Refactoring: Improving the Design of Existing Code

CS 4354
Summer II 2016

Jill Seaman

What is Refactoring?

- Refactoring: disciplined technique for changing a software system: altering its internal structure without changing its external behavior
  - To improve readability.
  - To improve structure.
  - Reduce complexity.
  - Easier to modify in the future
- No added functionality!!
- Preventative maintenance: reduces future (or current) maintenance costs

Refactoring process

- Required during Iterative Development to maintain the quality of the code as new code is added.
  - ✦Ongoing process, from start of development.
  - ✦Applied on a small scale
  - ✦Avoids structure degradation from the start
- Steps of the process:
  1. write new code and new tests
  2. test code using **all** the tests, make sure all tests pass
  3. refactor the code (in a series of steps)
  4. test code using all the tests. If any tests fail, repair or rollback changes.
  5. repeat from step 1 or step 3

Refactoring process

- **Refactoring** (noun): a <small> change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- **Refactor** (verb): to restructure software by applying a series of refactorings without changing its observable behavior.
- I'm often asked about how it should be scheduled. Should we allocate two weeks every couple of months to refactoring?
- In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you do that other thing.

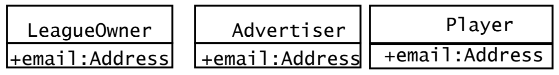
from: sourcemaking.com/refactoring

Some Refactorings

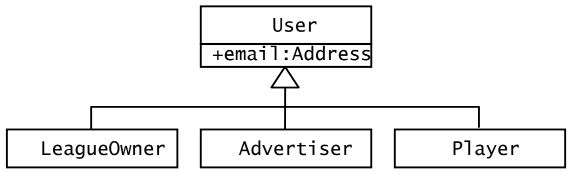
- Rename Method/Field/Class/Variable: change the name and all references to it.
- Extract Method: Replace some inline code with a call to a (new) method containing that code.
- Encapsulate Field: Make a public field private, generate getters and setters, replace references to the field with calls to these.
- Move Method/Field: Move element to the new class, use delegation to replace references to these elements in the original class.
- Pull Up Method/Field: If two subclasses use the same method, move the method to the superclass.
- Push Down Method/Field: If a method is used for only some of the subclasses, move it to those subclasses.

Example: Extract Superclass and Pull Up Field:

- Class diagram before transformations
  (You have three classes with similar attributes):



- Class diagram after transformations:



Refactoring example: Pull up field

- Two or more subclasses have the same field:

```
public class User {
}

public class Player extends User {
  private String email;
  //...
}
public class LeagueOwner extends User {
  private String eMail;
  //...
}
public class Advertiser extends User {
  private String email_address;
  //...
}
```

```
public class User {
  private String email;
}

public class Player extends User {
  //...
}

public class LeagueOwner extends User {
  //...
}

public class Advertiser extends User {
  //...
}
```

- This may break the code. . . how so?

Refactoring example: Pull up Constructor Body

- You have constructors on subclasses with identical bodies:

```
public class User {
   private String email;
}

public class Player extends User {
   public Player(String email) {
     this.email = email;
   }
}
public class LeagueOwner extends User{
   public LeagueOwner(String email) {
     this.email = email;
   }
}
public class Advertiser extendsUser{
   public Advertiser(String email) {
     this.email = email;
   }
}
```

```
public class User {
   private String email;
   public User(String email) {
     this.email = email;
   }
}
public class Player extends User {
     public Player(String email) {
        super(email);
     }
}
public class LeagueOwner extends User {
     public LeagueOwner(String email) {
        super(email);
     }
}
public class Advertiser extends User {
     public Advertiser(String email) {
        super(email);
     }
}
```

- This may fix some broken references to email in the subclasses

Where to apply refactoring (bad smells)

- Duplicate code
  - ✦Same or very similar code found at various places in a program.
  - ✦[Extract method]: put similar code into a single method/function
- Long method
  - ✦Long methods are difficult to understand, modify.
  - ✦Redesign as many shorter methods [Extract method]
- Switch statements
  - ✦Multiple switch statements with same case labels.
  - ✦Make subclasses, move each case into appropriate subclass.
  - ✦[Replace Conditional with Polymorphism]
- Data clumping
  - ✦The same group of items occur in several places in a program.
  - ✦Replace with a class that encapsulates all of the data [Extract Class]
- Speculative generality
  - ✦Unused parameters, classes, included "just in case". [Remove Parameter]

Bad Smell Refactoring example

- Note: classes are incomplete: constructors, getters/setters are not shown.
- What is the bad smell here?

```
class Employee {
   double monthlySalary;
   double commission;
   double bonus;
   int getType() { … }
   int payAmount() {
      switch (getType()) {
         case ENGINEER:
            return monthlySalary;
         case SALESMAN:
            return monthlySalary + commission;
         case MANAGER:
            return monthlySalary + bonus;
         default:
            throw new RuntimeException("Incorrect Employee");
      }
   }
}
```

Bad Smell Refactoring example

- [Replace Type Code with Subclasses]:
  Create a subclass for each value of the type code

```
class Employee {
   double monthlySalary;
   double commission;
   double bonus;
   int getType() { … }
   int payAmount() {
      switch (getType()) {
         case ENGINEER: return monthlySalary;
         case SALESMAN: return monthlySalary + commission;
         case MANAGER:  return monthlySalary + bonus;
         default: throw new RuntimeException("Incorrect Employee");
      }
   }
}
class Engineer extends Employee {
}
class Salesman extends Employee {
}
class Manager extends Employee {
}
```

Bad Smell Refactoring example

- [Replace Conditional with Polymorphism]:
  Move cases into overriding methods in subclasses

```
abstract class Employee {
   double monthlySalary;
   double commission;
   double bonus;
   abstract int payAmount();
}
class Engineer extends Employee {
   int payAmount() {
      return monthlySalary;
}  }
class Salesman extends Employee {
   int payAmount() {
      return monthlySalary + commission;
}  }
class Manager extends Employee {
   int payAmount() {
      return monthlySalary + bonus;
}  }
```

- Now we can clean this up further using another refactoring.

Bad Smell Refactoring example

- [Push down field]: when a field is used only by some subclasses

```
abstract class Employee {
   double monthlySalary;
   abstract int payAmount();
}
class Engineer extends Employee {
   int payAmount() {
      return monthlySalary;
   } }
class Salesman extends Employee  {
   double commission;
   int payAmount() {
      return monthlySalary + commission;
   } }
class Manager extends Employee  {
   double bonus;
   int payAmount() {
      return monthlySalary + bonus;
   }
}
```

Another example from Assignment 2

- Recall the Products sold by the Online Store
- Now the owner is adding a new product type: Electronics.
- Exercise: create a new subclass of Product for the Electronics, then apply some refactorings to clean up your code.
- Hint: use [Extract Superclass]

Product Type	Shipping credit	Commission
Movie (dvd)	$2.98	12% of sale price
Book	$3.99	15% of sale price
Toys	$4.49 + .50/lb	15% of sale price
Electronics	$4.49 + .50/lb	8% of sale price

Refactoring in Eclipse

- Many IDEs provide menu options to apply refactoring automatically.
- They also allow you to easily run JUnit tests before and after your refactorings.
- Usually you need to highlight the field, method, class or lines of code that you want to be affected by the refactoring, then select the desired refactoring from the menu.
- You may be presented with some options, and a preview of the changes before committing (especially if there are multiple files affected).

Refactoring with Eclipse: add the Electronics class

- Use the Assignment2 project that has the JUnit tests.
- Duplicate the Toy.java file using copy and paste in the Package Explorer pane (rename to Electronic when asked during paste).
- Rename TOY_COMMISSION_PERCENTAGE using **Refactor>Rename** (make change in place, hit enter).
- change the value of ELECTRONIC_COMMISSION_PERCENTAGE to 8
- Update showProductInfo (change toy to electronic)

- What is the bad smell now?
- How do you fix it (which refactoring to use)?

Refactoring with Eclipse: add the Electronics class

- In Electronic, **Refactor>Extract Superclass**, name = WeightedProduct
  - Types to extract: add Toy
  - extract weight, the 2 common named constants & shippingCredit()
  - Remove ShippingCredit() from both subclasses

Notes:

- the constructor is unchanged, do Pull up Constructor Body by hand.
- weight is protected. If you change it to private, you will get errors in Toy and Electronic.
- So do **Refactor>Encapsulate Field** on weight. When it's done, delete the setter (you don't need it).
- Run the JUnit tests again!

Now modify code in the Driver to allow the user to input an Electronic.