Code Refactoring

November 8, 2022

Byung-Gon Chun

(Slide credits: George Candea, EPFL and Armando Fox, UCB)

What Makes Code “Legacy” and How Can Agile Help?

Legacy Code Matters

- Since maintenance consumes ~60% of software costs, *it is probably the most important life cycle phase of software . . .*

"Old hardware becomes obsolete; old software goes into production every night."

Robert Glass, *Facts & Fallacies of Software Engineering (fact #41)*

*How do we understand and **safely** modify legacy code?*

Maintenance != bug fixes

- Enhancements: 60% of maintenance costs
- Bug fixes: 17% of maintenance costs

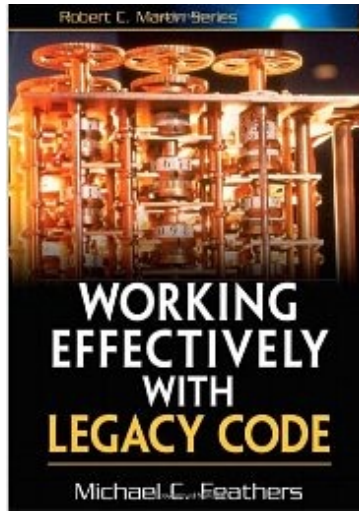
Hence the “60/60 rule”:

- 60% of software cost is maintenance
- 60% of maintenance cost is enhancements.

Glass, R. *Software Conflict.* Englewood Cliffs, NJ: Yourdon Press, 1991

What makes code “legacy”?

- Still meets customer need, **AND:**
- You didn’t write it, and it’s poorly documented
- You did write it, but a long time ago (and it’s poorly documented)
- *It lacks good tests (regardless of who wrote it)*—Feathers 2004



2 ways to think about modifying legacy code

- Edit & Pray
  - "I kind of think I probably didn't break anything"

- Cover & Modify
  - Let *test coverage* be your safety blanket

How Agile Can Help

1. **Exploration:** determine where you need to make changes (*change points)*
2. **Refactoring**: is the code around change points (a) tested? (b) testable?
   - (a) is true: good to go
   - !(a) && (b): apply BDD+TDD cycles to improve test coverage
   - !(a) && !(b): **refactor**

How Agile Can Help, cont.

3. Add tests to **improve coverage** as needed
4. **Make changes**, using tests as *ground truth*
5. **Refactor** further, to leave codebase better than you found it

- This is “embracing change” on long time scales

Approaching & Exploring a Legacy Codebase

Get the code running in development

- Check out a *scratch branch* that won't be checked back in, and get it to run
  - In a production-like setting or development-like setting
  - Ideally with something resembling a **copy** of production database
  - Some systems may be too large to clone
- Learn the user stories: Get customer to talk to you through what they're doing

Understand database schema & important classes

- Inspect database schema
- Create a model interaction diagram (UML class diagram) automatically or manually by code inspection
- What are the main (highly-connected) *classes,* their *responsibilities,* and their *collaborators?*

Codebase & “informal” docs

- Overall codebase *gestalt*
  - Subjective code quality?
  - Code to test ratio? Codebase size?
  - Major models/views/controllers?
  - Tests
- Informal design docs
  - Lo-fi UI mockups and user stories
  - Archived email, newsgroup, internal wiki pages or blog posts, etc. about the project
  - Design review notes
  - Commit logs in version control system (`git log`)
  - *Doc documentation

Summary: Exploration

- “Size up” the overall code base
- Identify key classes and relationships
- Identify most important data structures
  - “If you’ve chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.” – Rob Pike
- Ideally, identify place(s) where change(s) will be needed
- Keep design docs as you go
  - diagrams
  - GitHub wiki
  - comments you insert using *Doc (embedded documentation)

Qualitative: Code Smells

SOFA captures symptoms that often indicate code smells if code violates:

- Be short
- Do one thing
- Have few arguments
- Consistent level of abstraction

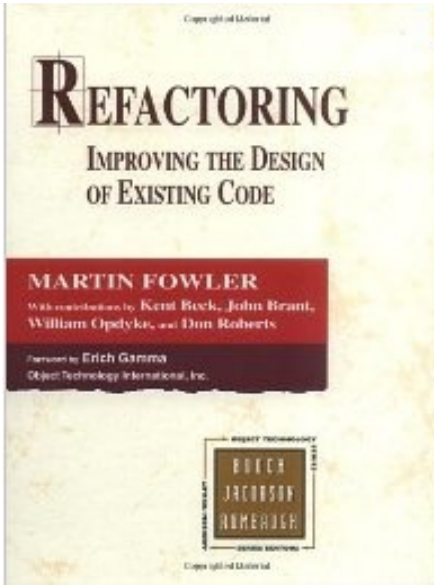
(Martin's *Clean Code*)

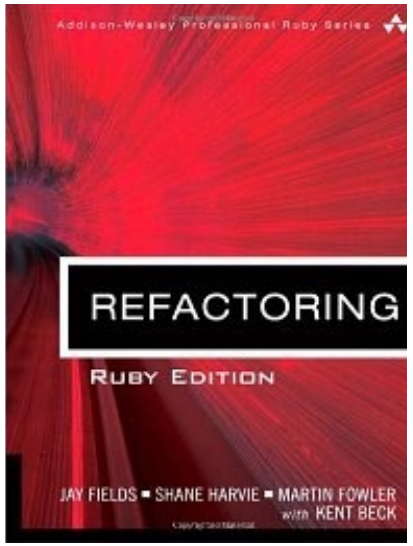
Refactoring: Idea

- Start with code that has 1 or more problems/smells
- Through a series of *small steps,* transform to code from which those smells are absent
- Protect each step with tests
- *Minimize time during which tests are red*

History & Context

- Fowler et al. developed mostly definitive catalog of refactorings
  - Adapted to various languages
  - Method- and class-level refactorings
- Each refactoring consists of:
  - Name
  - Summary of what it does/when to use
  - Motivation (what problem it solves)
  - Mechanics: step-by-step recipe
  - Example(s)





Composing Methods

Code Smell: Duplicated Code

- Same or very similar code repeated multiple times
  - In same method, same class, or anywhere in your system
- Hard to maintain
  - Changes in one version must be applied to all versions
  - Slight differences are hard to see

Extract Method

- Code fragment that can be grouped together
- Turn fragment into a method whose name explains its purpose
  - Method name serves as documentation
- Improves clarity
  - Encapsulates functionality by name
  - Shortens methods
  - Sequence of method calls reads almost like pseudo-code

Extract Method

- Move code into separate method
  - Variables not used anywhere else can be moved as well
- Variables that are used but not modified become parameters
  - Use “Replace Temp with Query” to reduce dependencies
- A single modified variable becomes the return value
- Multiple modified variables prevent simple extraction
  - Extract smaller or larger piece of code
  - Apply “Replace Method with Method Object”

```
public class Customer {
   private String name;
   private Map<Integer, Order> orderBook;

   private void printOwing() {
      Collection<Order> orders = orderBook.values();
      double outstanding = 0.0;

      // print banner
      System.out.println("**********************");
      System.out.println("** Customer Owes  **");
      System.out.println("**********************");

      // calculate outstanding
      for (Order order : orders) {
         outstanding += order.getAmount();
      }

      // print details
      System.out.println("name: " + name);
      System.out.println("amount: " + outstanding);
   }
}
```

```
public class Customer {
   private String name;
   private Map<Integer, Order> orderBook;

   private void printOwing() {
      Collection<Order> orders = orderBook.values();
      double outstanding = 0.0;
      printBanner();

      // calculate outstanding
      for (Order order : orders) {
         outstanding += order.getAmount();
      }

      // print details
      System.out.println("name: " + name);
      System.out.println("amount: " + outstanding);
   }
}
```

```
private void printBanner() {
   System.out.println("********************");
   System.out.println("** Customer Owes  **");
   System.out.println("********************");
}
```

```
public class Customer {
   private String name;
   private Map<Integer, Order> orderBook;

   private void printOwing() {
      Collection<Order> orders = orderBook.values();
      double outstanding = 0.0;

      printBanner();

      // calculate outstanding
      for (Order order : orders) {
         outstanding += order.getAmount();
      }

      printDetails(outstanding);
   }

   private void printDetails(double outstanding) {
      System.out.println("name: " + name);
      System.out.println("amount: " + outstanding);
   }
```

```
public class Customer {
   private String name;
   private Map<Integer, Order> orderBook;

   private void printOwing() {
      printBanner();
      double outstanding = getOutstanding();
      printDetails(outstanding);
   }

   private double getOutstanding() {
      Collection<Order> orders = orderBook.values();
      double outstanding = 0.0;

      for (Order order : orders) {
            outstanding += order.getAmount();
      }

      return outstanding;
   }
```

Inline Method

- A method’s body is just as clear as its name
- Put method body into callers and remove the method
- Use cases
  - Documentation through method name is not needed
  - Usually after other refactorings that simplified the body
  - Method just delegates to another one
  - Inline dependent methods before extracting them again in different granularity

Inline Method

```
int numberOfLateDeliveries;

private int getRating() {
   return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

private boolean moreThanFiveLateDeliveries() {
   return numberOfLateDeliveries > 5;
}
```

Inline Method

```
int numberOfLateDeliveries;

private int getRating() {
   return numberOfLateDeliveries > 5 ? 2 : 1;
}

private boolean moreThanFiveLateDeliveries() {
   return numberOfLateDeliveries > 5;
}
```

Replace Temp with Query

- A temporary variable holds the result of an expression
- Extract expression into a method and replace all references
  - Allow expression to be used in other methods
- Use cases
  - As preparation for Extract Method, to reduce use of temps
  - Reuse expression in other methods

Replace Temp with Query

```
private double totalPrice() {
   double basePrice = quantity * itemPrice;
   if (basePrice > 1000) {
      return basePrice * 0.95;
   } else {
      return basePrice * 0.98;
   }
}
```

Replace Temp with Query

```
private double totalPrice() {

   if (basePrice() > 1000) {
      return basePrice() * 0.95;
   } else {
      return basePrice() * 0.98;
   }
}

private double basePrice() {
   return quantity * itemPrice;
}
```

- Can hurt performance
  - If in a critical loop
  - Can be optimized by compiler
- Prepares for Extract Method
  - Less useful on its own

Substitute Algorithm

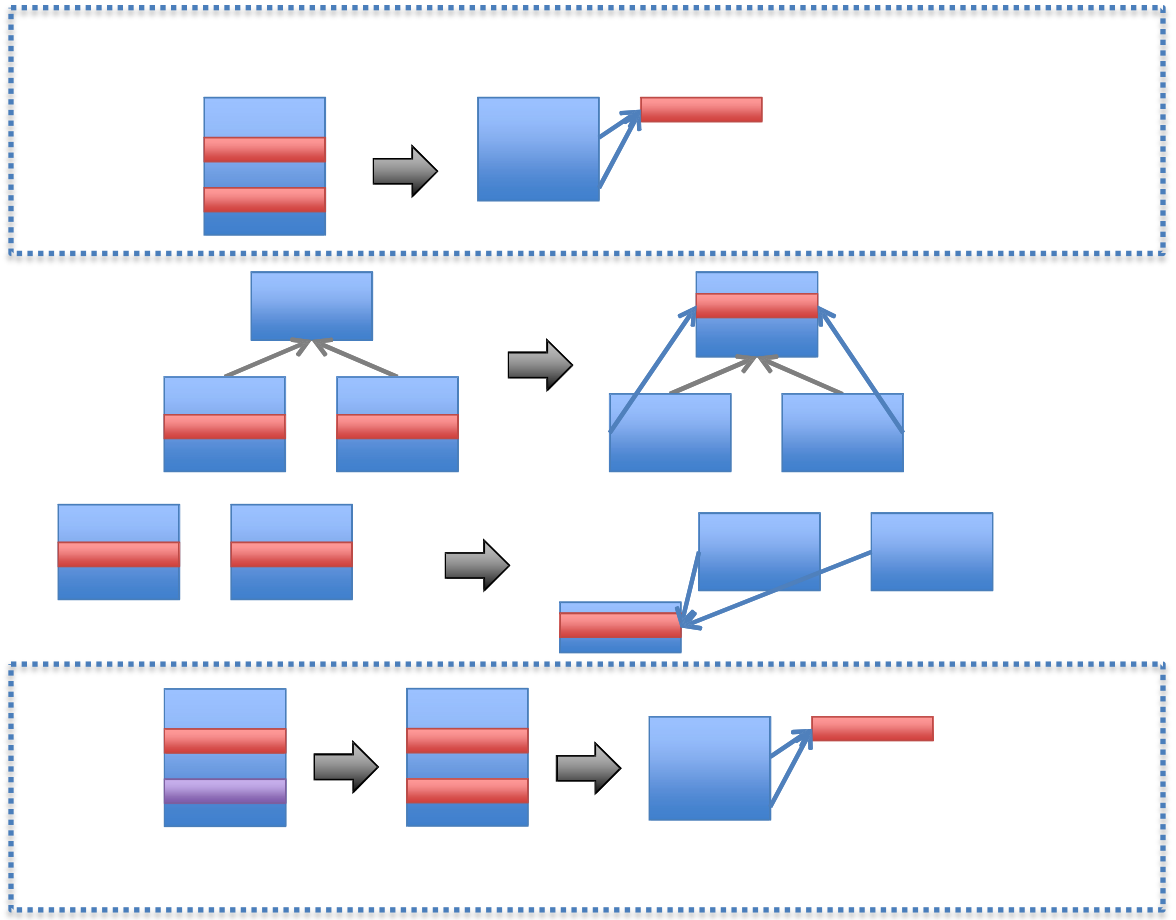
- Replace an algorithm with one that is clearer
- Run a new algorithm against tests
- Use an old algorithm as baseline, identify and debug divergent test cases

```
public String foundPerson(String[] people) {
   for (int i = 0; i < people.length; i++) {
      if (people[i].equals("Don")) {
         return "Don";
      }
      if (people[i].equals("John")) {
         return "John";
      }
      if (people[i].equals("Kent")) {
         return "Kent";
      }
   }
   return "";
}
```

```
public String foundPerson(String[] people){
   List<String> candidates = Arrays.asList("Don", "John", "Kent");
   for (int i = 0; i < people.length; i++)
      if (candidates.contains(people[i]))
         return people[i];
   return "";
}
```

Refactoring Duplicated Code

- In same class or method
  - Extract Method
- In sibling classes
  - Extract Method + Pull up Method
- In different classes
  - Extract classes
- Different algorithm
  - Substitute Algorithm + Extract Method



Summary

- Duplicated Code is one of the worst code smells
  - Not a bug, but a very bad sign
- Extract Method
  - Key refactoring for simplifying code
- Inline Method, Replace Temp with Query
  - Preparatory steps for Extract Method
- Substitute Algorithm
  - Replaces code with simpler, equivalent version

Moving Features

Code Smell: Feature Envy

- Code uses methods and fields of a different class more than those of its own
- Move code to where it wants to be
  - Move Method
- Decompose mixed code first
  - Extract Method

Code Smell: Shotgun Surgery

- Making a change requires many small changes in many different classes
- Bring functionality and data closer together
  - Move Field, Move Method
  - Inline class
- Create suitable abstraction for grouping affected code
  - Extract Class

Move Method

- Method uses more features of another class than of its own
  - Or is used more in another class
- Move method to other class
  - Must suit abstraction of that class
- Benefits
  - Improves maintainability: changes become more localized
  - Reduces coupling
  - Improves readability: more related code is within the same class

Move Method - Mechanics

- Identify related methods and variables of the original class
  - Also move these to the target class, or
  - Use a reference to source object in an instance variable or parameter, or
  - Pass required variables as parameters
- Find existing reference to target object in source or create one
- Copy method in target class
  - Make original method delegate calls to target object, or
  - Directly replace all calls by calls to method in target object

```
public class Account {
  private AccountType type;
  private int daysOverdrawn;

  double overdraftCharge() {
    if (type.isPremium()) {
      double result = 10;
      if (daysOverdrawn > 7) {
        result += (daysOverdrawn - 7) * 0.85;
      }
      return result;
    } else {
      return daysOverdrawn * 1.75;
    }
  }

  double bankCharge() {
    double result = 4.5;
    if (daysOverdrawn > 0) {
      result += overdraftCharge();
    }
    return result;
  }
}
```

Move Method

```
public class AccountType {

   double overdraftCharge() {

      if (type.isPremium()) {
         double result = 10;
         if (daysOverdrawn > 7) {
            result += (daysOverdrawn - 7) * 0.85;
         }
         return result;
      } else {
         return daysOverdrawn * 1.75;
      }
   }
}
```

dangling references

```
public class AccountType {

   double overdraftCharge(int daysOverdrawn) {
      if (isPremium()) {
         double result = 10;
         if (daysOverdrawn > 7) {
            result += (daysOverdrawn - 7) * 0.85;
         }
         return result;
      } else {
         return daysOverdrawn * 1.75;
      }
   }
}
```

supply required variable as parameter

or

```
double overdraftCharge(Account account) {
   if (isPremium()) {
      double result = 10;
       if (account.getDaysOverdrawn() > 7) {
          result += (account.getDaysOverdrawn() - 7) * 0.85;
       }
       ...
```

… or pass the entire object

```
public class Account {
   private AccountType type;
   private int daysOverdrawn;

   double overdraftCharge() {
      return type.overdraftCharge(daysOverdrawn);
   }

   double bankCharge() {
      double result = 4.5;
      if (daysOverdrawn > 0) {
         result += overdraftCharge();
      }
      return result;
   }
}
```

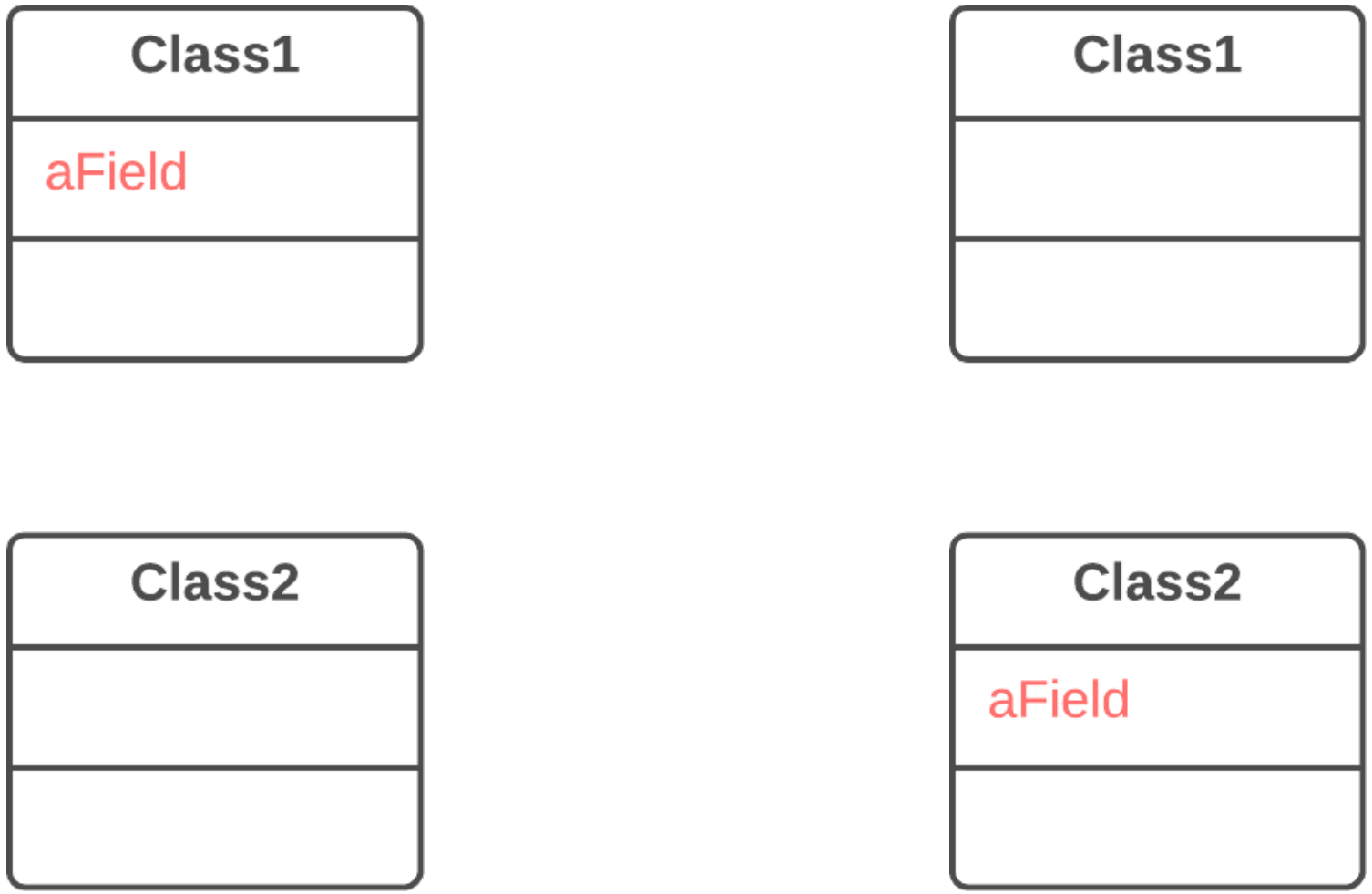
```
public class Account {
   private AccountType type;
   private int daysOverdrawn;



   double bankCharge() {
      double result = 4.5;
      if (daysOverdrawn > 0) {
         result += type.overdraftCharge(daysOverdrawn);
      }
      return result;
   }
}
```

Applied “Inline Method”

Move Field

- Field used more by methods of another class
- Move field to other class
  - Must suit the abstraction of that class
- Use cases
  - Can improve encapsulation – eliminate write accesses from the outside
  - Reduce coupling
  - Preparation for Extract Class, Move Method

Move Field



Move Field - Mechanics

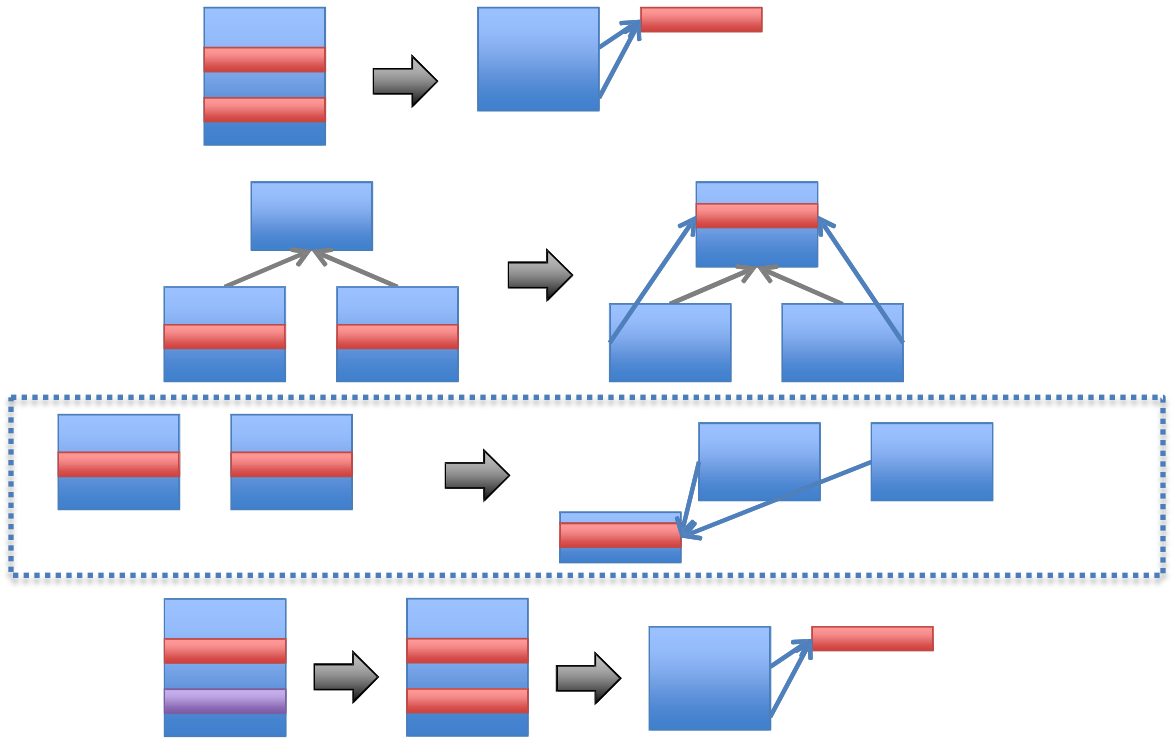
- Encapsulate field using getters / setters
- Create and encapsulate field in target class
- Find existing reference to target object in source or create one
- Remove field in source and replace all references with calls

Extract Class

- A class contains too much functionality
  - Clusters of methods and data that go together
- Create a new class and move fields and methods
- Benefits
  - Improve readability
  - Encapsulate functionality
  - Allow more targeted inheritance

Refactoring Duplicated Code

- In same class or method
  - Extract Method
- In sibling classes
  - Extract Method + Pull up Method
- In different classes
  - Extract classes
- Different algorithm
  - Substitute Algorithm + Extract Method



Extract Class - Mechanics

- Split functionality into two groups
- Create a new class with suitable abstraction
- For each method and field that needs to move
  - *apply Move Field or Move Method*
  - *compile and test*
- Reduce the interface
  - eliminate two-way referencing

```
public class Person {
   private String name;
   private String officeAreaCode;
   private String officeNumber;

   public String getName() {
      return name;
   }
   public String getTelephoneNumber() {
      return ("(" + officeAreaCode + ") " + officeNumber);
   }
   String getOfficeAreaCode() {
      return officeAreaCode;
   }
   void setOfficeAreaCode(String arg) {
      officeAreaCode = arg;
   }
   String getOfficeNumber() {
      return officeNumber;
   }
   void setOfficeNumber(String arg) {
      officeNumber = arg;
   }
}
```

```
public class Person {
   private String name;
   private TelephoneNumber officeTelephone;

   public String getName() {
      return name;
   }
   public String getTelephoneNumber() {
      return officeTelephone.getTelephoneNumber();
   }
   TelephoneNumber getOfficeTelephone() {
      return officeTelephone;
   }
}

public  class  TelephoneNumber  {
   private  String  officeAreaCode;
   private String officeNumber;

   public String getTelephoneNumber() {
      return ("(" + officeAreaCode + ") " + officeNumber);
   }
   String getOfficeAreaCode() {
      return officeAreaCode;
   }
```

one-way link

delegate to target object

Inline Class

- A class that has almost no functionality is not necessary
- Move remaining features to class that uses it most and delete it
- Use cases
  - other refactorings removed almost all functionality from the class
  - class only delegates, without actual purpose

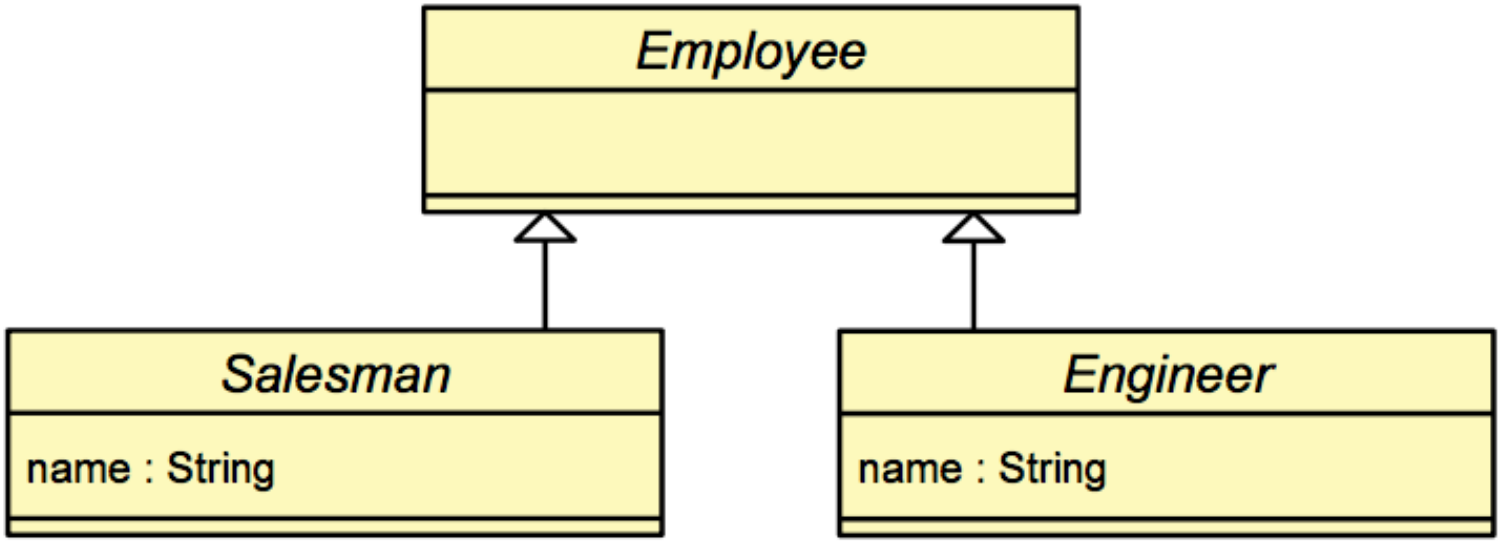
Summary

- Feature Envy
  - code mostly uses methods / variables of another class
  - Extract Method, Move Method / Field
- Shotgun Surgery
  - applying changes entails little changes in many places
  - Move Method / Field, Extract / Inline Class
- Move Method, Move Field
  - move code where it belongs
  - improve abstractions
- Extract Class
  - split responsibilities
  - simplify code through abstraction
- Inline Class
  - eliminate useless classes

Refactoring Inheritance

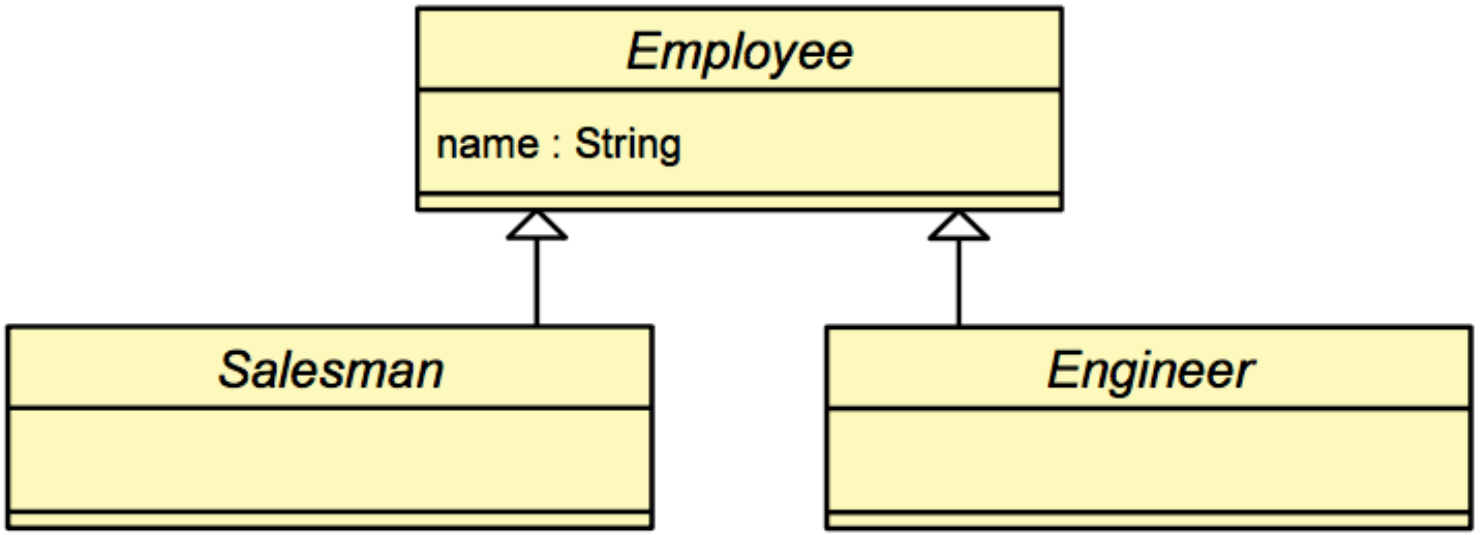
Pull Up Field

- Subclasses have the same field, which is used in the same way
- Move field to superclass
- Make field protected or (preferred) encapsulate it



Pull Up Field

- Subclasses have the same field, which is used in the same way
- Move field to superclass
- Make field protected or (preferred) encapsulate it

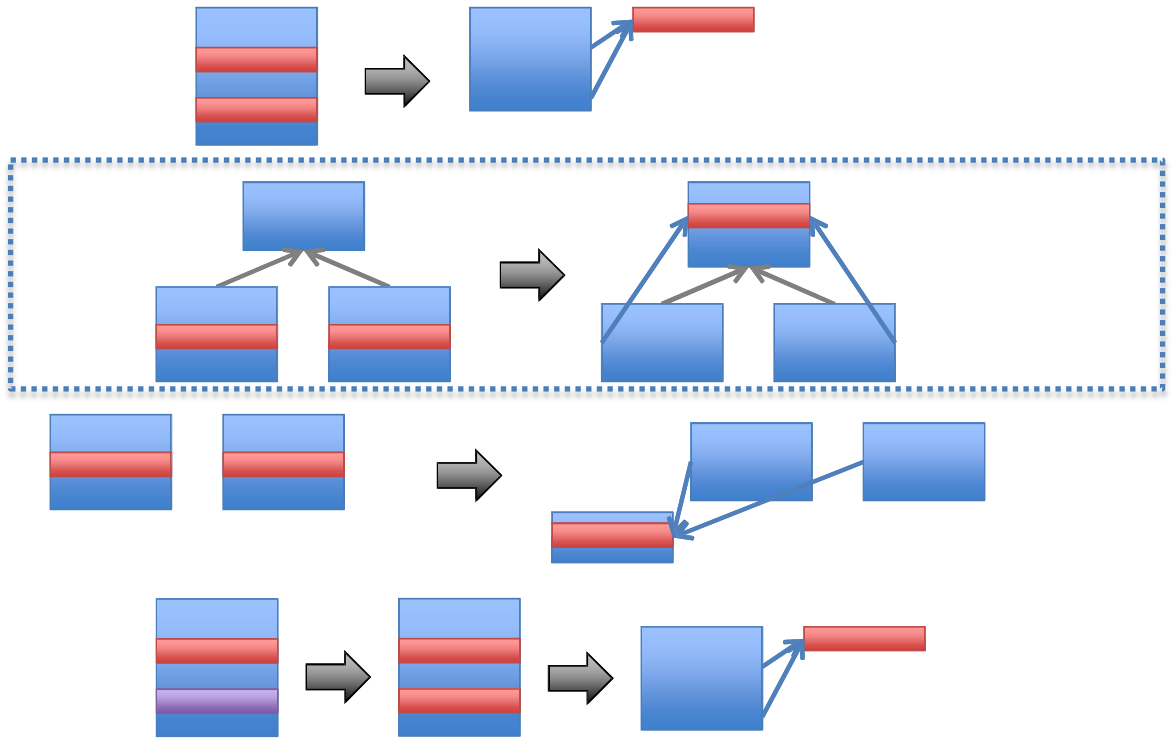


Pull Up Method

- Two subclasses have methods with identical behavior
  - Duplicate code, Substitute Algorithm if necessary
- Move methods to superclass
  - similar to Move Method
- Leverage inheritance
  - can’t use explicit references as in Move Method
  - use features of subclass through abstract methods (Template Method)

Refactoring Duplicated Code

- In same class or method
  - Extract Method
- In sibling classes
  - Extract Method + Pull up Method
- In different classes
  - Extract classes
- Different algorithm
  - Substitute Algorithm + Extract Method

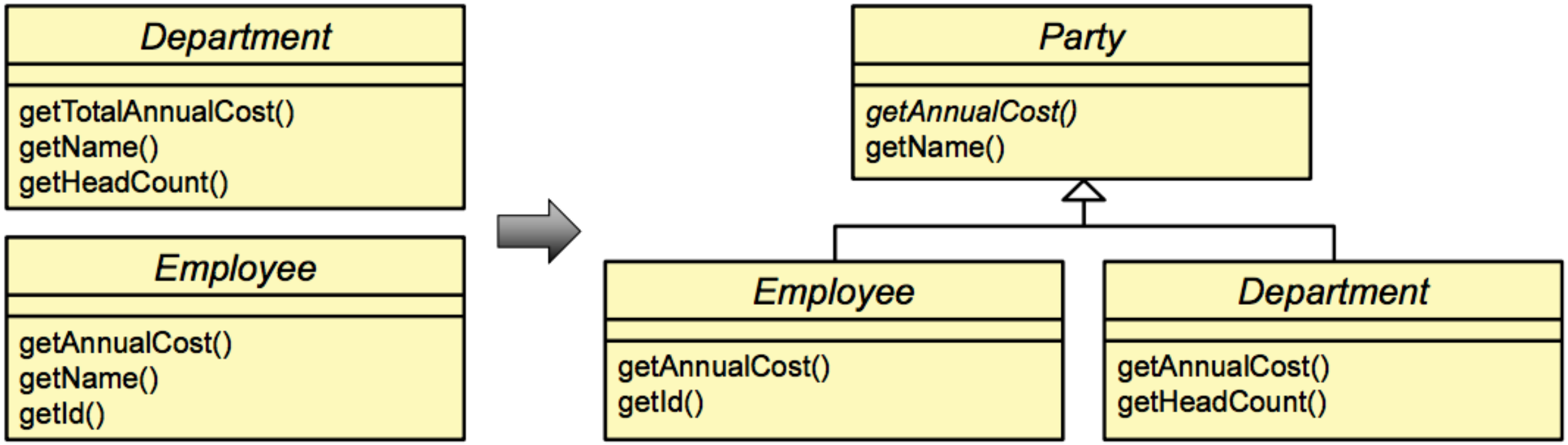


Push Down Field / Method

- Field or method is really specific to a subclass
- Move field to subclass
- Mechanics
  - copy field / method to subclasses, remove from superclass
  - compile & test
  - erase from subclasses that don’t use the field / method
  - if used by multiple subclasses, consider additional level of hierarchy (Extract Superclass / Subclass)

Extract Superclass

- Two classes share similar features
- Create superclass and pull up common features



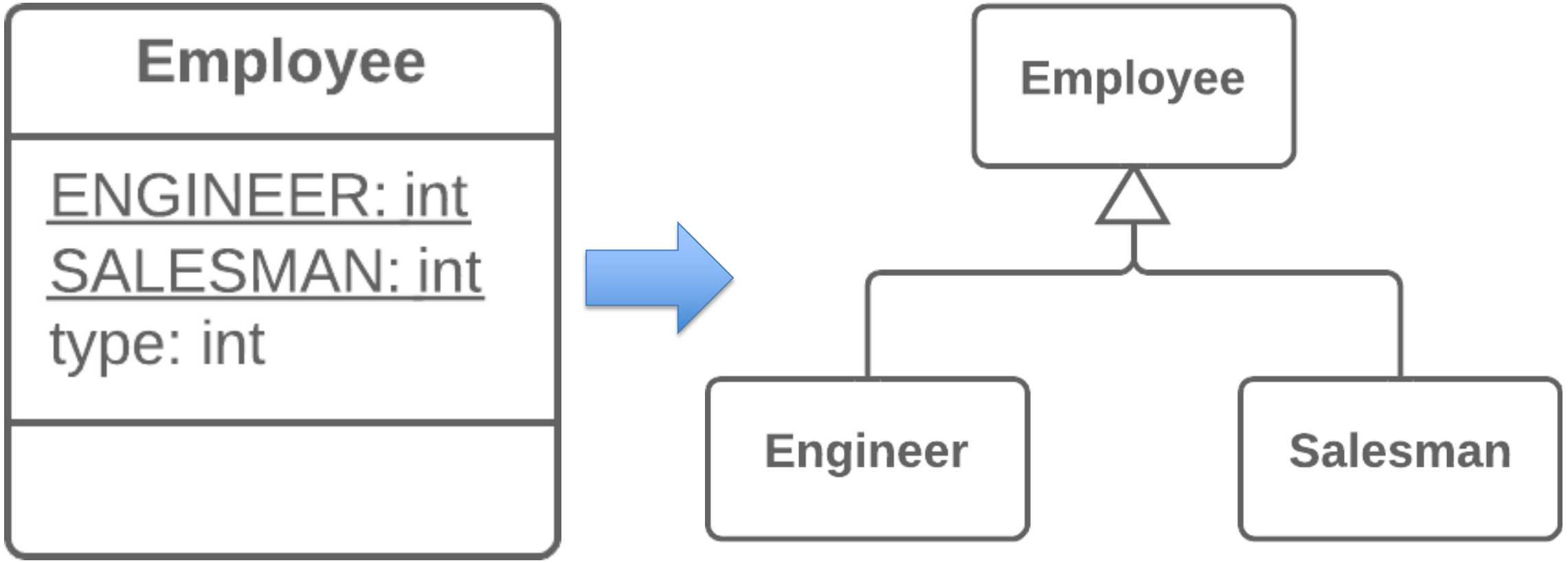
Extract Subclass

- Some features of a class are used only by some instances
- Create a subclass for the feature subset
- Eliminates type codes
  - instance variables that encode some type information: ints, Booleans, ...
  - can be set to appropriate constants in subclasses

```
class AccountType {
   boolean isPremium;
   ...
}
```

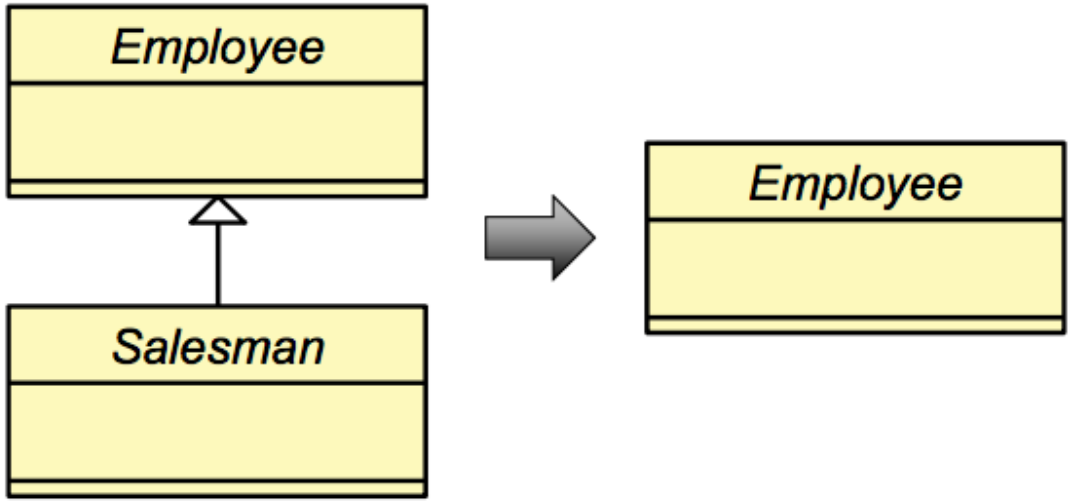
```
class GoldAccount extends AccountType {
   boolean isPremium = true;
   ...
}
```

Replace Type Code with Subclasses



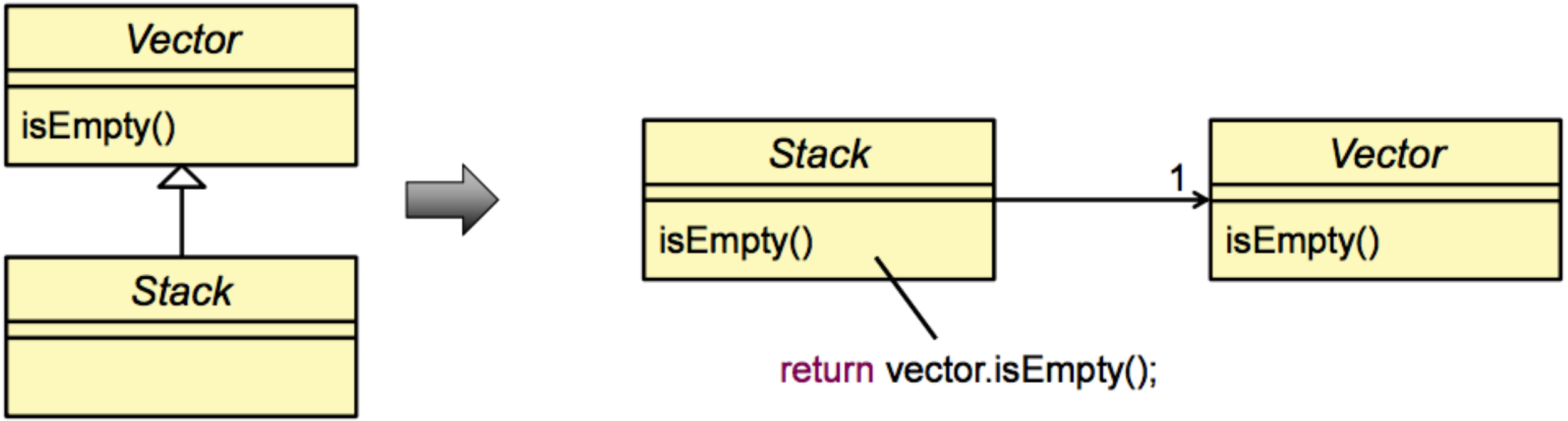
Collapse Hierarchy

- A superclass and its subclass are very similar
- Merge into one, adapt references
- Use cases
  - Refactoring eliminated difference between hierarchy levels
  - Change in requirements eliminated siblings



Replace Inheritance with Delegation

- A subclass violates the “is a” relationship or the LSP
- Use superclass through instance field, remove subclassing



Summary

- Move methods / variables to appropriate level of abstraction
  - using Pull Up / Push Down Field / Method
- Refactor class hierarchy according to need
  - apply Extract Super- / Subclass, Collapse Hierarchy
- Consider containment instead of inheritance
  - use Extract Class instead of Extract Super- / Subclass
  - replace existing inheritance with delegation