Refactoring

Refactoring

- Refactoring is:
  - restructuring (rearranging) code...
  - ...in a series of **small**, semantics-preserving transformations
    - i.e. the code keeps working
  - ...in order to make the code **easier to maintain** and modify
- Refactoring is not just any old restructuring
  - You need to keep the code working
  - You need to have unit tests to prove the code works
- There are numerous well-known refactoring techniques
  - You should be at least somewhat familiar with these before inventing your own

When to refactor

- You should refactor:
  - Any time that you see a better way to do things
    - “Better” means making the code easier to understand and to modify in the future
  - You can do so without breaking the code
    - Unit tests are essential for this
- You should not refactor:
  - Stable code (code that won’t ever need to change)
  - Someone else’s code
    - Unless you’ve inherited it (and now it’s yours)

- The longer you wait before paying your debt, the bigger the bill
  - common sense

- The bigger the mess, the less you want to clean it up
  - Joshua Kerievsky

- Perfection is reached not when there remains nothing to add, but when there remains nothing to remove
  - Antoine de Saint-Exupéry, 1900-1944

Design vs. coding

- “Design” is the process of determining, in detail, what the finished product will be and how it will be put together
- “Coding” is following the plan
- In traditional engineering (building bridges), design is perhaps 15% of the total effort
- In software engineering, design is 85-90% of the total effort
  - By comparison, coding is cheap

The refactoring environment

- Traditional software engineering
  - Modeled after traditional engineering practices (= design first, then code)
  - Assumptions:
    - The desired end product can be determined in advance
    - Workers of a given type (plumbers, electricians, etc.) are interchangeable
- “Agile” software engineering is based on different assumptions:
  - Requirements (and therefore design) change as users become acquainted with the software
  - Programmers are professionals with varying skills and knowledge
  - Programmers are in the best position for making design decisions
- Refactoring is fundamental to agile programming
  - Refactoring is sometimes necessary in a traditional process, when the design is found to be flawed

Back to refactoring

- When should you refactor?
  - Any time you find that you can improve the design of existing code
  - You detect a **“bad smell”** (an indication that something is wrong) in the code
- When can you refactor?
  - You should be in a supportive environment
    - agile programming team, or doing your own work
  - You should have an **adequate** set of unit tests

Example 1: switch statements

- **`switch`** statements are very rare in properly designed object-oriented code
  - Therefore, a **`switch`** statement is a simple and easily detected “bad smell”
    - Of course, not all uses of switch are bad
  - A **`switch`** statement should not be used to distinguish between various kinds of object
- There are several well-defined refactorings for this case
  - The simplest is the creation of subclasses

Example 1, continued

```
class Animal {
   final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
   int myKind;  // set in constructor
   ...
   String getSkin() {
      switch (myKind) {
         case MAMMAL: return "hair";
         case BIRD: return "feathers";
         case REPTILE: return "scales";
         default: return "integument";
      }
   }
}
```

Example 1, improved

```
class Animal {
     String getSkin() { return "integument"; }
}
class Mammal extends Animal {
     String getSkin() { return "hair"; }
}
class Bird extends Animal {
     String getSkin() { return "feathers"; }
}
class Reptile extends Animal {
     String getSkin() { return "scales"; }
}
```

How is this an improvement?

- Adding a new animal type, such as `Amphibian`, does not require revising and recompiling existing code
- Mammals, birds, and reptiles are likely to differ in other ways, and we’ve already separated them out (so we won’t need more `switch` statements)
- We’ve gotten rid of the flags we needed to tell one kind of animal from another
- Basically, we’re now using Objects the way they were meant to be used

JUnit tests

- As we refactor, we need to run JUnit tests to ensure that we haven’t introduced errors
-
```
public void testGetSkin() {
    assertEquals("hair", myMammal.getSkin());
    assertEquals("feathers", myBird.getSkin());
    assertEquals("scales", myReptile.getSkin());
    assertEquals("integument", myAnimal.getSkin());
}
```
- This should work equally well with either implementation
- The `setUp()` method of the test fixture may need to be modified

Common Code Smells in OOP

- Duplicated code
  - The worst stinker of all!
- Long method
  - Difficult to reuse and difficult to understand.
  - The object programs that live best and longest are those with short methods.
  - The best object programs are endless sequences of delegation. Their payoffs are explanation and sharing.
- Long parameter list
  - Hard to understand, methods become inconsistent and difficult to use and more likely to require change.

Common Code Smells in OOP

- Uncommunicative name
  - Does the name of the method succinctly describe what that method does?
  - Could you read the method’s name to another developer and have them explain to you what it does? If not, rename it or rewrite it.
- Inconsistent names
  - Pick a set of standard terminology and stick to it throughout your methods.
  - If you use “add”, then use “add” in all relevant situations – not “create”. Another example is “delete”/”remove”.
- Inappropriate intimacy
  - Classes delving into each other’s private parts.
  - Excessive use of class A’s accessors from class B.
  - Over intimate classes need to be broken up as lovers were in ancient days!

Common Code Smells in OOP

- Indecent exposure
  - Beware of classes that unnecessarily expose their internals. You should have a compelling reason for every item you make public. If you don’t, hide it.
- Large Classes
  - Too much code
  - Too many instance variables
  - Use layering and decomposition to avoid this problem.
- Classes with too little code
  - Often data classes that can become domain classes
  - Or candidates for downsizing!

Common Code Smells in OOP

- Divergent change
  - A class is commonly changed in different ways for different reasons
  - Example – database + functionality
- A large number of function classes
  - Indicate that the designers are thinking upside-down, i.e.. thinking of that which can be done to an object rather than that which can done by the object.

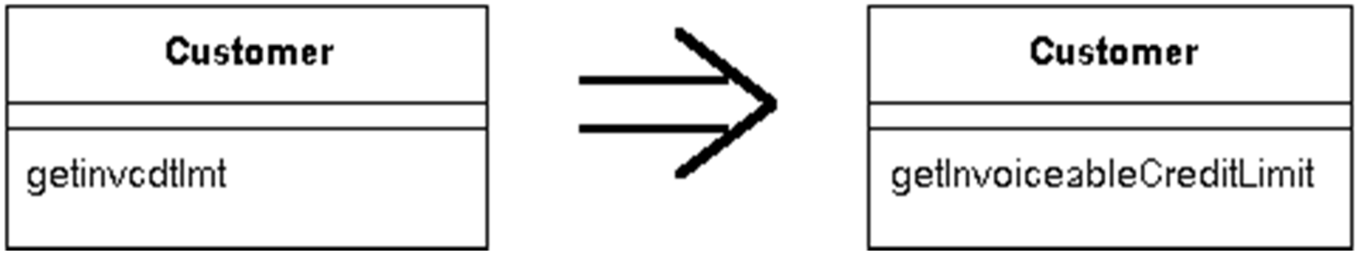
Common Code Smells in OOP

- Feature envy
  - A method that seems more interested in a class other than the one it actually is in.
  - Many messages to the same object from the same method
- Data clumps
  - Data items often hang around in groups together.
  - Extracting them into a class of their own will help reduce field lists and parameter lists.

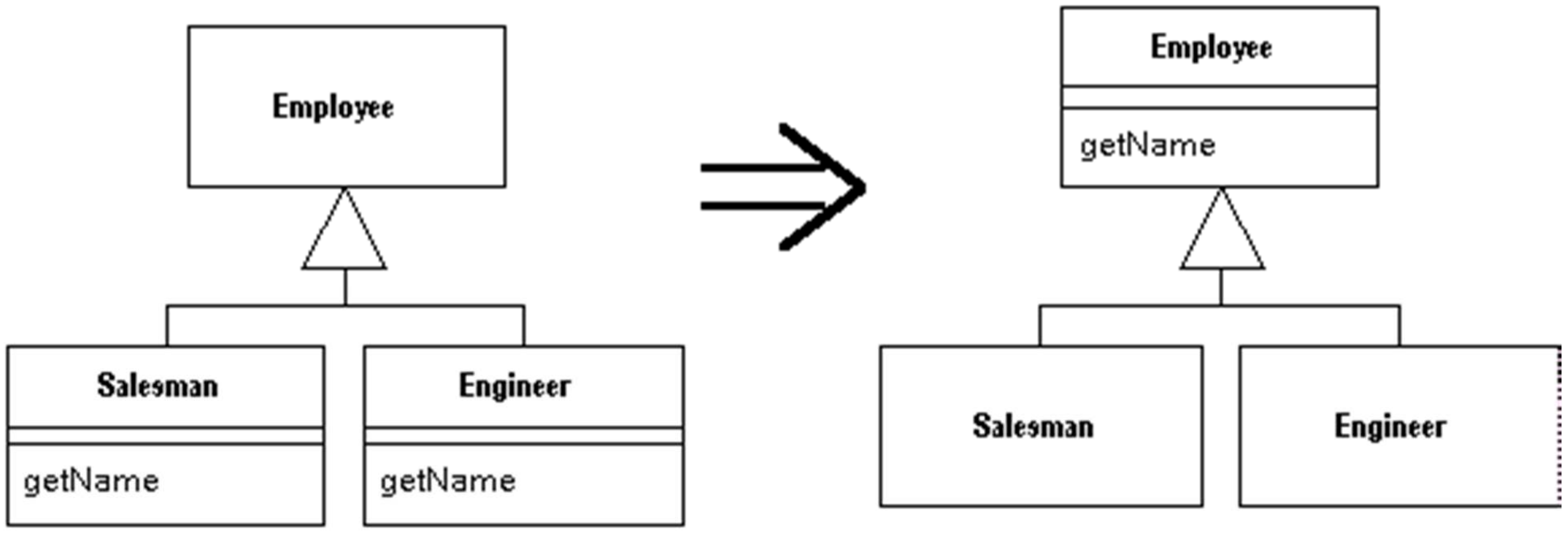
Common Code Smells in OOP

- Switch statements
  - Leads to duplication which leads to shotgun surgery
  - Consider polymorphism instead.
- Too many casts
  - Over generalization
- Comments
  - Often used as a deodorant for stinkers
  - Superfluous comments

Refactorings
Rename Method



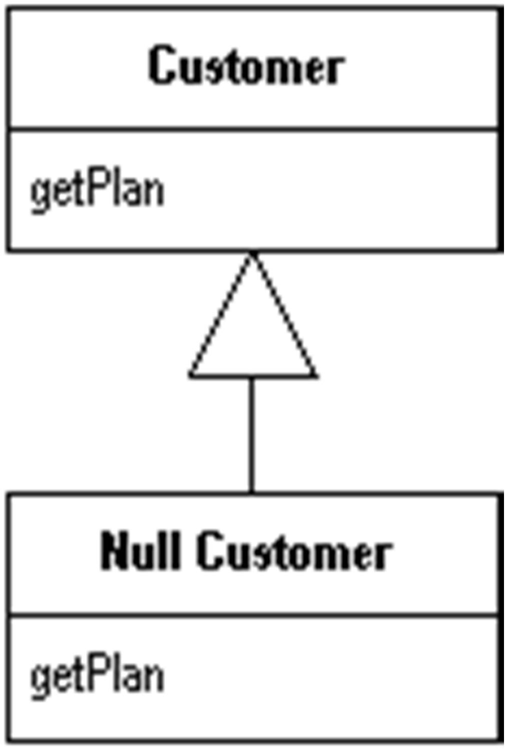
Refactorings
Pull Up Method



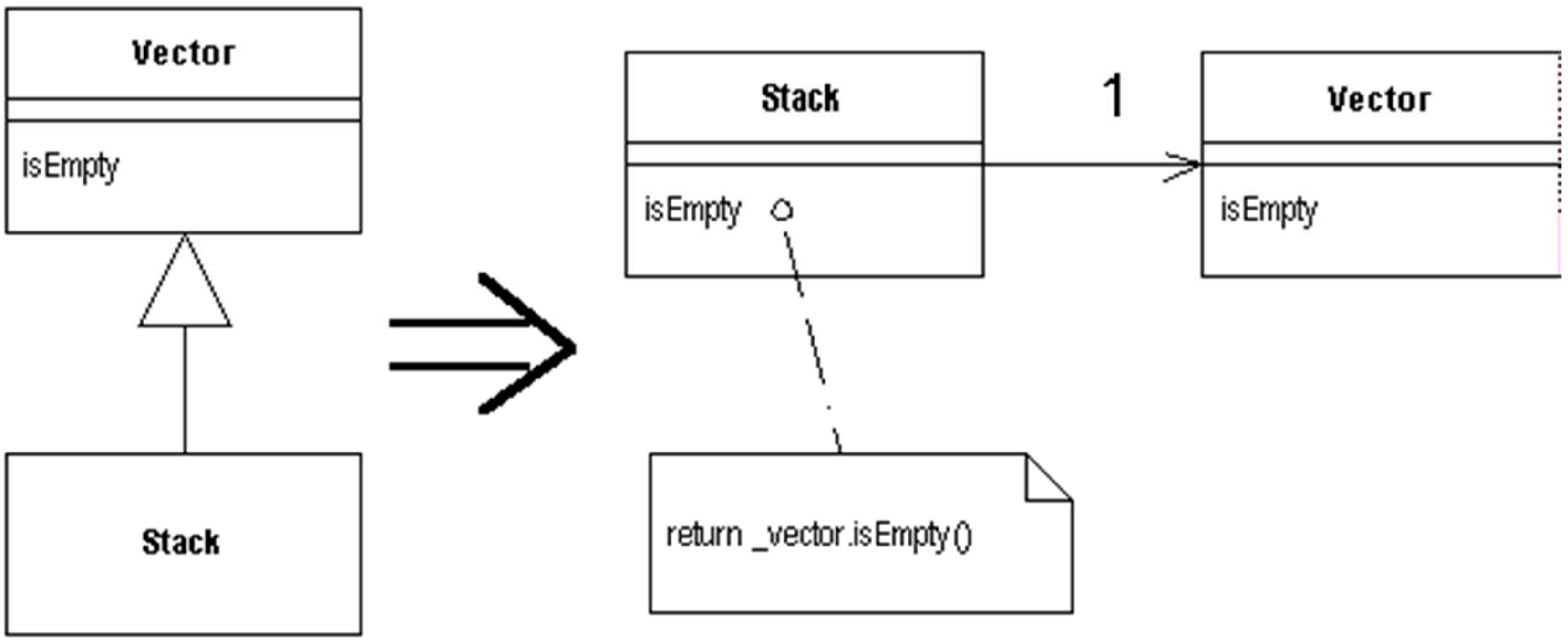
Refactorings
Introduce Null Object

```
if (customer == null) {
    plan = BillingPlan.basic();
} else {
    plan = customer.getPlan();
}
```

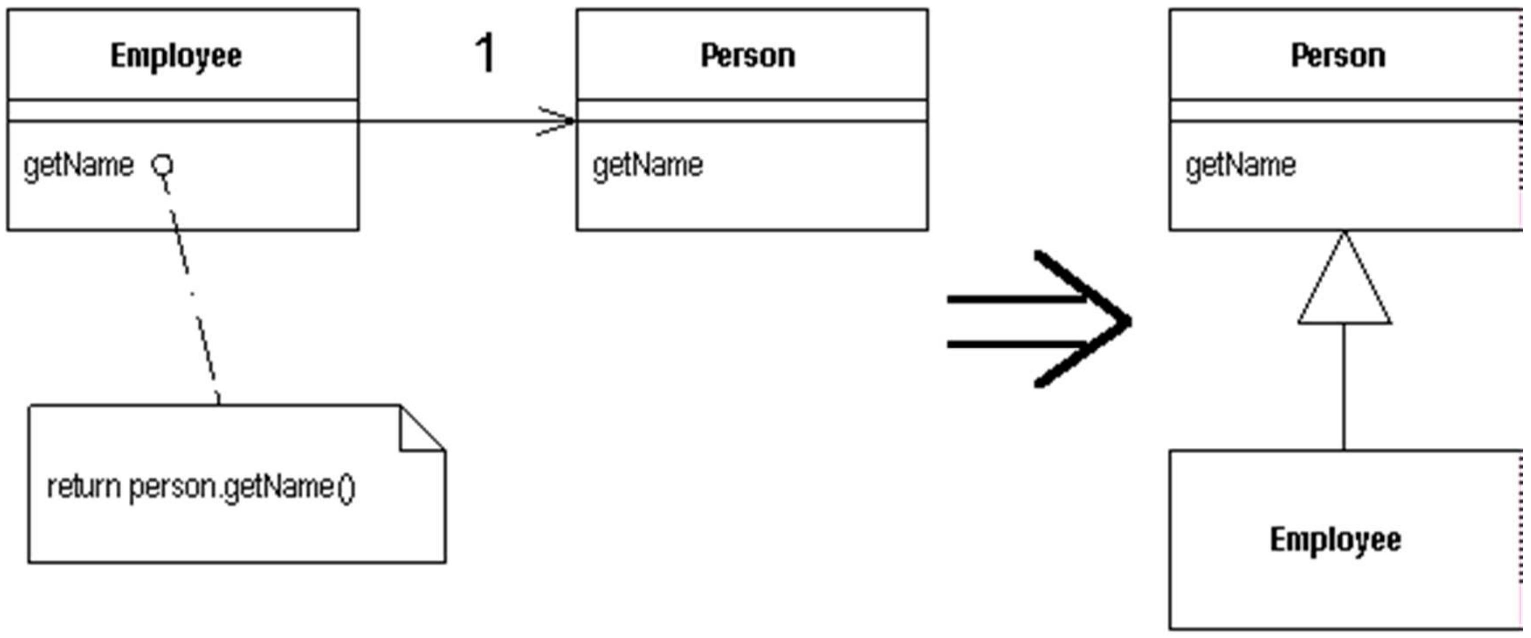
$\Rightarrow$



Refactorings
Replace Inheritance with Delegation



Refactorings
Replace Delegation with Inheritance



Readings

- http://www.refactoring.com/catalog/index.html
- Smells To Refactorings
  - http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Group assignments

- Make a coding standard document and commit to your group's project.
- Find a sample code for **each** bad smells introduced
    - In the language that you are using in your group project.
  - Refactor them
  - How to do it
    - Create a chore in your group's pivotaltracker, maybe with many tasks
    - Commit the code in to your project's github,
      - in a separated branch named refactor.
- Refactor your code.