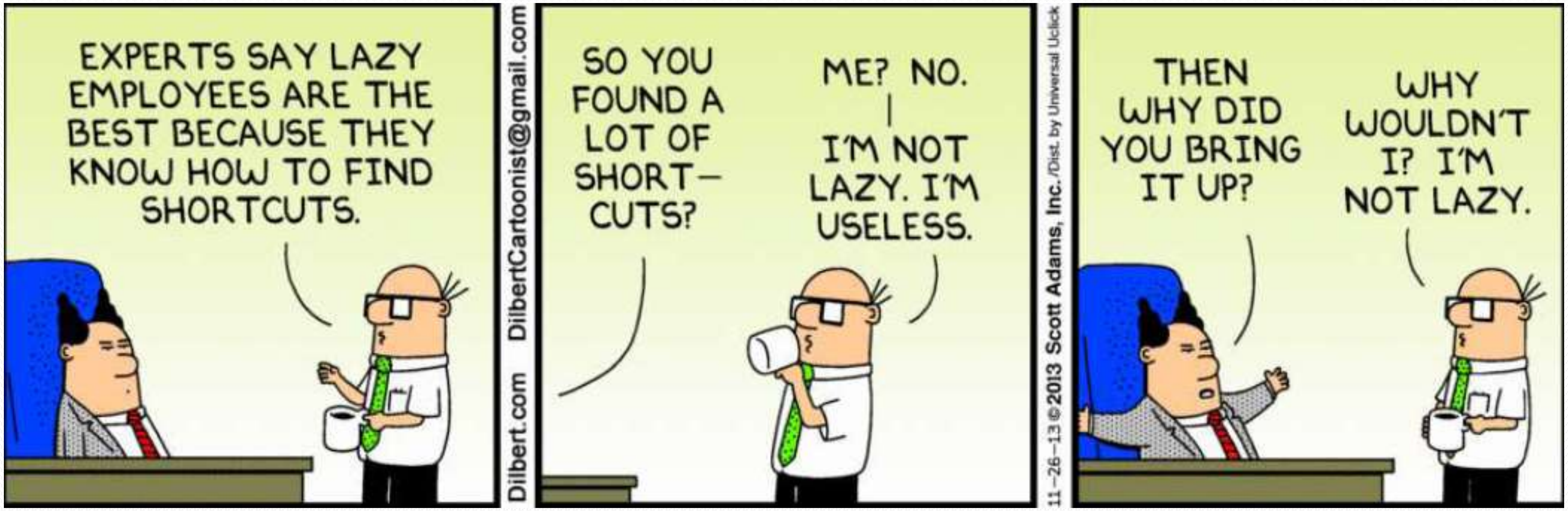
DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com	alphasecure@gmail.com
alphapeeler.sf.net/pubkeys/pkey.htm	http://alphapeeler.sourceforge.net
pk.linkedin.com/in/armahmood	http://alphapeeler.tumblr.com
www.twitter.com/alphapeeler	armahmood786@jabber.org
www.facebook.com/alphapeeler	alphapeeler@aim.com
abdulmahmood-sss alphasecure	mahmood_cubix 48660186
armahmood786@hotmail.com	alphapeeler@icloud.com
http://alphapeeler.sf.net/me	http://alphapeeler.sf.net/acms/

The Need for Design Principles - Refactoring

Lecture 04

Can you understand it?


EXPERTS SAY LAZY EMPLOYEES ARE THE BEST BECAUSE THEY KNOW HOW TO FIND SHORTCUTS.
Dilbert.com DilbertCartoonist@gmail.com
SO YOU FOUND A LOT OF SHORT-CUTS?
ME? NO. I'M NOT LAZY. I'M USELESS.
11-26-13 ©2013 Scott Adams, Inc./Dist. by Universal Uclick
THEN WHY DID YOU BRING IT UP?
WHY WOULDN'T I? I'M NOT LAZY.

Software Rots

- When the requirements change, the system must change often in ways not anticipated by the initial design. Over time the software begins to acquire design smells... The software rots!
  - The design should be kept as clean, simple, and as expressive as possible
    - Never say, we'll do that later
    - ...because you won't!
  - When a requirement changes, the design should be updated to be resilient to that kind of change in the future.

Refactoring

- How to modify design and code to prevent rot?
- Refactoring
  - ...the process of changing a software system in such a way that <u>it does not alter the external behavior of the code yet improves its internal structure</u>.
  - You can refactor code & design.

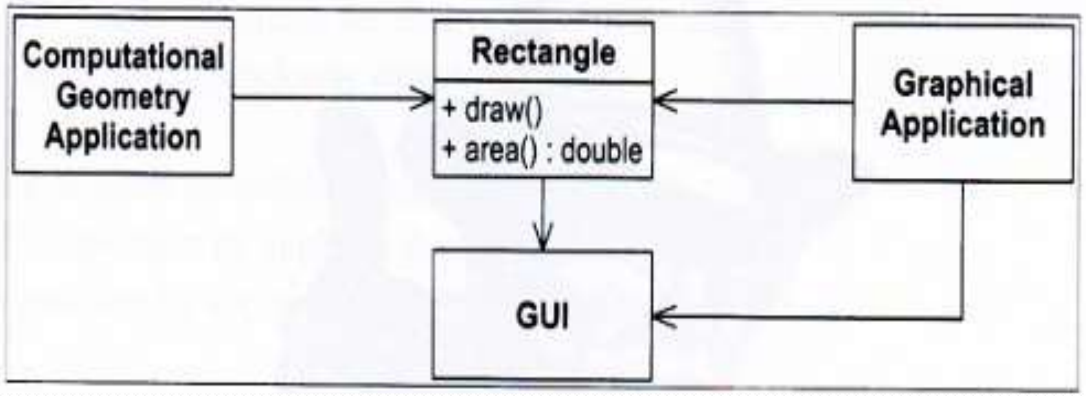
Single-Responsibility Principle (SRP)

- A class should have only one responsibility.

OR

- A class should have only one reason to change.
- A class with several responsibilities creates unnecessary couplings between those responsibilities

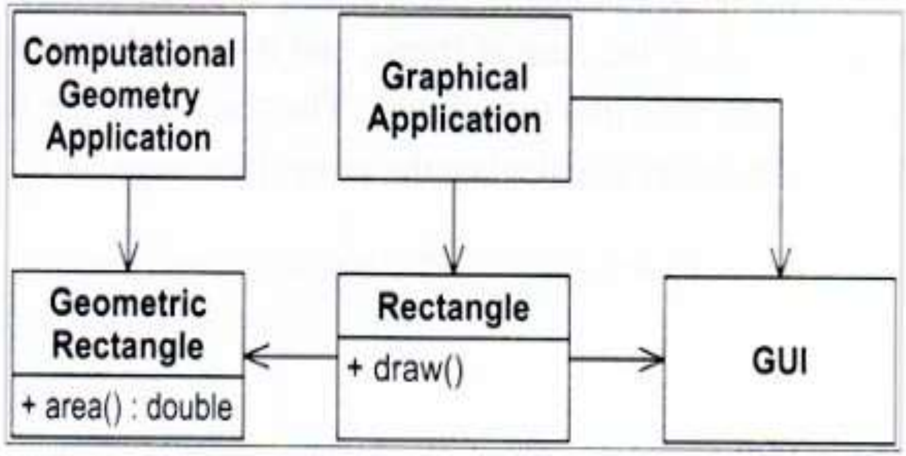
e.g. Rectangle Class



- The <u>Geometry Application </u>is concerned with the mathematics of geometric shapes
- The <u>Graphical Application </u>may also involve some geometry, but it also needs to draw geometric shapes
- The Rectangle class has two responsibilities:
- Provide a mathematical model of a rectangle Render a rectangle

e.g. Rectangle Class

- Problems created:
  - Inclusion: The GUI must be included in the Geometry Application
  - A change required for one application may affect the other (e.g. adding a color attribute)
- Solution:
- Separate the two responsibilities (math rep. + drawing) into two separate classes



Ex.2 SRP – Employee Class

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  public boolean isPromotionDueThisYear(){
    //promotion logic implementation
  }
  public Double calcIncomeTaxForCurrentYear(){
    //income tax logic implementation
  }
  //Getters & Setters for all the private attributes
}
```

- The Employee class looks logically correct. It has all the employee attributes like employeeId, name, age, address & dateOfJoining. It even tells you if the employee is eligible for promotion this year and calculates the income tax he has to pay for the year.
- However, Employee class breaks the Single Responsibility Principle. Lets see how

Ex.2 SRP – Employee Class

- The logic of determining whether the employee is due this year is actually not a responsibility which the employee owns. - HR department owns this responsibility.
- Similarly, income tax calculation is not a responsibility of the Employee. - finance department's responsibility
- Lastly, Employee class should have the single responsibility of maintaining core attributes of an employee.

Ex.2 SRP – Employee Class

- Solution

HRPromotions.java

```
public class HRPromotions{
  public boolean isPromotionDueThisYear(Employee emp){
    //promotion logic implementation using the employee info. passed
  }
}
```

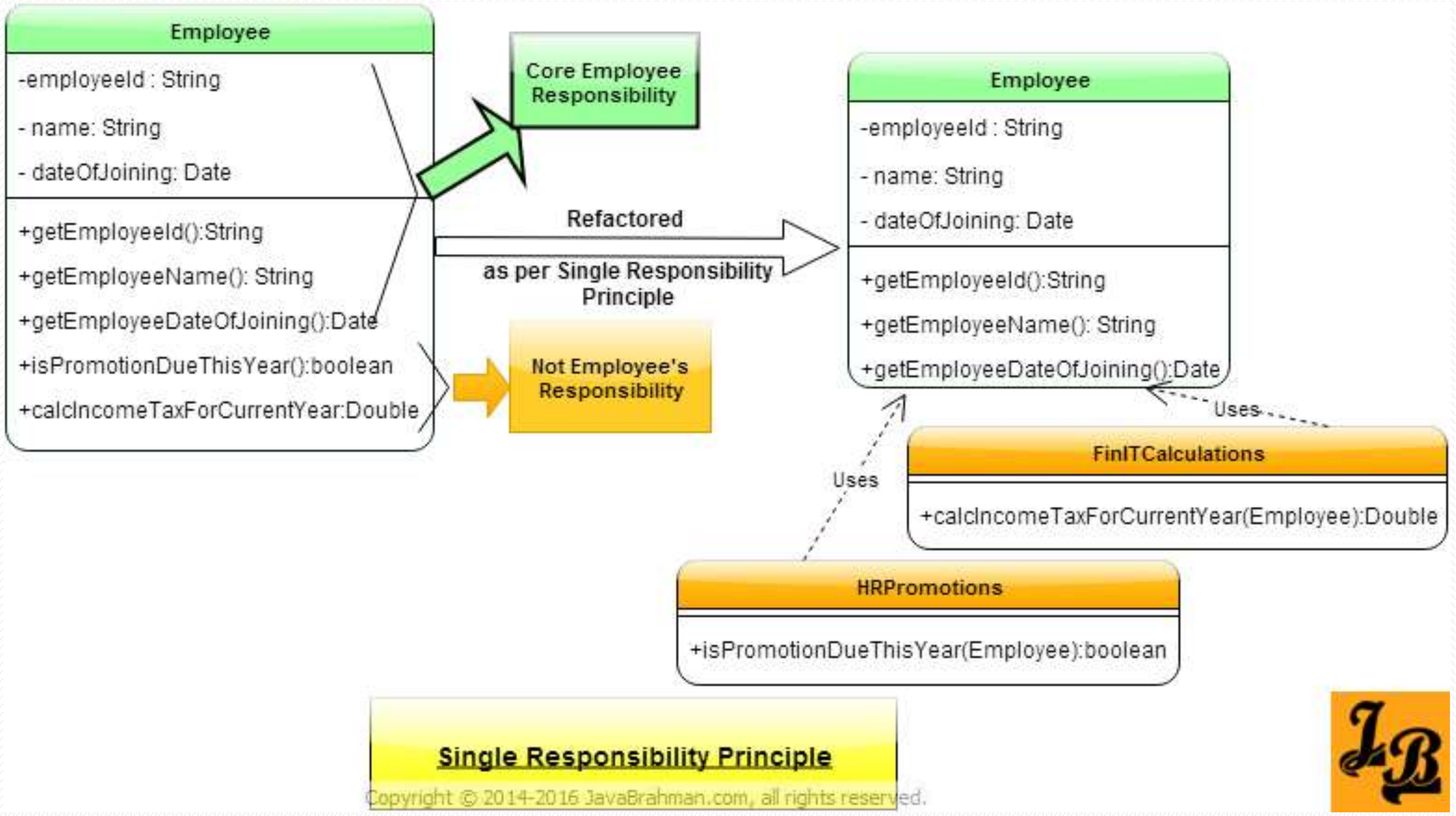
FinITCalculations.java

```
public class FinITCalculations{
  public Double calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation using the employee info. passed
  }
}
```

Employee.java adhering to Single Responsibility Principle

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  //Getters & Setters for all the private attributes
}
```

SRP based refactoring


Employee
-employeeId : String
- name: String
- dateOfJoining: Date
+getEmployeeId():String
+getEmployeeName(): String
+getEmployeeDateOfJoining():Date
+isPromotionDueThisYear():boolean
+calcIncomeTaxForCurrentYear:Double
Core Employee Responsibility
Not Employee's Responsibility
Refactored
as per Single Responsibility Principle
Employee
-employeeId : String
- name: String
- dateOfJoining: Date
+getEmployeeId():String
+getEmployeeName(): String
+getEmployeeDateOfJoining():Date
Uses
FinITCalculations
+calcIncomeTaxForCurrentYear(Employee):Double
Uses
HRPromotions
+isPromotionDueThisYear(Employee):boolean
Single Responsibility Principle
Copyright © 2014-2016 JavaBrahman.com, all rights reserved.

Design Tip of the Day

Meaningful Names in Design and Implementation

- Quicker to Comprehend
- Easier to Remember
- Clear Intent
- Maintainable Code

Symptoms of Poor Design

Software Design Smells

Rigidity	• The design is hard to change
Fragility	• The design is easy to break
Immobility	• The design is hard to reuse
Viscosity	• Doing things right is harder than doing things wrong
Needless Complexity	• Overdesign
Needless Repetition	• Copy paste code
Opacity	• Disorganized expressions

Rigidity

Rigidity

- The system is hard to change because every change forces many other changes to other part of the system
- Single change causes cascade of subsequent changes in dependent modules
- The more modules must be changed the more rigid the design
- Comes from High coupling and Low Cohesion
- Telltale sign: "Huh, it was a lot more complicated than I thought."
- Examples
  - Status fields – New status values
  - Dependency on concrete classes

Rigidity Example

```
class A
{
  B _b;
  public A() {
    _b = new B();
  }
  public void Foo() {
    // Do some custom logic.
    _b.DoSomething();
    // Do some custom logic.
  }
}
class B {
   public void DoSomething() {
    // Do something
   }
}
```

Here class A depends on class B very much. So, if in future you need to use another class instead of class B, this will require changing class A and will lead to it being retested. In addition, if class B affects other classes, the situation will become much complicated.

```
interface IComponent {
  void DoSomething();
}
class A {
  IComponent _component;
  public A(IComponent component) {
    _component = component;
  }
  void Foo() {
     // Do some custom logic.
     _component.DoSomething();
     // Do some custom logic.
   }
}
class B implements IComponent {
  void DoSomething() {
    // Do something
  }
}
```

The workaround is an abstraction that is to introduce the IComponent interface via the constructor of class A. In this case, it will no longer depend on the particular class B and will depend only on the IComponent interface. Class B in its turn must implement the IComponent interface.

Fragility

Fragility

- The tendency of a program to break in many places when a single change is made
- Code breaks in unexpected places that have no conceptual relationship with the changed area
- On every fix the software breaks in unexpected ways
- Fixing the problems causes new problems
- Examples
  - Dependency on concrete classes
  - No edge cases or bound checks

Fragility of the design

- Eaxmple: Kids play with a deck of cards to build a house of cards. When it comes about 3-4 feet you very careful to put stuff on it. And you say don't open the door. You very quite and move very slowly. And we grow and create software. We type the code it compiles, we move slowly from the computer, and we say “The code works don't touch it, I'm going home! ShipIt!”
- Telltale signs:
  - Some modules are <u>constantly on the bug list</u>
  - <u>Time is used finding bugs</u>, not fixing them
  - Programmers are <u>reluctant to change </u>anything in the code

Immobility

I am asking for some one, NOT everyone!

Inability to re-use software

Software is easy to rewrite instead of reuse

Immobility

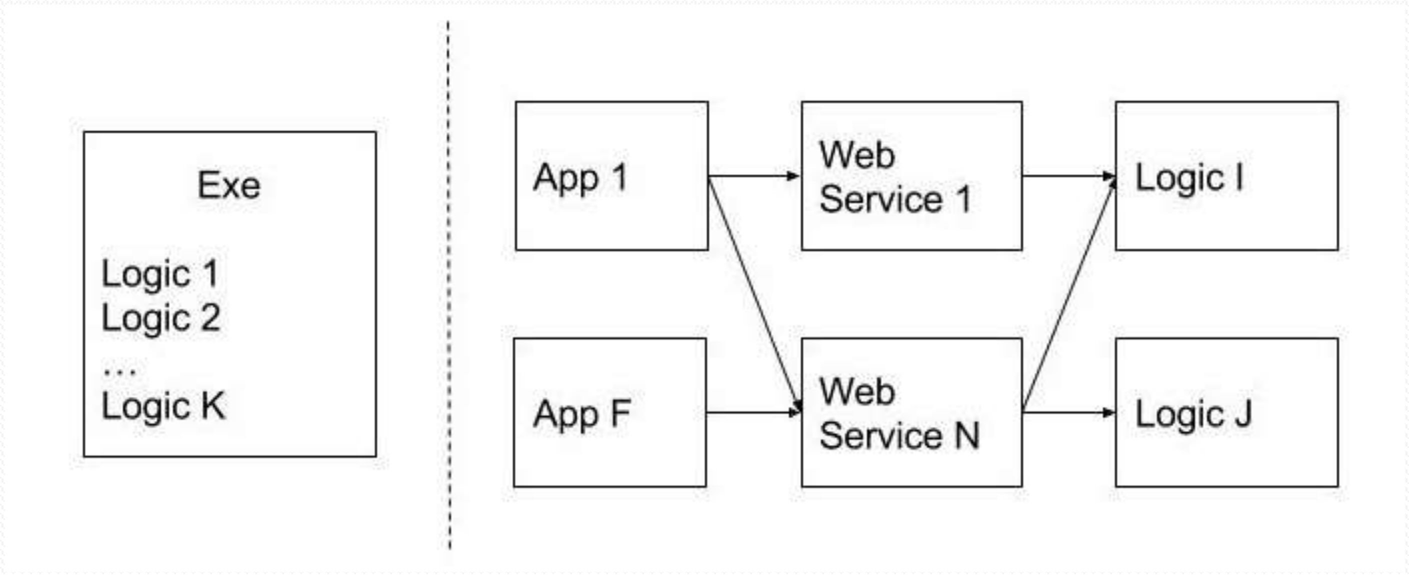
- It is hard to separate the system into components that can be reused in other systems because risk involved separating parts from original system is too much
- The useful modules have too many dependencies
- The cost of rewriting is less compared to the risk to separate those parts
- Telltale sign: A module could be reused but the effort and risk of separating it from the original environment is too high
- Examples
  - Too many constants
  - Too many global variables
  - Too much coupling

Immobility

- Consider a desktop program, the whole code of which is implemented in the (.exe) file and business logic is not built in separate modules or classes. Later, the developer has faced the following business requirements:
  - To change the user interface by turning it into a Web application;
  - To publish the functionality of the program as a set of Web services available to third-party clients to be used in their own applications.

Immobility

- The picture below shows an example of an immobile design in contrast to the one that does not have this indicator.



Viscosity

- Doing things right is harder than doing things wrong
- When the design preserving methods are more difficult to use than the hacks, the viscosity of the design is high – the hack is cheaper to implement than the solution within the design
- When the development environment is slow and inefficient, developers will be tempted to do wrong things
- -> Unit Testing will help!
- Examples
  - Use of public member fields
  - Insufficient documentation of the implemented classes
  - Highly complex design
  - Low performing code

Needless Complexity

- Overdesign! Design contains infrastructure that adds no direct benefit
- Design becomes littered with constructs that are never used
- Makes software complex and difficult to understand
- Job security! Nobody can understand except me ☺
- "Simple things should be simple and complex things should be possible" - Alan Kay
- Examples
  - Too much generic code
  - Make use of middle men
  - Long hierarchy of interfaces and classes

Needless Repetition

Needless Repetition

- The design contains repeating structures that could be unified under a single abstraction
- Problem is due to developer’s abuse of cut and paste
- It is really hard to maintain and understand the system with duplicate code
- Examples
  - Bad Algorithms
  - Make use of switch statements
  - Duplicate code in different classes
  - Avoiding abstraction – copying the existing code and tweak according to the requirements

Opacity

- Very <u>difficult to understand code </u>on what it does.
- It does not express its purpose well
- The boss says don't touch that code the guy who wrote it is no longer works here!
- Telltale sign: You are reluctant to fix somebody else’s code or even your own!
- Example
  - Too much commented code
  - Incorrect naming convention
  - Not well indented code
  - Algorithm written in a complex manner

Signs of Good Design

Adaptability	• The design is easy to change
Robustness	• The design is hard to break
Reusability	• The design can be reused
Fluidity	• It is easy to do the right thing
Simplicity	• The design is the “simplest thing that will work”
Terseness	• No unneeded duplication of code
Perspicuity	• Organized and clear