

# REFACTORING

---

Yuqun Zhang

CS 304

# What is Refactoring?

- Semantic-preserving program transformations
  - A change made to the internal structure of a program without modifying its observable behavior to make it
    - Easier to understand
    - Cheaper to modify
- **Refactoring patterns**
  - “Improving the design after the code has been written”
    - Seems a bit odd since we usually design first then code
    - Refactoring usually entails small changes with large cumulative effects

# Why Refactor?

- Code degenerates under maintenance
- Code was poorly written to begin with

# Bad Code Smells and Refactoring

- **Code smells**

- Indicative of bad software design

- List of bad smells:

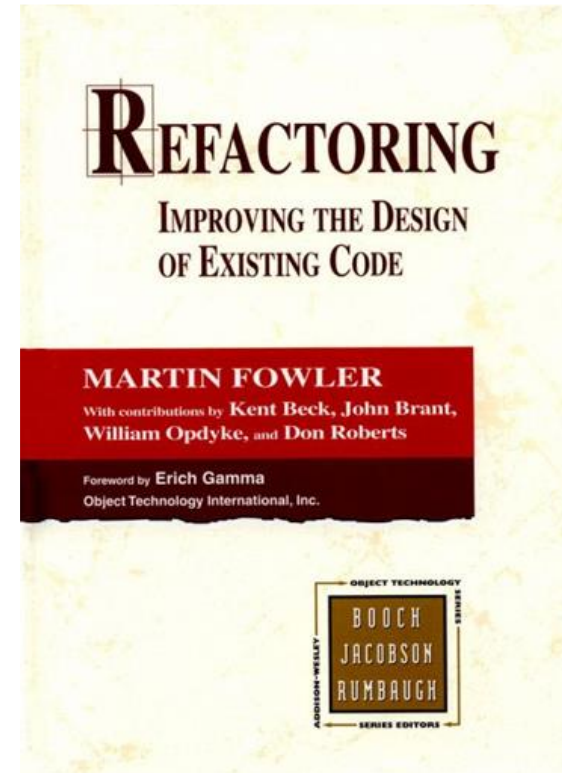
- <http://blog.codinghorror.com/code-smells/>

- Useful “catalog” of refactorings:

- <http://www.refactoring.com/catalog/>

- Mapping of smells to refactorings:

- <http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf>



# Refactorings

- The book is basically a catalog of common refactoring patterns
  - Each includes a name, summary, motivation, mechanics, and examples
- Not formal (they can't be, since determining program equivalence is undecidable)
- Similar in nature to design patterns
  - Defining a shared vocabulary

# EXAMPLES OF CODE SMELLS

---

And associated refactorings

# #1: Duplicated Code

- You've done this before
- You know it's bad
- Examples
  - Same expression in two methods in the same class
  - Same expression in two methods in sibling classes
  - Same expression in two unrelated classes
- Explicit and subtle duplication
  - E.g., identical code (explicit) vs. structures or processing steps that appear different but are essentially the same (subtle)
- Potential useful refactorings:
  - **Extract method**, **Extract class**, **Template method pattern**, **Strategy pattern**



# Extract Method

- Applies when you have a code fragment inside some code blocks where the lines of code should always be grouped together
- ✓ *Turn the fragment into a method whose name explains the purpose of the block of code*



# Extract Method Refactoring Example

```
void printOwing() {  
    printBanner();  
  
    //print details  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + getOutstanding());  
}
```

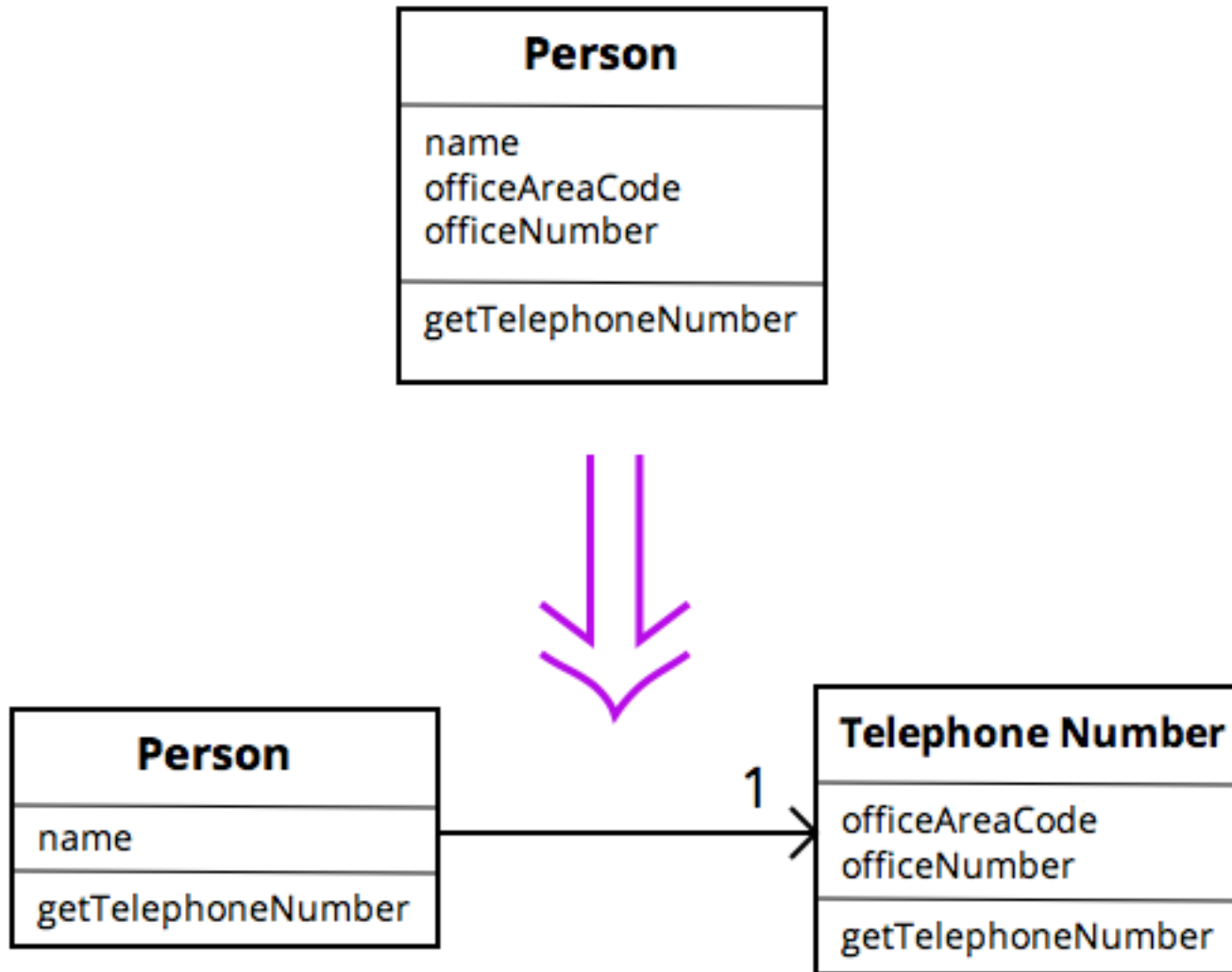


```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails (double outstanding) {  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + outstanding);  
}
```

# Extract Class

- You have one class doing work that should be done by two different classes
- ✓ *Create a new class and move the relevant fields and methods from the old class to the new class*

# Extract Class Example



# Some codes

- `class Person...`
- `private String name;`
- `private TelephoneNumber officeTelephone = new TelephoneNumber();`
- `public String getName() {`
- `return name;`
- `}`
- `public String getTelephoneNumber() {`
- `return officeTelephone.getTelephoneNumber();`
- `}`
-

## Some codes (continued.)

- `class TelephoneNumber...`
- `private String number;`
- `private String areacode;`
- `public String getTelephoneNumber() {`
- `return "(" + areacode + ") " +`
- `number);`
- `}`
- 
-

# Template Method Pattern

- A Template Method describes the skeleton behavior of a method
  - Defers some substeps to subclasses
- By defining the “primitive operations” comprising the template method, the subclasses provide different behaviors



# Style Smells

- Comments
  - There's a fine line between comments that illuminate and comments that obscure
  - Make sure comments are actually needed; if possible, refactor the code so that the comments aren't required
    - Copious comments can be indicative of bad code
- Naming
  - Avoid placing types in method or variable names (because if you change the type, you'll have to change the name)
  - Make sure that the names of methods and variables succinctly describe what the purpose is
  - Pick a standard way of naming things and stick with it; make sure that analog functions have analog names (e.g., if you can `open()` you ought to be able to `close()`)
- Dead code
  - Delete it. Use version control.



# Long Method

- Two long methods are more likely to share duplicated code/logic
- Small methods help explain code
  - If you don't understand a long method, breaking it into smaller, well named methods helps readability
- Systems with smaller methods tend to be easier to extend and maintain
- Summary: all other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot
- Potential useful refactorings:
  - **Extract method** (vast majority of the time)





# Large Class

- This often happens when we code before careful design or prototype a design and then keep building it
- Too many instance variables
  - A class is trying to do too much
  - The class has too many responsibilities
- **Single Responsibility Principle**: one class should be responsible for only one functional purpose
- Potential refactorings
  - **Extract class**, **Extract subclass**
    - Hint: look for common prefixes/suffixes in identifiers
  - **Observer**
    - Common for GUIs



# Long Parameter List

- Long lists of parameters (common in procedural programming) are likely to be **volatile**
  - i.e., likely to change often and rapidly
- Consider which parameters are essential
  - Leave the rest to the object to track down as necessary
- Potential refactorings:
  - **Replace parameter with method**, **Introduce parameter object**, **Preserve whole object**



# Replace Parameter with Method

- An object invokes a method then passes the result as a parameter for a method
  - The receiver can also invoke this method
- ✓ *Why the indirection? Remove the parameter and let the receiver invoke the method.*

# Replace Parameter with Method Example

```
int basePrice = _quantity * _itemPrice;  
discountLevel = getDiscountLevel();  
double finalPrice = discountedPrice (basePrice, discountLevel);
```



```
int basePrice = _quantity * _itemPrice;  
double finalPrice = discountedPrice (basePrice);
```

# Some codes (originally)

- `public double getPrice(){`
- `int basePrice = quantity * itemPrice;`
- `int discountLevel;`
- `if (quantity > 100) discountLevel = 2;`
- `else discountLevel = 1;`
- `double finalPrice = discountedPrice (basePrice,`  
`discountLevel);`
- `return finalPrice;`
- `}`
  
- `private double discountedPrice (int basePrice, int`  
`discountLevel) {`
- `if (discountLevel == 2) return basePrice * 0.1;`
- `else return basePrice * 0.05;`
- `}`

# Some codes (originally)

- `public double getPrice(){`
- `int basePrice = quantity * itemPrice;`
- `int discountLevel;`
- `if (quantity > 100) discountLevel = 2;`
- `else discountLevel = 1;`
- `double finalPrice = discountedPrice (basePrice,`  
`discountLevel);`
- `return finalPrice;`
- `}`
  
- `private double discountedPrice (int basePrice, int`  
`discountLevel) {`
- `if (discountLevel == 2) return basePrice * 0.1;`
- `else return basePrice * 0.05;`
- `}`

# Some codes (Initial modification)

- `public double getPrice() {`
- `int basePrice = quantity * itemPrice;`
- `int discountLevel = getDiscountLevel();`
- `double finalPrice = discoutedPrice (basePrice,`  
    `discountLevel);`
- `return finalPrice;`
- `}`
  
- `private int getDiscountLevel () {`
- `if (quantity > 100) discountLevel = 2;`
- `else discountLevel = 1;`
- `}`

# Some codes (a little further)

- `private double discountedPrice (int basePrice, int discountLevel) {`
- `if (getDiscountLevel() == 2) return basePrice * 0.1;`
- `else return basePrice * 0.05;`
- `}`



# Some codes (Now what do we have)

- `public double getPrice() {`
- `int basePrice = quantity * itemPrice;`
- `int discountLevel = getDiscountLevel();`
- `double finalPrice = discountedPrice`  
 `(basePrice);`
- `return finalPrice;`
- `}`
- `private double discountedPrice (int basePrice)`  
`{`
- `if (getDiscountLevel() == 2) return`  
 `basePrice * 0.1;`
- `else return basePrice * 0.05;`
- `}`

# Some codes (Something is not necessary)

- `public double getPrice() {`
- `int basePrice = quantity * itemPrice;`
- `int discountLevel = getDiscountLevel();`
- `double finalPrice = discountedPrice`  
 `(basePrice);`
- `return finalPrice;`
- `}`
  
- `private double discountedPrice (int basePrice)`  
`{`
- `if (getDiscountLevel() == 2) return`  
 `basePrice * 0.1;`
- `else return basePrice * 0.05;`
- `}`

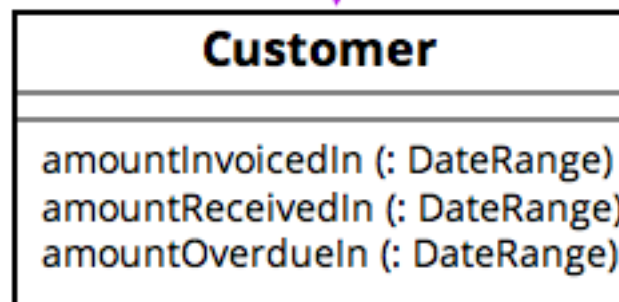
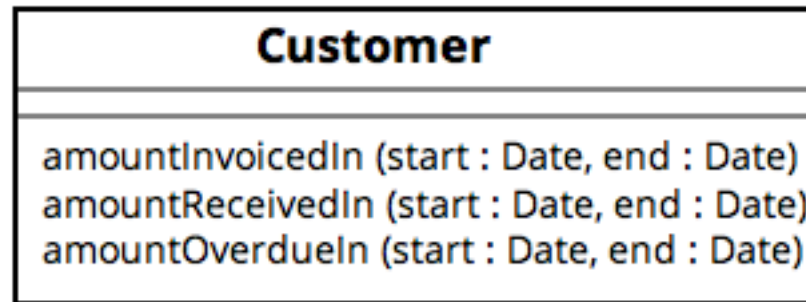
# Some codes (final version)

- `public double getPrice() {`
- `return discountedPrice();`
- `}`
  
- `private double discountedPrice () {`
- `if (getDiscountLevel() == 2) return`  
 `getBasePrice() * 0.1;`
- `else return getBasePrice() * 0.05;`
- `}`
  
- `private double getBasePrice() {`
- `return quantity * itemPrice;`
- `}`

# Introduce Parameter Object

- You have a group of parameters that naturally (often) go together
- ✓ *Replace them with a single object*

# Introduce Parameter Object Example



# Preserve Whole Object

- You get a bunch of values from an object but then pass those objects together to another method call
- ✓ *Maybe you should just pass the whole object instead.*

# Preserve Whole Object Example

```
int low = daysTempRange().getLow();  
int high = daysTempRange().getHigh();  
withinPlan = plan.withinRange(low, high);
```



```
withinPlan = plan.withinRange(daysTempRange());
```

# Divergent Change

- Commonly change a particular class in different ways for different reasons
  - Separating divergent responsibilities decreases the chance that one change negatively affects a different function
  - E.g., in `class X`, change `mA()`, `mB()`, and `mC()` every time we add a new database; change `mD()`, `mE()`, and `mF()` every time we add a new financial instrument
- Potential refactoring:
  - **Extract class**





# Shotgun Surgery

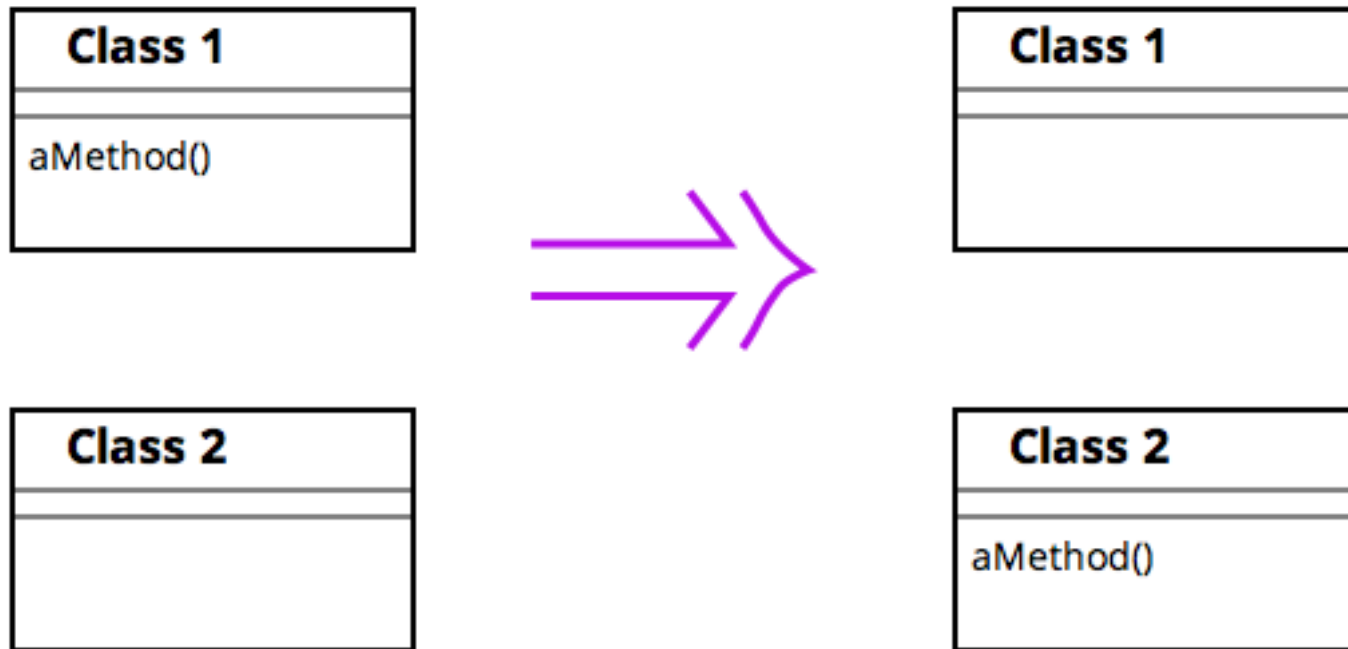
- Opposite of divergent change
- One change alters many classes; constantly making lots of little changes to a lot of different classes
  - It's easy to miss an important change
- Special case:
  - Parallel inheritance hierarchies – every time you make a subclass of one class, you have to make a subclass of another
- Potential refactorings:
  - **Move method**, **Move field**, **Inline class**



# Move Method

- A method is, or will be, using or used by more features of a class other than the class within which it is defined
- ✓ *Well, then, move it. Create a new method with a similar body in the class it uses most. Turn the old method into a simple delegation or remove it altogether.*

# Move Method Example



# Move Method (another) Example

```
class Project {
    Person[] participants;
}

class Person {
    int id;
    boolean participate(Project p) {
        for(int i=0; i<p.participants.length; i++) {
            if (p.participants[i].id == id) return(true);
        }
        return(false);
    }
}

... if (x.participate(p)) ...
```

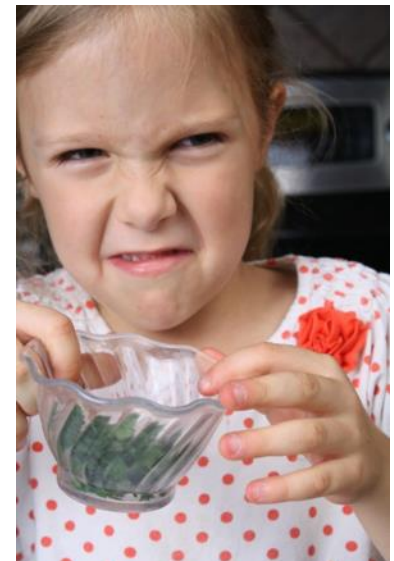
```
class Project {
    Person[] participants;
    boolean participate(Person x) {
        for(int i=0; i<participants.length; i++) {
            if (participants[i].id == x.id) return(true);
        }
        return(false);
    }
}

class Person {
    int id;
}

... if (p.participate(x)) ...
```

# Feature Envy

- A method in a class seems more interest in some other class's internals than its own
  - The most common target of the envy is data
  - E.g., a class repeatedly calls getter and setter methods on some other class
- [Strategy pattern is an exception]
- Potential refactorings:
  - **Extract method**, **Move method**, **Move field**



# Data Clumps

- Bunches of data that hang around together should be made into their own object
  - Fields in several classes, parameters that are always chained together, etc.
  - Ask yourself the question: are the others sensible when one is removed?
- Potential refactorings:
  - **Extract class**, **Preserve whole object**, **Introduce parameter object**



# Primitive Obsession

- Old-timers are reluctant to use small objects for money, strings, intervals, etc.
- Instead result in an over-emphasis on primitive objects (e.g., strings, arrays, integers, etc.)
- Classes generally provide a simpler and more natural way to directly model things than primitives do
  - Higher level abstractions clarify code
- Potential refactorings:
  - **Replace data value(s) with object**,  
**Replace type code with class**, **Replace type code with state/strategy**

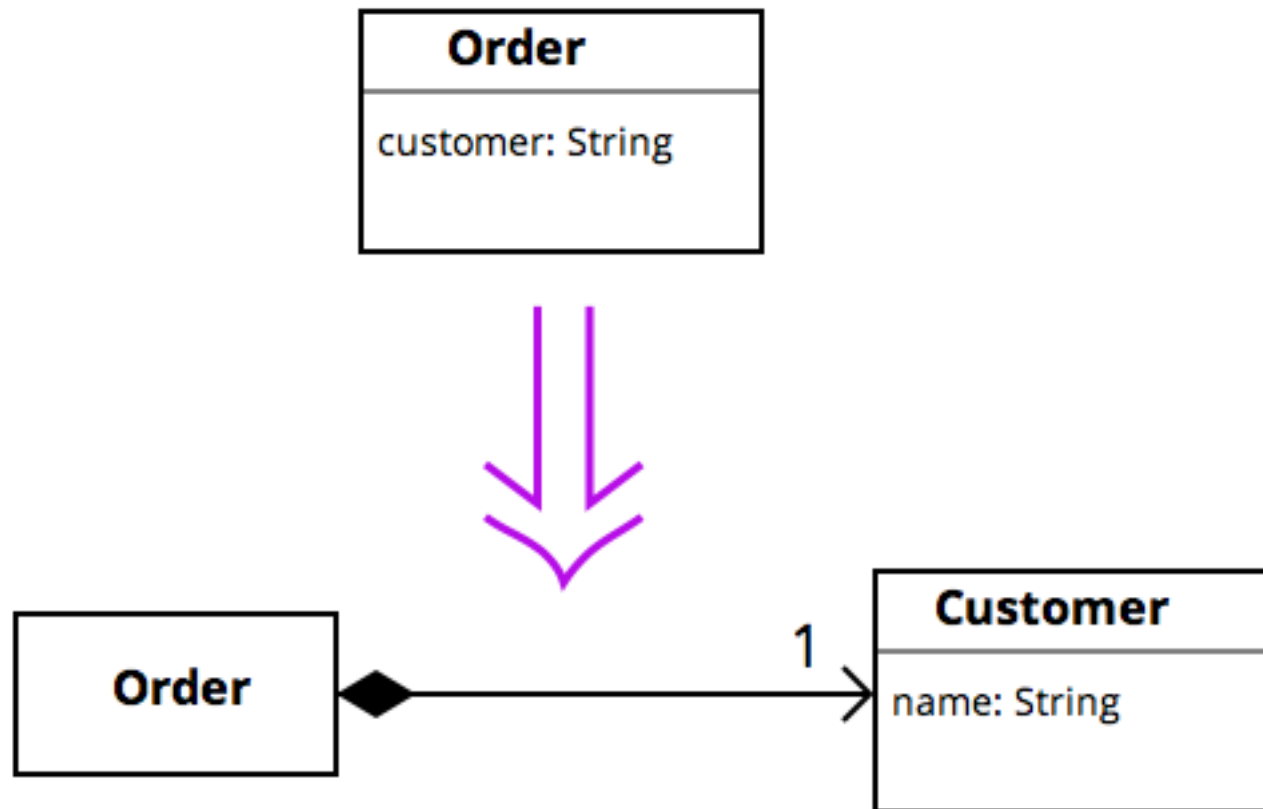


# Replace Data Value with Object

- You have a data item that needs additional data or behavior
  - Really, try not to start with primitives and add more and more primitives that are conceptually (but not concretely) linked
- ✓ *Instead, turn the data item into an object*



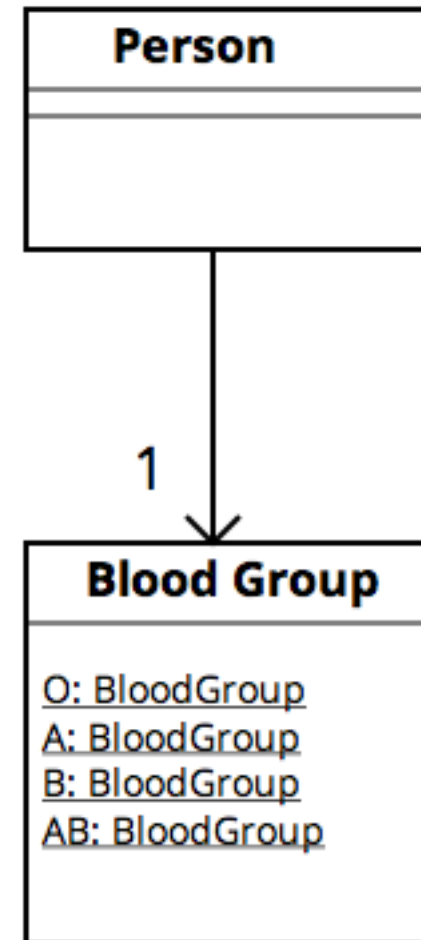
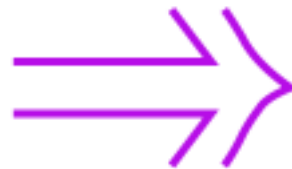
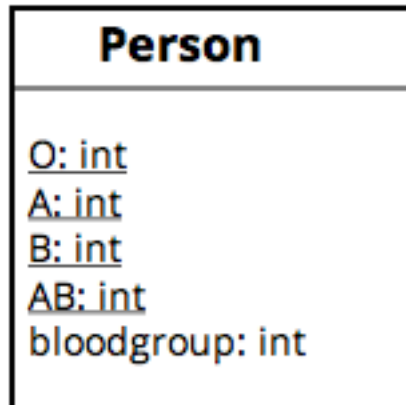
# Replace Data Value with Object Example



# Replace Type Code with Class

- A class has a (numeric) type code that does not affect its behavior
- ✓ *Replace the number with a new class*

# Replace Type Code with Class Example



# Replace Type with Class: Code Examples

```
class Person {  
    public static final int O = 0;  
    public static final int A = 1;  
    public static final int B = 2;  
    public static final int AB = 3;  
  
    private int _bloodGroup;  
  
    public Person (int bloodGroup) {  
        _bloodGroup = bloodGroup;  
    }  
  
    public void setBloodGroup(int arg) {  
        _bloodGroup = arg;  
    }  
  
    public int getBloodGroup() {  
        return _bloodGroup;  
    }  
}
```

## Replace Type with Class: Code Examples (cont.)

```
class BloodGroup {  
    public static final BloodGroup O = new BloodGroup(0);  
    public static final BloodGroup A = new BloodGroup(1);  
    public static final BloodGroup B = new BloodGroup(2);  
    public static final BloodGroup AB = new BloodGroup(3);  
    private static final BloodGroup[] _values = {O, A, B, AB};  
  
    private final int _code;  
  
    private BloodGroup (int code ) {  
        _code = code;  
    }  
  
    public int getCode() {  
        return _code;  
    }  
  
    public static BloodGroup code(int arg) {  
        return _values[arg];  
    }  
}
```

## Replace Type with Class: Code Examples (cont.)

```
class Person {  
    public static final int O = BloodGroup.O.getCode();  
    public static final int A = BloodGroup.A.getCode();  
    public static final int B = BloodGroup.B.getCode();  
    public static final int AB = BloodGroup.AB.getCode();  
  
    private BloodGroup _bloodGroup;  
  
    public Person (int bloodGroup) {  
        _bloodGroup = BloodGroup.code(bloodGroup);  
    }  
  
    public int getBloodGroup() {  
        return _bloodGroup.getCode();  
    }  
  
    public void setBloodGroup(int arg) {  
        _bloodGroup = BloodGroup.code (arg);  
    }  
}
```

# Replace Type with Class: Code Examples (cont.)

```
class Person {
    ...
}
```

```
public class Person {
    ...
}
```

```
public class Person (int bloodGroup) {
    ...
}
```

```
public class Person {
    ...
}
```

```
class Person {
    ...
}
```

```
public void setBloodGroup(int arg) {
    _bloodGroup = BloodGroup.code (arg);
}
```

```
public void setBloodGroup(BloodGroup arg) {
    _bloodGroup = arg;
}
```

## Replace Type with Class: Code Examples (cont.)

```
Person thePerson = new Person (Person.A);
```

```
class Person ...  
    public static final int O = BloodGroup.O.getCode();  
    public static final int A = BloodGroup.A.getCode();  
    public static final int B = BloodGroup.B.getCode();  
    public static final int AB = BloodGroup.AB.getCode();  
    public Person (int bloodGroup) {  
        _bloodGroup = BloodGroup.code(bloodGroup);  
    }  
    public int getBloodGroup() {  
        return _bloodGroup.getCode();  
    }  
    public void setBloodGroup(int arg) {  
        _bloodGroup = BloodGroup.code (arg);  
    }  
}
```



# Replace Type with Class: Code Examples (cont.)

- Now class “person” is like...
- `class person...`
- `public int getBloodGroupCode() {`
- `return _bloodGroup.getCode();`
- `}`
- `public BloodGroup getBloodGroup() {`
- `return _bloodGroup;`
- `}`
- `public Person (BloodGroup bloodGroup) {`
- `_bloodGroup = bloodGroup;`
- `}`
- `public void setBloodGroup(BloodGroup arg) {`
- `_bloodGroup = arg;`
- `}`

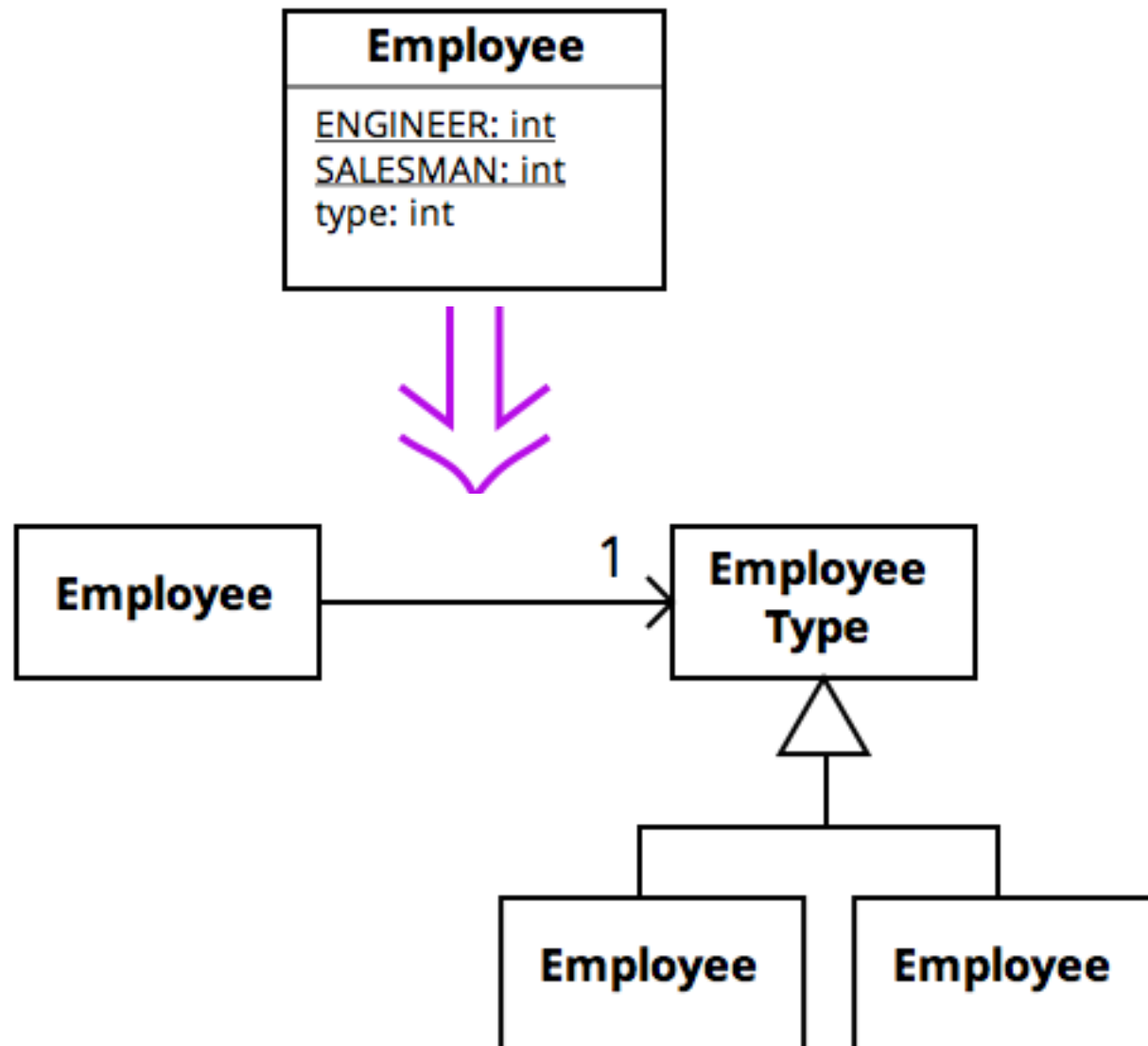
# Replace Type with Class: Code Examples (cont.)

- Any more improvements for class “BloodGroup”?

# Replace Type Code with State/Strategy

- You have a type code that affects the behavior of the class, but you cannot use subclassing
- ✓ *Replace the type code with a state/strategy object*

# Replace Type Code with State/Strategy Example



# Replace Type Code with State/Strategy Example

```
class Employee {  
    private EmployeeType type;  
    private float salary;  
    private float commission;  
    ...  
    public void setEmployeeType(EmployeeType type) {  
        this.type = type;  
    }  
    public float salary()  
        return salary;  
    }  
    ...  
    public float pay() {  
        return type.pay(  
    }  
}
```

```
class Engineer extends EmployeeType {  
    float pay(Employee employee) {  
        return employee.salary();  
    }  
}  
  
class Salesman extends EmployeeType {  
    float pay(Employee employee) {  
        return employee.salary() +  
            employee.commission();  
    }  
}
```

# Replace Type Code with State/Strategy Example

```
class Employee {
    private EmployeeType type;
    private float salary;
    private float commission;
    ...
    public void setEmployeeType(EmployeeType type) {
        this.type = type;
    }
    public float salary() {
        return salary;
    }
    ...
    public float pay() {
        return type.pay(
    }
}
```

```
enum EmployeeType {
    ENGINEER {
        float pay(Employee employee) {
            return employee.salary();
        }
    },
    SALESMAN {
        float pay(Employee employee) {
            return employee.salary() +
                employee.commission();
        }
    };
    abstract float pay(Employee employee);
}
```

# Switch statements

- Ugh.
- Switch statements often end up duplicated across the system
- Indicative of lack of OO style and underuse of polymorphism
- Special case: a conditional that chooses different behavior based on the type of an object
- Potential refactorings:
  - **Extract method**, **Move method**, **Replace type code with subclasses**, **Replace type code with state/strategy**, **Replace conditional with polymorphism**



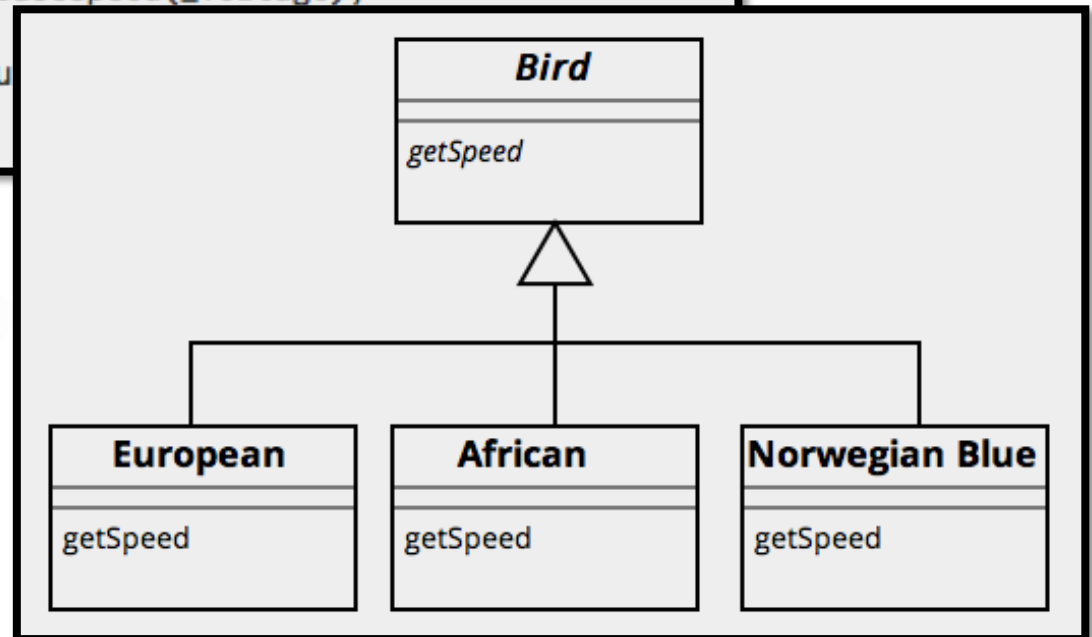
# Replace Conditional with Polymorphism

- You have a condition that chooses different behavior depending on the type of object
- ✓ *Move each leg of the conditional to an overriding method in a subclass*
  - Make the original method abstract (why?)
    - Otherwise, you're introducing an instance of the **Refused Request** smell... coming up



# Replace Conditional with Polymorphism Example

```
double getSpeed() {  
    switch (_type) {  
        case EUROPEAN:  
            return getBaseSpeed();  
        case AFRICAN:  
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;  
        case NORWEGIAN_BLUE:  
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);  
    }  
    throw new RuntimeException ("Should not reach here");  
}
```

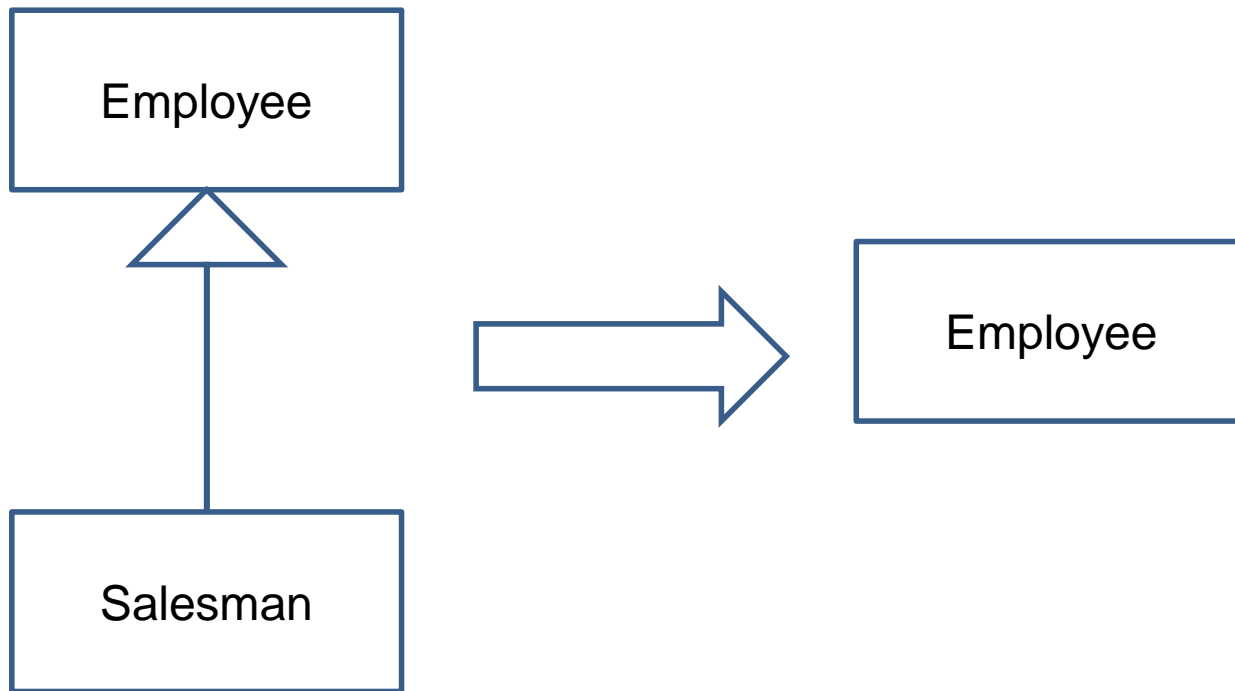


# Lazy Class

- Each class costs something to maintain and understand
  - We don't often intentionally make lazy classes, but it can commonly result from downsizing or adding things speculatively
- Potential refactorings:
  - **Collapse hierarchy**, **Inline class**

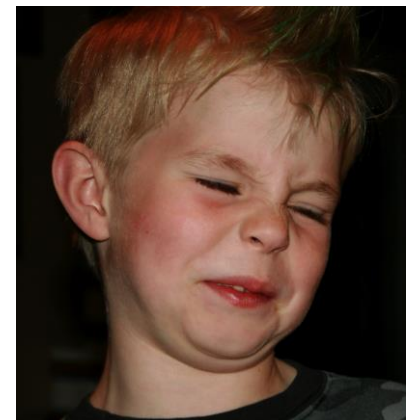


# Collapse Hierarchy

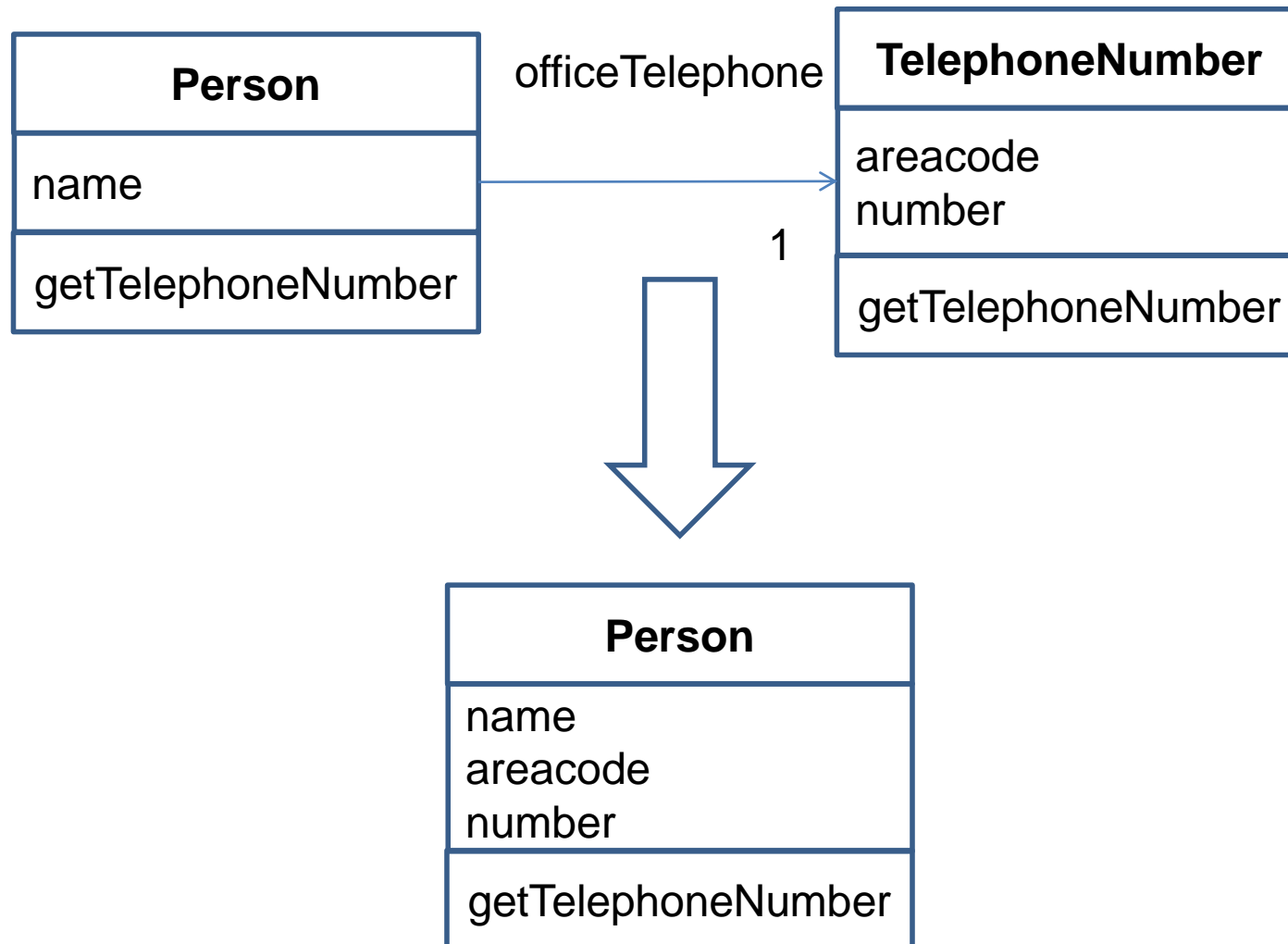


# Speculative Generality

- We sometimes create hooks and special cases to handle things that aren't required
  - E.g., “we might need a method to do X some day”
  - This is evident when you have generic or abstract code that is not actually needed (at least not yet)
- Potential refactorings:
  - **Collapse hierarchy**, **Rename method**, **Remove parameter**, **Inline class**



# Inline class



# Temporary Field

- The inclusion of an instance variable that is only set in some instances
- The rest of the time, the field is empty or (worse) contains irrelevant data
  - This hampers understandability and can lead to accidental errors based on context
- Potential refactorings
  - **Extract class**, **Introduce null object**

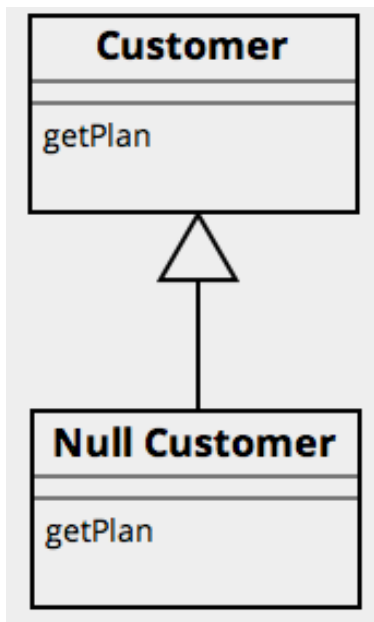


# Introduce Null Object

- You have repeated checks for a null values
  - Ugh. It's ugly and hard to read.
- ✓ *So replace the null value with a null object!*

# Introduce Null Object Example

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





# Message Chains

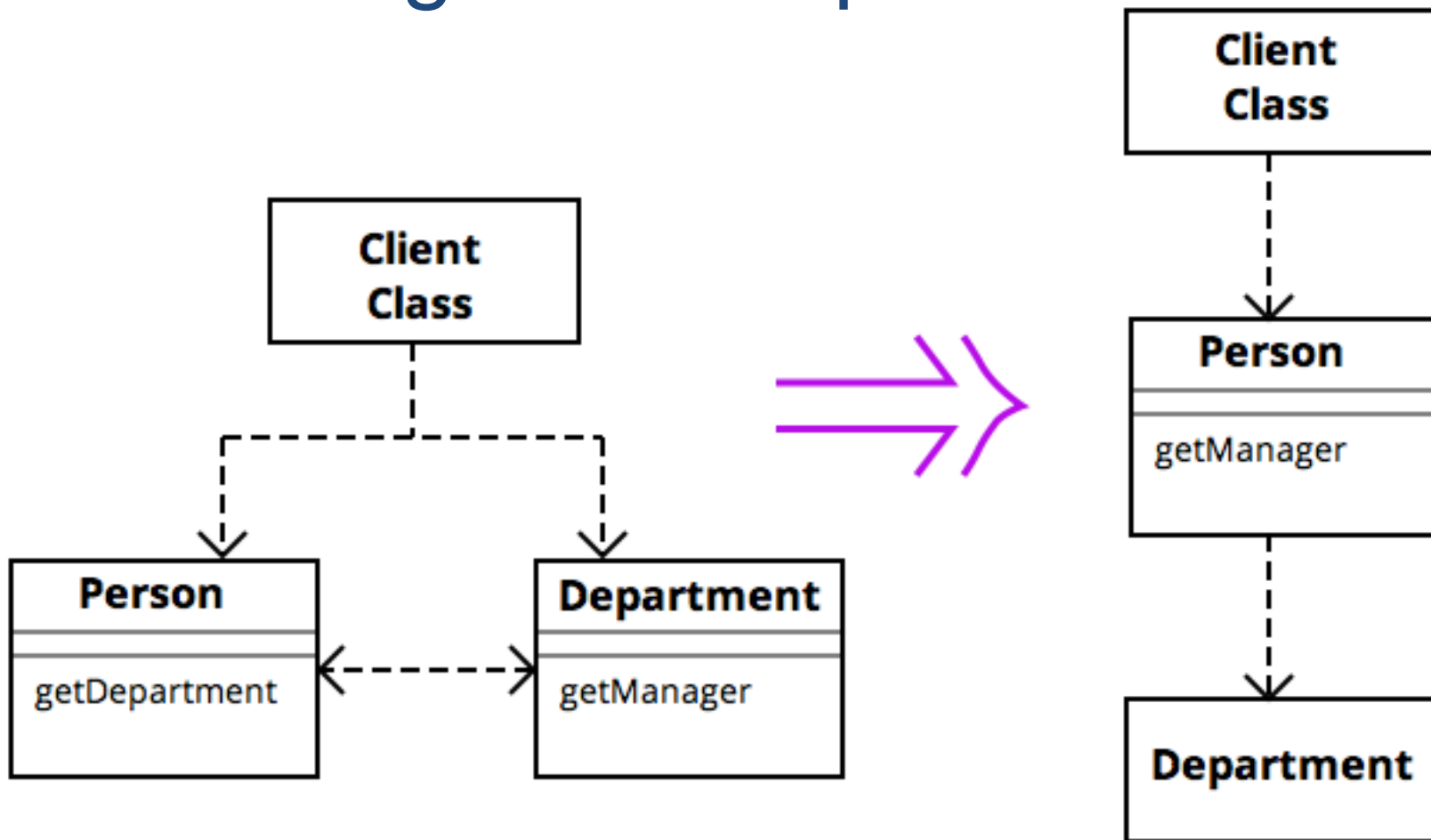
- Occur when you see a long line of method calls or temporary variables to get some data
  - E.g., long string of `getThis().getThat().getSomething()`
- Makes the code dependent on the algorithm for navigating the relationships between components
  - Failure to shelter outside objects from the implementation details
- Potential refactorings:
  - **Hide delegate**, **Extract method**, **Move method**



# Hide Delegate

- A client is calling a delegate class of an object
- ✓ *Create methods on the server to hide the delegate*

# Hide Delegate Example



# Some codes

- `class person{`
- `Department _department;`
- 
- `public Department getDepartment() {`
- `return _department;`
- `}`
- 
- `public void setDepartment(Department arg) {`
- `_department = arg;`
- `}`
- `}`

# Some codes (continued.)

- `class Department {`
- `private String _chargeCode;`
- `private Person _manager;`
- 
- `public Department (Person manager) {`
- `_manager = manager;`
- `}`
- 
- `public Person getManager() {`
- `return _manager;`
- `}`
- `...`

## Some codes (continued.)

- If you want to obtain a manager:
- `manager = john.getDepartment().getManager();`
- modification:
- `public Person getManager(){`
- `return _department.getManager();`
- `}`

# Middle Man

- Delegation is good, and that's why we make objects
- But sometimes, we end up with a design that all an object is doing is passing along calls to another object
  - For no apparent reason (e.g., an Adapter would be an exception)
- There is a fine line between information hiding and delegation overhead.
- Potential refactorings:
  - **Remove middle man** (duh!), **Inline method**, **Replace delegation with inheritance**

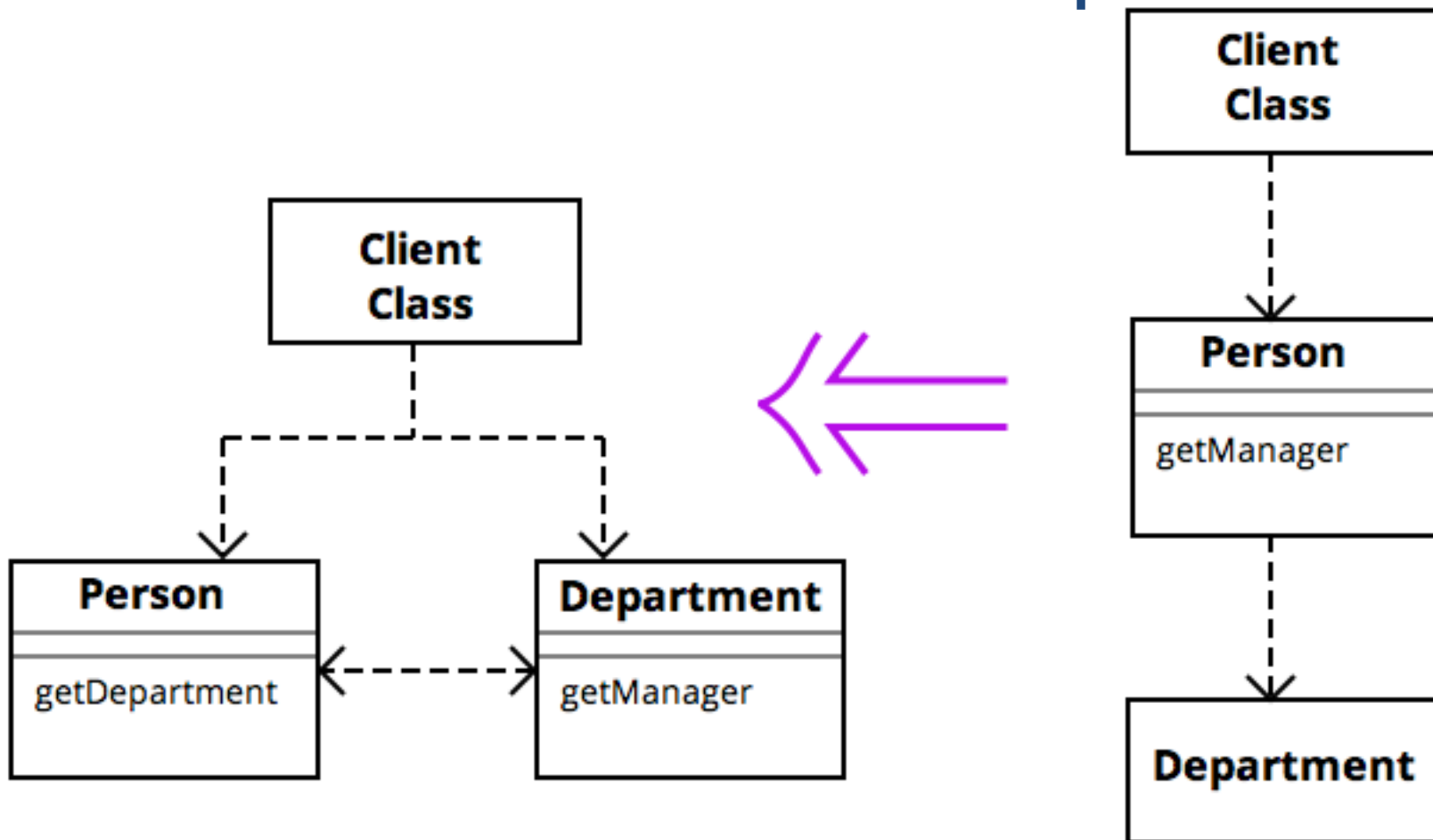


# Remove Middleman

- A class is doing too much simple delegation
- ✓ *Get the client to call the delegate directly*
  - This is the exact dual for **Hide Delegate**



# Remove Middleman Example



# Inline Method

- A method's body is just as clear as its name
- ✓ *So put the method's body into the body of its callers and remove the method*

# Inline Method Example

```
int getRating() {  
    return (moreThanFiveLateDeliveries()) ? 2 : 1;  
}  
boolean moreThanFiveLateDeliveries() {  
    return _numberOfLateDeliveries > 5;  
}
```



```
int getRating() {  
    return (_numberOfLateDeliveries > 5) ? 2 : 1;  
}
```

# Inappropriate Intimacy

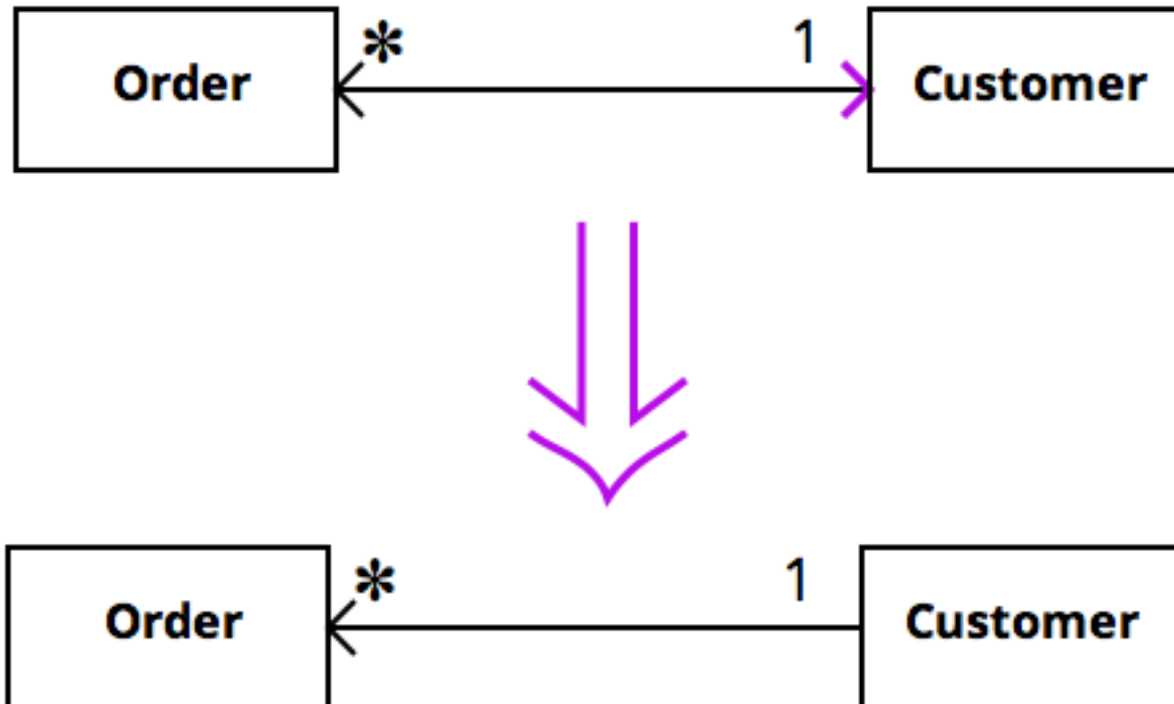
- Classes sometimes end up delving too much into each others' private methods and fields
- Related: *Data Class* – classes that have fields and getters and setters but nothing else
  - Almost assuredly being manipulated in far too much detail by others
- Potential refactorings:
  - **Move method**, **Move field**, **Change bidirectional association to unidirectional association**, **Extract class** (if classes do in fact have common interests), **Hide delegate** (allow another class to act as a go-between), **Encapsulate collection** (for Data Class)



## Change Bidirectional Association to Unidirectional

- You have a two-way association but one class no longer needs access to the other
- ✓ *So drop the unneeded end of the association*

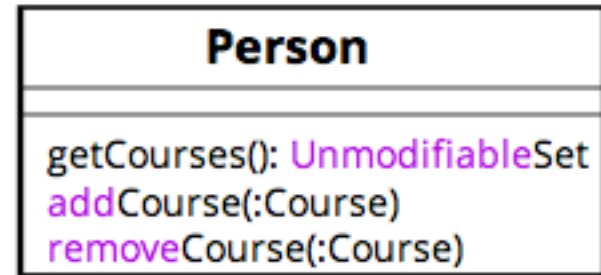
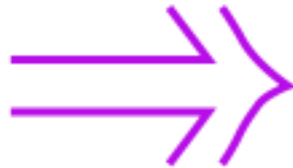
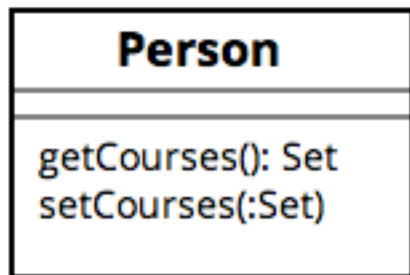
## Change Bidirectional Association to Unidirectional



# Encapsulate Collection

- A method returns a collection
  - This can be confusing because it may seem to the caller that he can make changes to the collection
- ✓ *Make it return a read-only view and provide add/remove methods*

# Encapsulate Collection Example





# Alternative Classes with Different Interfaces

- Classes can be completely different on the outside but end up being the same internally
- Basically, you should find the similarities in the two classes, then refactor them to share a common interface
- Potential refactorings:
  - **Extract superclass**, **Unify interfaces with adapter**



# Refused Bequest

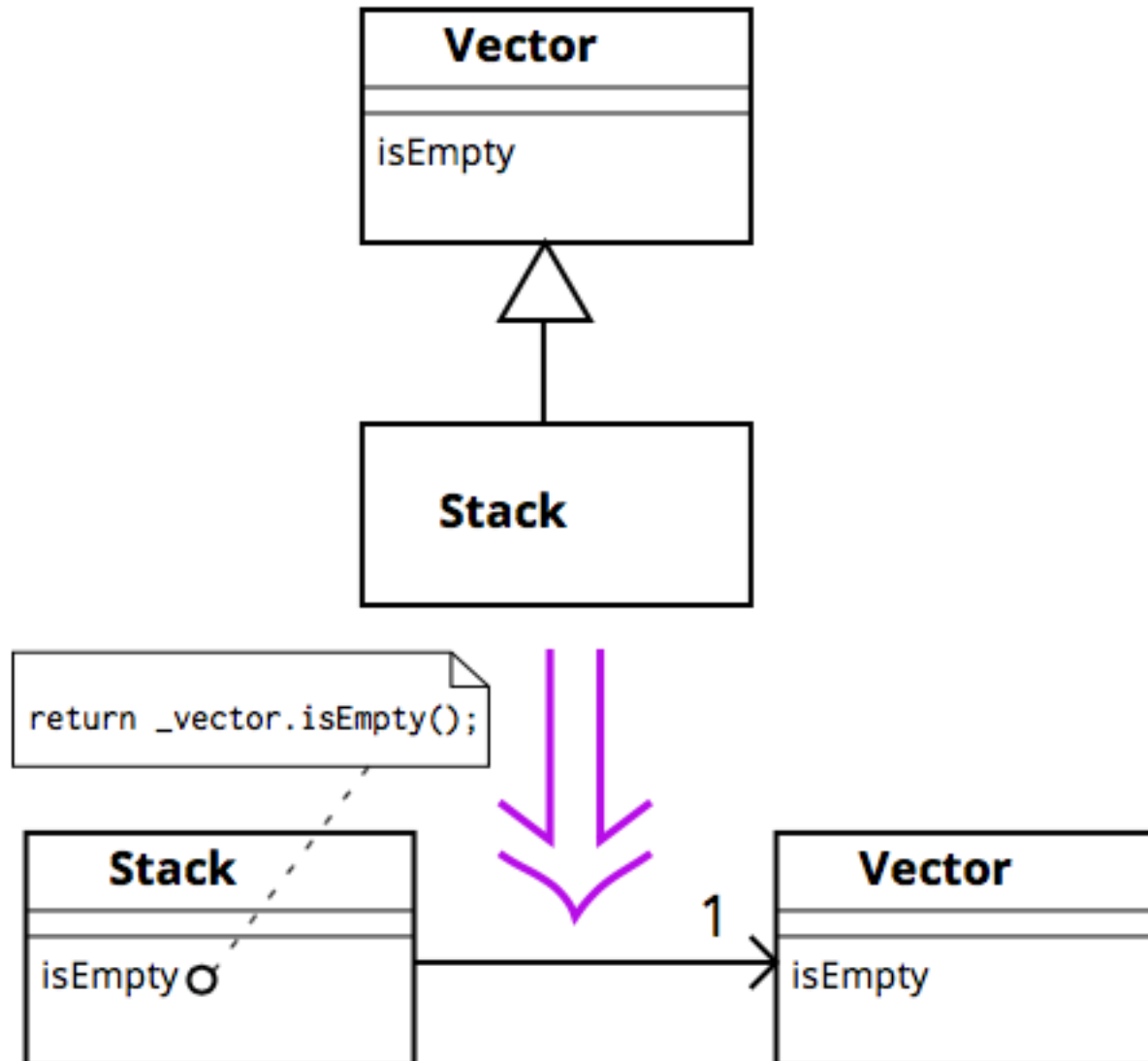
- Happens when you inherit code you don't want
  - i.e., a child class uses very little of the functionality of some parent (base) class
- The worst (strongest smell) here is when the child reimplements the behavior from the parent class
- Potential refactorings:
  - **Push down field**, **Push down method**, **Replace inheritance with delegation**



# Replace Inheritance with Delegation

- A subclass uses only part of a superclass's interface or does not want to inherit data
- ✓ *Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing*

# Replace Inheritance with Delegation Example



# QUESTIONS?

---