

Advanced Software Development: Refactoring

**CPSC 501: Advanced Programming Techniques
Fall 2022**

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Friday, September 2, 2022



UNIVERSITY OF
CALGARY

We're actually doing it?

We're actually doing it!

Refactoring Tools

- **Automate the refactoring process**
 - Restructures code while preserving behavior
 - Reduces the need to test
- Are incorporated into some IDEs
 - Xcode supports 6 common refactorings:
 - Rename, Extract, Create Superclass, Move up, Move down, Encapsulate
 - Eclipse supports ~18 refactorings
- But note that **Fowler (text) lists ~72 refactorings**
 - Manual refactoring will still often be necessary

Refactoring Principles

- **Refactoring:** is the **disciplined process** of **changing** the **internal structure** of software to make it easier to understand and maintain, **without changing** its **external** observable behavior

Refactoring Principles

- **Why refactor?**
 1. **Improves the design of software**
 - Reverses the “**decay**” of cumulative ad hoc changes
 2. **Makes software more readable**
 - A clear design is easier to understand and maintain
 - Use refactoring to learn about unfamiliar code
 3. **Helps you find and eliminate bugs**
 4. **Helps you program faster**
 - A **poor design** prevents rapid development

Refactoring Principles

- **When should you refactor?**
 1. Continuously, as you develop or modify code
 2. Whenever you duplicate code
 3. When adding functionality to code
 - i.e. change the design to make adding features easy
 4. As you find and fix bugs
 - It's easier to spot bugs when the design is clear
 5. As you do a code review

Not so fast

Refactoring Principles

- **Problems with Refactoring**
 - **Many refactorings change a class's public interface**
 - E.g. methods may be renamed or removed
 - Not a problem if you can edit all calling code
 - **If the interface is published, you need a transition period** where the old interface is kept until clients adopt the new interface
 - Mark an old method as **deprecated** and have it call the new method

Refactoring Principles

- **You may not be able to refactor your way out of a design mistake**
 - May be necessary to do more upfront design
- **If software is tightly coupled to a database, changing the object model may cause changes to the database schema**
 - Forces you to migrate data, which is difficult and expensive
 - Isolate changes by putting a layer between the database and object model

Rule of thumb



Which thumb?

Refactoring Principles

- **Don't refactor when:**
 - Its easier to rewrite from scratch
 - You are close to a release deadline
- Refactoring and design
 - **Refactoring is not a replacement for upfront design**
 - But it **lets you create a simple, upfront design** that does not build in unneeded flexibility
 - i.e. you can always refactor later if necessary

Refactoring Principles

- Refactoring and performance
 - **Refactoring often makes software run more slowly**
 - **More function structure is complexity with runtime cost**
 - **But also more amenable to performance tuning**
 - If well factored, **“hot spots”** will be isolated to a few short methods
 - Found using a profiler late in development
 - **Tune the hot spots only**
 - Tuning the other code is a waste

When to Refactor

- **No hard and fast rules**
 - Best to use informed intuition
 - i.e. try to detect **“Bad smells in code”**

Ok lots of 'rules'

Lots of thumbs?

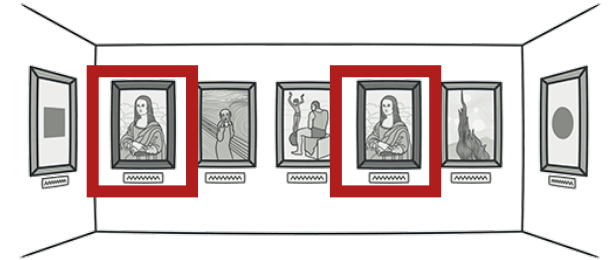
**Names may be slightly different
between these edition 1 and 2018
edition 2**

Smell: Duplicated Code

When to Refactor - Duplicated code

- **Duplicated code**

- (some IDEs will find simple versions)



Illustrations by [Dmitry Zhart](#)

1. If the same code in two or more places in the same class

- **Extract Method**, and call it from each place

2. If the same code in two sibling classes

- **Extract Method**, if necessary
- **Pull Up Method** into common superclass

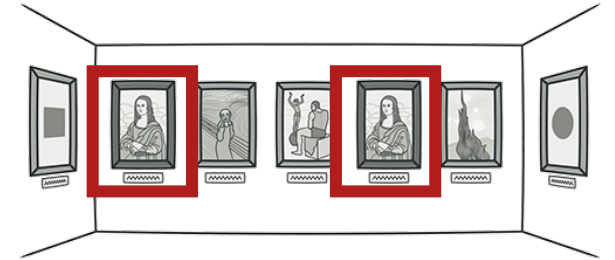
Pull Up Method (from intro)

```
public class Employee {  
    private String lastName;  
    public String getLastName() {  
        return lastName;  
    }  
    ...  
}  
  
public class Clerk extends Employee {  
    ...  
}  
  
public class Cashier extends Employee {  
    ...  
}
```



When to Refactor - Duplicated code

- **Duplicated code**



Illustrations by [Dmitry Zhart](#)

3. If similar code in sibling classes

- **Extract Method**, if necessary
- **Form Template Method** to put common code in superclass, differing code in subclasses (example to come later)

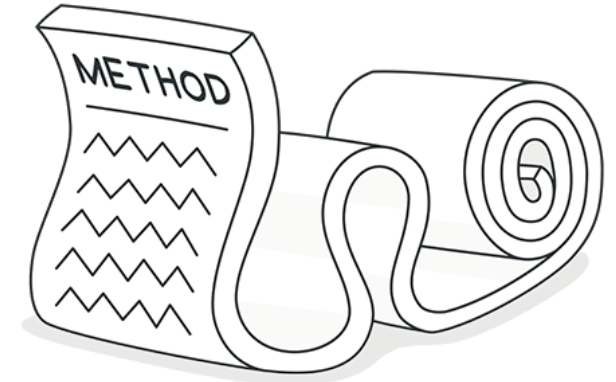
4. If the same code in unrelated classes

- **Extract Class** in one class, and use the new class in the other classes

Smell: Long Code

When to Refactor – Long Method

- **Long method** (*my main is 500 lines long!*)
 - Decompose into small methods
 - Sometimes just one line long
 - **Extract Method** on blocks of code that can be separated out
 - Look for “clumps”
 - E.g. Commented blocks, loops, conditionals, etc.
 - May need to **Replace Temp with Query** to enable the extraction



Illustrations by [Dmitry Zhart](#)

**Ex: Long Code –
Replace temp with
query**

Ex: Replace Temp with Query

- You have parameter initialization that is temporary
 - Replace this code with a function query that returns the result that was initialization

Ex: Replace Temp with Query

```
int basePrice = this._quantity * this._itemPrice;  
if (basePrice > 1000)  
    ...
```

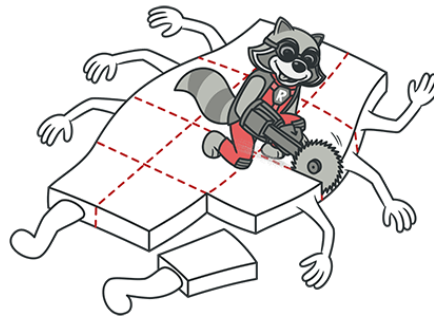
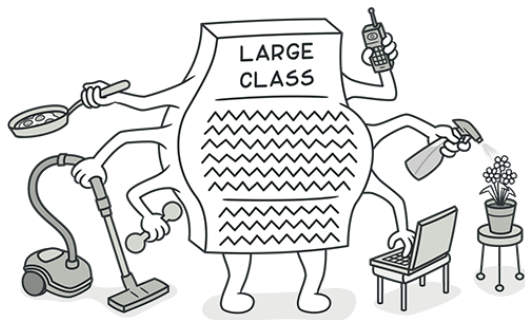
- Change above into the following

```
int getBasePrice() {this._quantity * this._itemPrice;}  
...  
int basePrice = getBasePrice();  
if (basePrice > 1000)  
    ...
```

Smell: Large Class

When to Refactor – Large Class

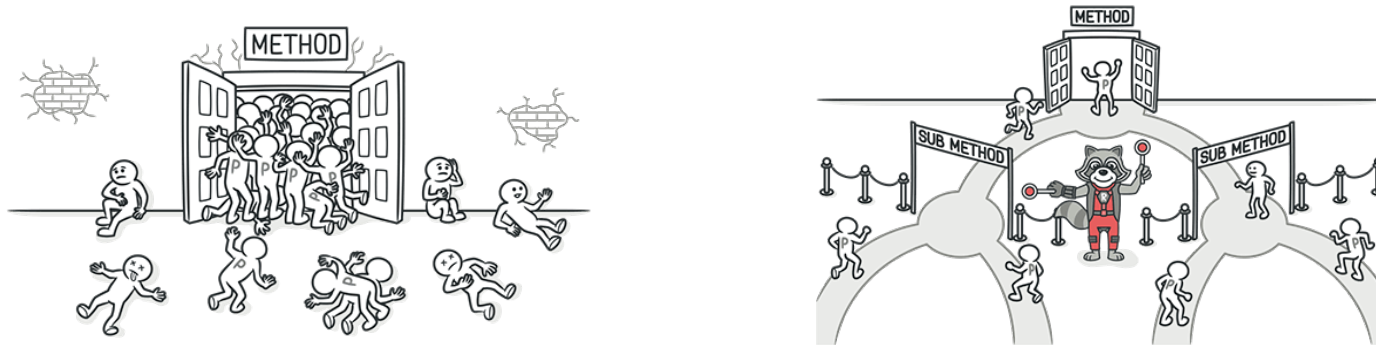
- **Large class** (*my main class handles database I/O!*)
 - **Tries to do too many different things (not cohesive)**
 - Too many instance variables, and/or
 - Too much code
 - **Extract Class** or **Extract Subclass** to separate out “bundles” of data and responsibilities



Smell: Long Parameter List

When to Refactor - Long Parameter List

- **Long parameter list**
 - Better to pass in an object, so the method can get the data it needs
 - Shorten list with **Preserve Whole Object** (pass in object instead of pulling of data as multiple parameters) or **Introduce Parameter Object**



Smell: Divergent Change

When to Refactor – Divergent Change

- **Divergent change** (5 methods handle database, 3 handle networking in class!)
 - Occurs when a class changes in distinct ways for differing reasons
 - E.g. You change 3 methods together for one reason, and 5 other methods for another
 - Determine what changes for a single cause, and **Extract Class** to bundle these together

Smell: Shotgun Surgery

When to Refactor – Shotgun surgery

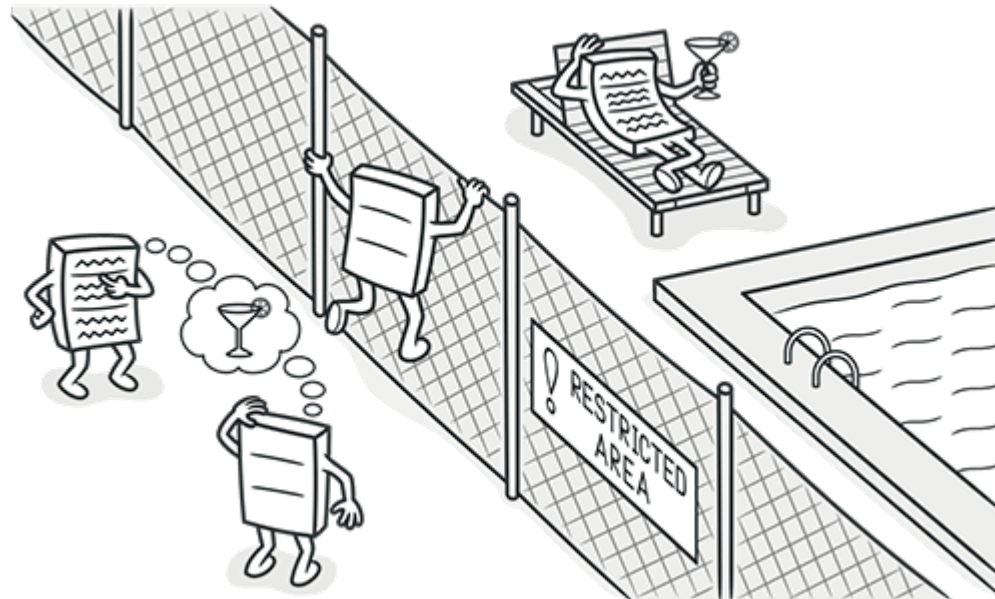
- **Shotgun surgery** (*my database code is in 5 different classes!*)
 - A single change causes many little changes to several different classes
 - Use **Move Method** and **Move Field** to put changes into a single class
 - Sometimes best to **Inline Class**

Smell: Feature Envy

When to Refactor – Feature Envy

- **Feature Envy**

- A class does a calculation that belongs elsewhere
 - i.e. it uses too much data from another class
- Put it into the proper class with **Move Method**

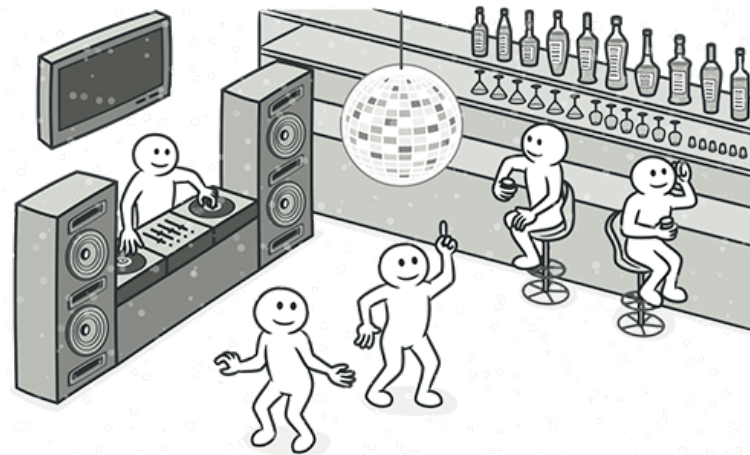


Smell: Data Clumps

When to Refactor – Data clumps

- **Data clumps**

- Data clusters together in fields or parameter lists
- **Extract Class** to change clumps into an object
- Shrink parameter lists with **Introduce Parameter Object** or **Preserve Whole Object**

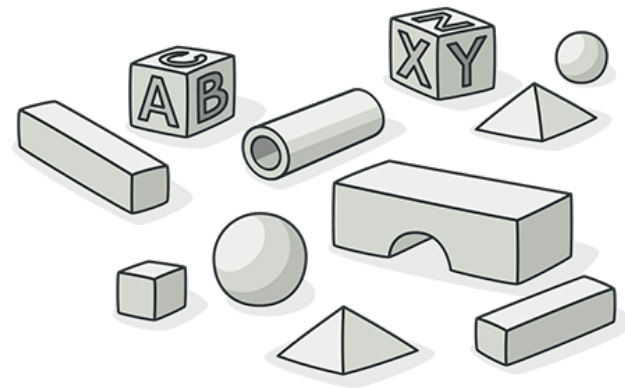


Illustrations by [Dmitry Zhart](#)

Smell: Primitive Obsession

When to Refactor – Primitive Obsession

- **Primitive Obsession**
 - Often better to use a class instead of a primitive type
 - Allows things like range checking, formatting, etc.
 - Done with **Replace Data Value with Object**
 - If the primitive is a type code, use
 - **Replace Type Code** with
 - **Class, or**
 - **Subclasses, or**
 - **State/Strategy**

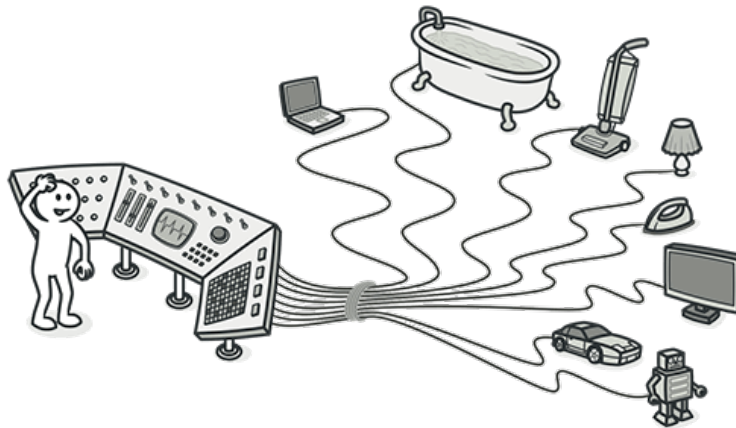


Illustrations by [Dmitry Zhart](#)

Smell: Switch Statements

When to Refactor – Switch statements

- **Switch statements**
 - Are rare in good OO code
 - If switching on a type code, **Replace Conditional with Polymorphism**
 - Easier to add subclasses than changing many switch statements



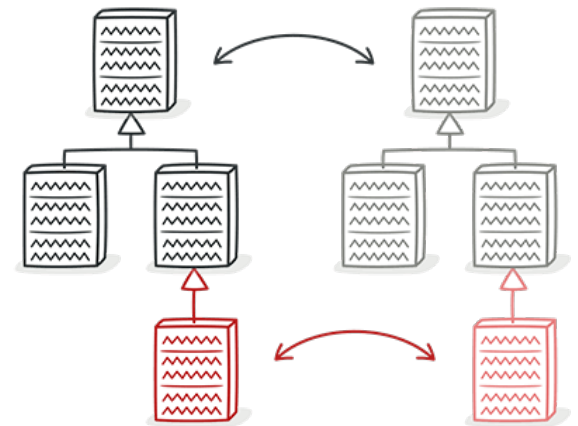
Illustrations by [Dmitry Zhart](#)

Smell: Parallel Inheritance

When to Refactor – Parallel inheritance

- **Parallel inheritance hierarchies**

- When you make a subclass of one class, you also make a subclass of another
 - Special case of shotgun surgery
- Eliminate one hierarchy by shifting data and responsibilities to the other
 - **Move Method** and **Move Field**

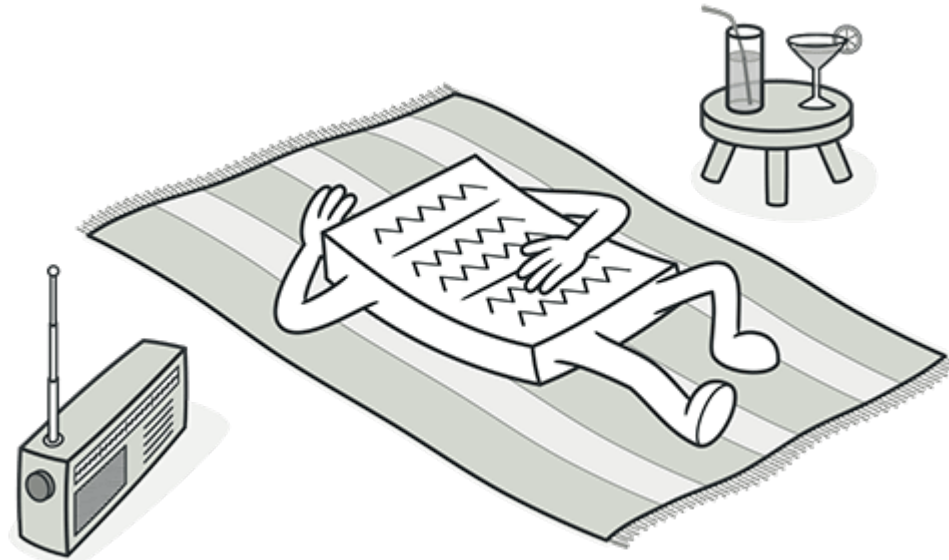


Illustrations by [Dmitry Zhart](#)

Smell: Lazy Class

When to Refactor – Lazy class

- **Lazy class**
 - A class doesn't do enough to justify its existence
 - May result from other refactorings like **Move Method**
 - Eliminate it with **Collapse Hierarchy** or **Inline Class**

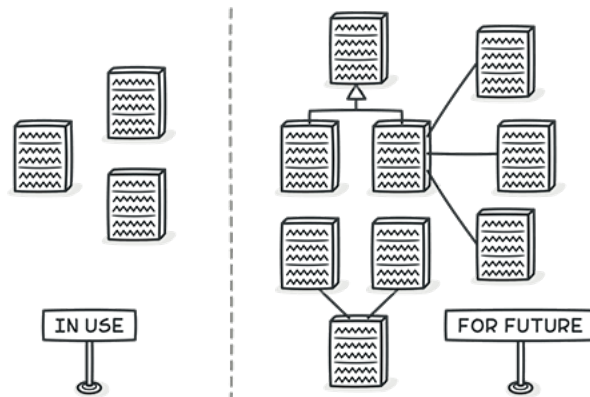


Illustrations by [Dmitry Zhart](#)

Smell: Speculative Generality

When to Refactor – Speculative generality

- **Speculative generality**
 - You added code for future expansion that never occurred
 - Remove useless abstract classes with **Collapse Hierarchy**
 - Remove unneeded delegation with **Inline Class**
 - Remove unused parameters with **Remove Parameter**



Smell: Temporary Field

When to Refactor – Temporary Field

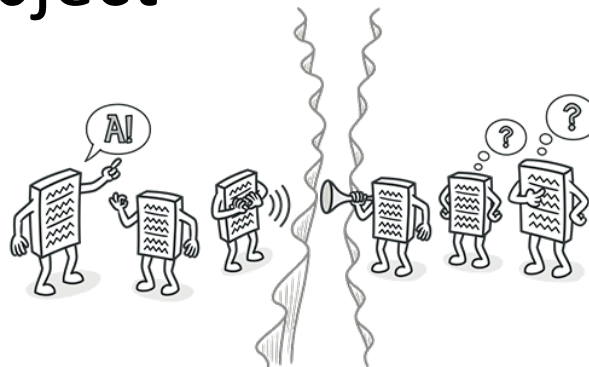
- **Temporary field**
 - An instance variable is set and used only part of the time
 - **Extract Class**, moving over the “orphan variables” and related methods

Smell: Message Chains

When to Refactor – Message chains

- **Message chains**

- A client follows a chain of referring objects, and sends a message to the last object
 - Any change to intermediate relationships causes client code to change
- **Hide Delegate** on the first object in the chain so it returns the last object

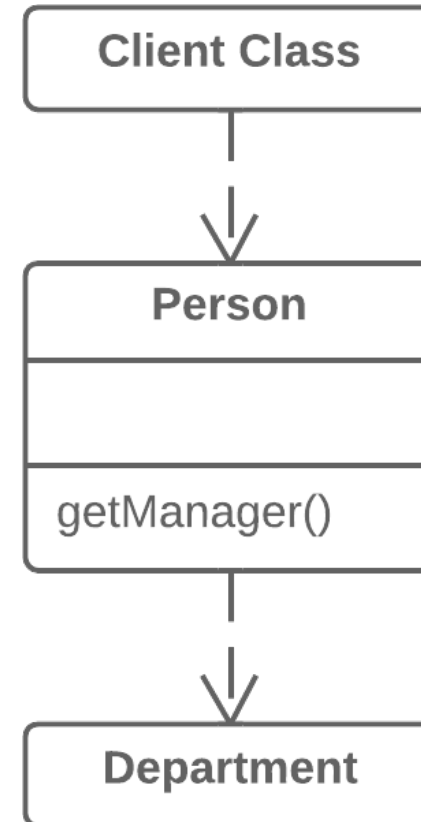
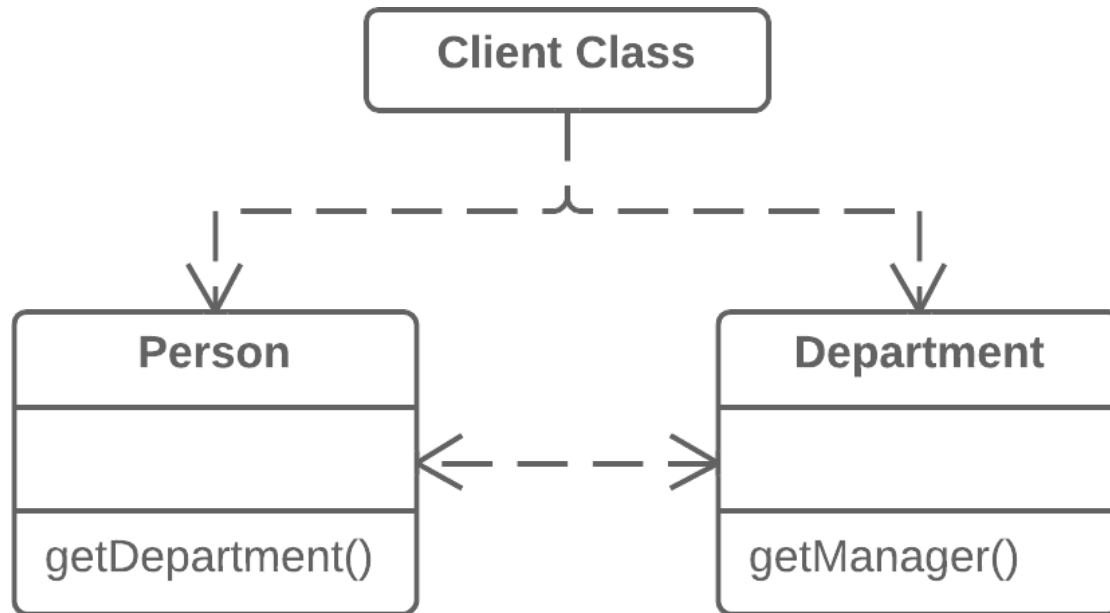


Ex: Message Chains – Hide delegate

Hide Delegate

- Client talks to one object to get data, then talks to object in that data to do something
 - Maybe farther down chain
- Put method in first object that is in charge of passing on message (detaches client from chain structure)

Hide Delegate



Smell: Middle Man

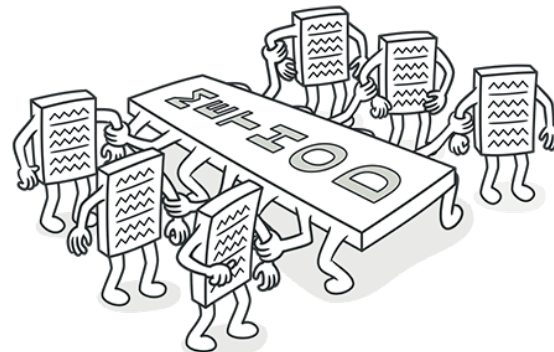
When to Refactor – Middle Man

- **Middle Man**
 - Where most methods of a class delegate to another class
 - **Remove Middle Man**, so you talk to the delegated object directly

Smell: Inappropriate Intimacy

When to Refactor - Inappropriate intimacy

- **Inappropriate intimacy**
 - A class knows too much about another class's private parts
 - **Move Method** and **Move Field** to the first class
 - Or **Extract Class** to put commonality in a safe place
 - **Replace Inheritance with Delegation** if a subclass knows too much about its parents



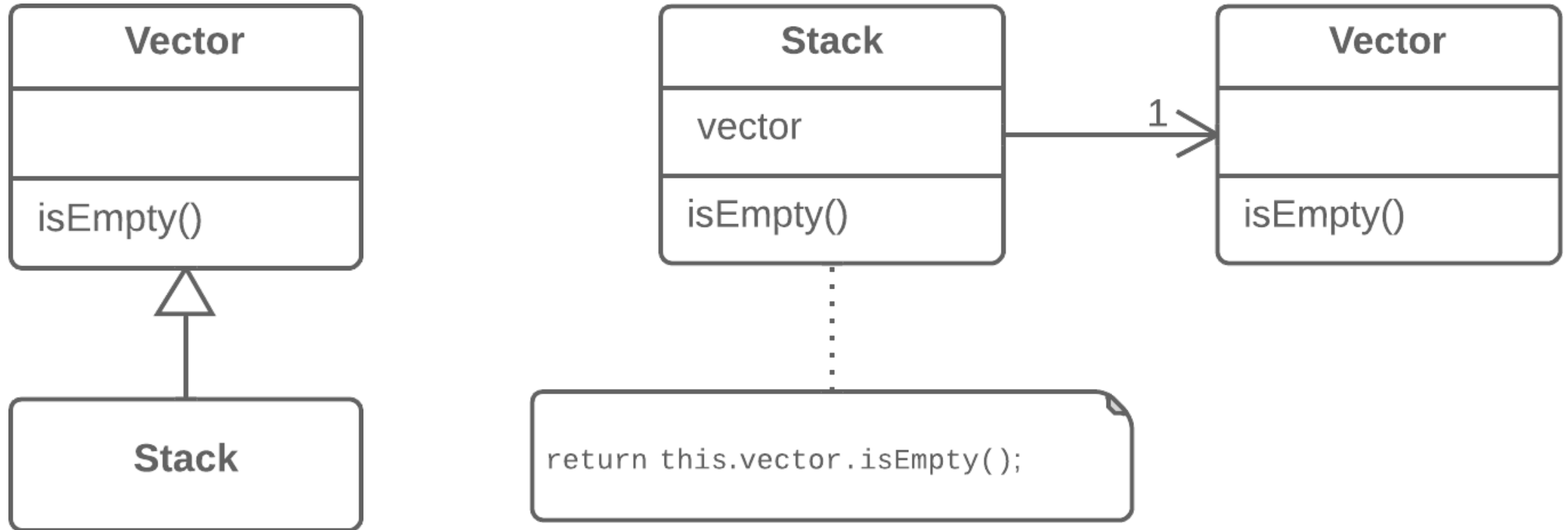
Illustrations by [Dmitry Zhart](#)

**Ex: Inappropriate
Intimacy – Replace
Inheritance with
Delegation**

Replace Inheritance with Delegation

- Inheritance structures can leave parts of a super-class exposed by a sub-class
 - Instead of a class extending a parent, the previous super-class can instead be initialized as a data object in the previous sub-class
 - This protects things exposed via regular inheritance

Replace Inheritance with Delegation



Smell: Alternative Classes

When to Refactor – Alternative classes

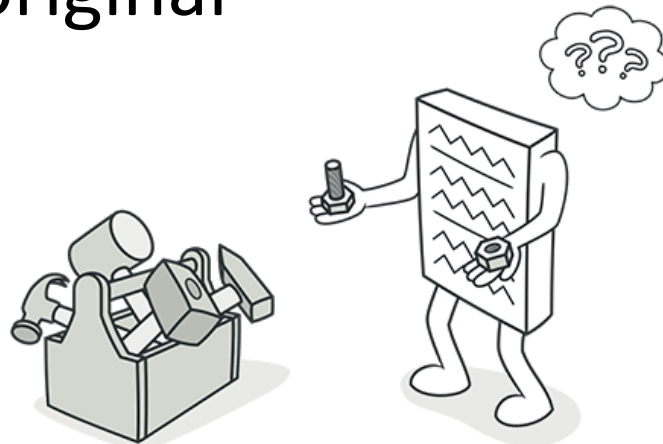
- **Alternative classes with different interfaces**
 - Two or more classes do the same thing, but have inconsistent interfaces
 - Use **Rename Method** and **Move Method** to give the classes identical interfaces
 - If redundant, **Extract Superclass**



Smell: Incomplete Library Class

When to Refactor – Incomplete Library Class

- **Incomplete Library Class**
 - You can't use **Move Method** on code you can't change
 - **Introduce Foreign Method** into a client class
 - Best for only one or two methods
 - **Introduce Local Extension** to create a subclass or wrapper of the original



Illustrations by [Dmitry Zhart](#)

Ex: Incomplete Library Class – Introduce Foreign Method

Introduce Foreign Method

- A utility class doesn't contain the method that you need and you can't add the method to the class.
- Add the method to a client class and pass an object of the utility class to it as an argument.

Introduce Foreign Method

```
class Report {
    void sendReport() {
        Date nextDay = new Date(previousEnd.getYear(),
            previousEnd.getMonth(), previousEnd.getDate() + 1);
        ...
    }
}
```

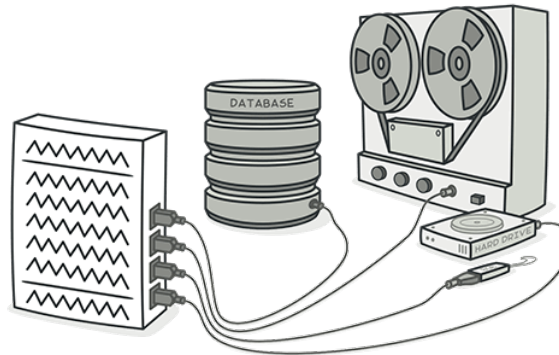
```
class Report {
    void sendReport() {
        Date nextDay = nextDay(previousEnd);
        ...
    }
    private static Date nextDay(Date arg) {
        return new Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1);
    }
}
```

Smell: Data Class

When to Refactor – Data Class

- **Data Class**

- Is a class with no behavior
 - i.e. has only get and set methods
- **Move Methods** (that apply to that data) into the data class
 - May need to **Extract Method** first



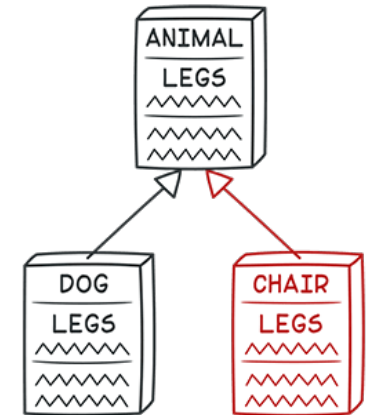
Illustrations by [Dmitry Zhart](#)

Smell: Refused Bequest

When to Refactor - Refused Bequest

- **Refused Bequest**

- A subclass doesn't use all the methods and data that it inherits
- Reorganize the class hierarchy
 - **Push Down Method** and **Push Down Field** to create a sibling for the unused behavior
- If the subclass does not support the superclass interface, **Replace Inheritance with Delegation**

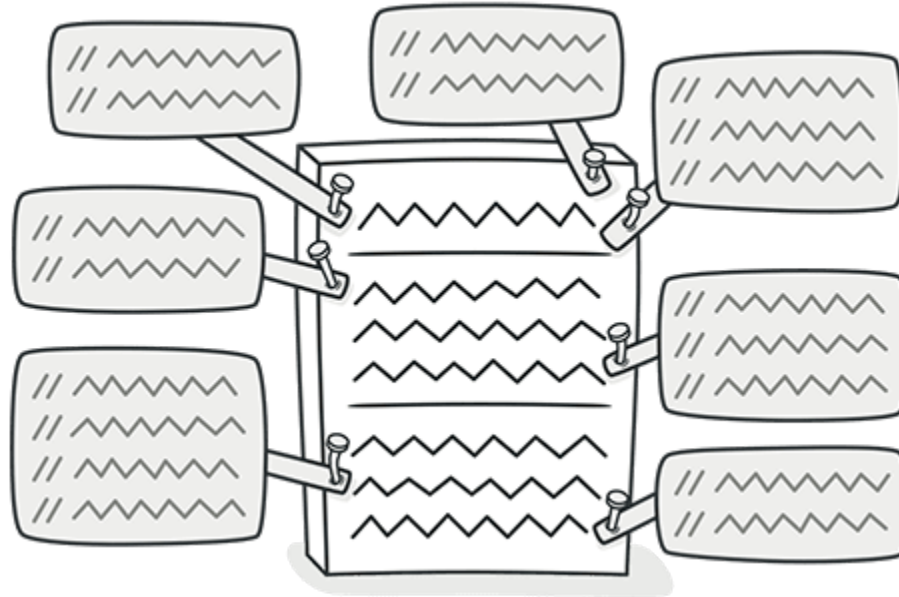


Illustrations by [Dmitry Zhart](#)

Smell: Worrisome Comments

When to Refactor - Worrisome comments

- **Comments that explain bad code**
 - **Extract Method** on commented blocks of code
 - **Rename Method** to make purpose clearer



Illustrations by [Dmitry Zhart](#)

That was a lot of things

I don't remember all the changes

Catalog of Refactorings

- Format:
 - Name
 - Summary
 - Motivation
 - Mechanics
 - Examples

Catalog of Refactorings

- Organized into chapters with related refactorings:
 - **Composing Methods**
 - Are refactorings that reorganize the methods of a class
 - And deal with troublesome local variables
 - **Extract Method** most commonly used

Catalog of Refactorings

- Organized into chapters with related refactorings:
 - **Composing Methods**
 - Are refactorings that reorganize the methods of a class
 - And deal with troublesome local variables
 - **Extract Method** most commonly used
 - **Moving Features Between Objects**
 - Reassigns responsibilities to other classes
 - **Move Method, Move Field, and Extract Class** are commonly used

Catalog of Refactorings

- Organized into chapters with related refactorings:
 - **Organizing Data**
 - Make working with data easier
 - Some refactorings promote better encapsulation
 - E.g. **Encapsulate Field**
 - Others eliminate type codes
 - **Simplifying Conditional Expressions**
 - Used to make logic within a method clearer
 - E.g. **Decompose Conditional**
 - **Replace Conditional with Polymorphism** changes the class structure

Catalog of Refactorings

- Organized into chapters with related refactorings:
 - **Making Method Calls Simpler**
 - Use **Rename Method** to make intentions clearer
 - Some refactorings shorten parameter lists
 - E.g. **Preserve Whole Object**
 - Others simplify a class's interface
 - E.g. **Hide Method** and **Remove Setting Method**

Catalog of Refactorings

- Organized into chapters with related refactorings:
 - **Dealing with Generalization**
 - Some refactorings move responsibilities up/down the class hierarchy
 - E.g. **Pull Up Field, Push Down Method**
 - Other change the hierarchy by creating/destroying classes
 - E.g. **Extract Subclass, Collapse Hierarchy**

Catalog of Refactorings

- Organized into chapters with related refactorings:
 - **Big Refactorings**
 - Are much lengthier and time consuming than the previous refactorings
 - Involves many small refactorings
 - **Tease Apart Inheritance**
 - **Convert Procedural Design to Objects**
 - **Separate Domain from Presentation**
 - **Extract Hierarchy**

Resources for reference

- <https://refactoring.guru/refactoring/techniques>
- Fowler:
 - Read chapter 5 (Catalog of Refactorings)
 - Browse chapters 6 – 12 (individual methods)

Onward to ... refactoring examples.

Jonathan Hudson
jwhudson@ucalgary.ca
<https://pages.cpsc.ucalgary.ca/~jwhudson/>



UNIVERSITY OF
CALGARY