Lecture 8 –Refactoring

COMP2013 (AUT1 23-24)

Dr Marjahan Begum and Dr Horia A. Maior

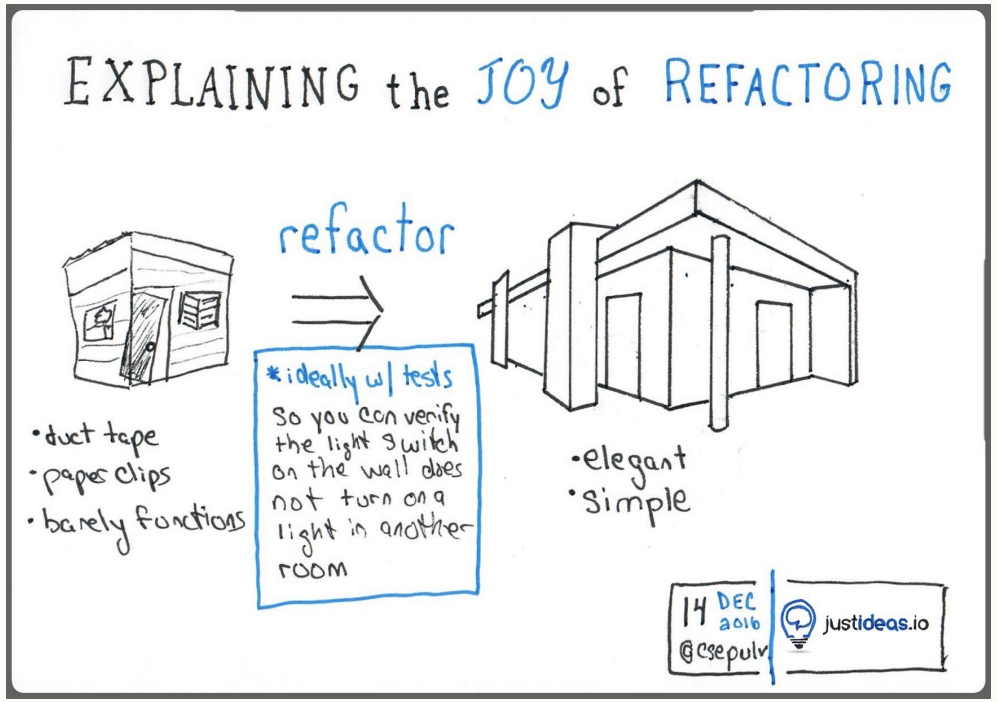
Register your attendance

COMP2013: Developing Maintainable Software
Week 9 – 4:00pm Monday – 20 November 2023

valid for 65 minutes from 3:55pm
generated 2023-10-10 03:14

Today's Learning Objectives

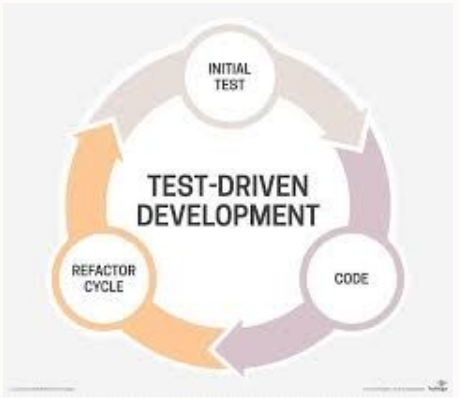
- To be able to assess when code needs refactoring by identifying Code Smells
- To understand the concept of refactoring
  - To refactor small sections of code to increase maintainability
  - To refactor larger code structures for the application of SOLID/Design Patterns
- To understand the links between regression testing, test driven development, refactoring and legacy code



https://medium.com/justideas-io/explaining-the-joy-of-refactoring-to-the-non-developer-72d97223359c

The Software Maintenance Starter Pack

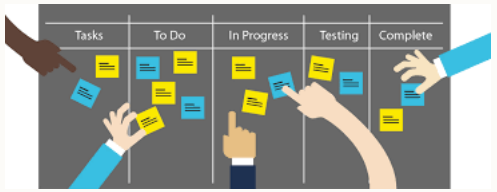
GitHub



Gradle

rm –rf *

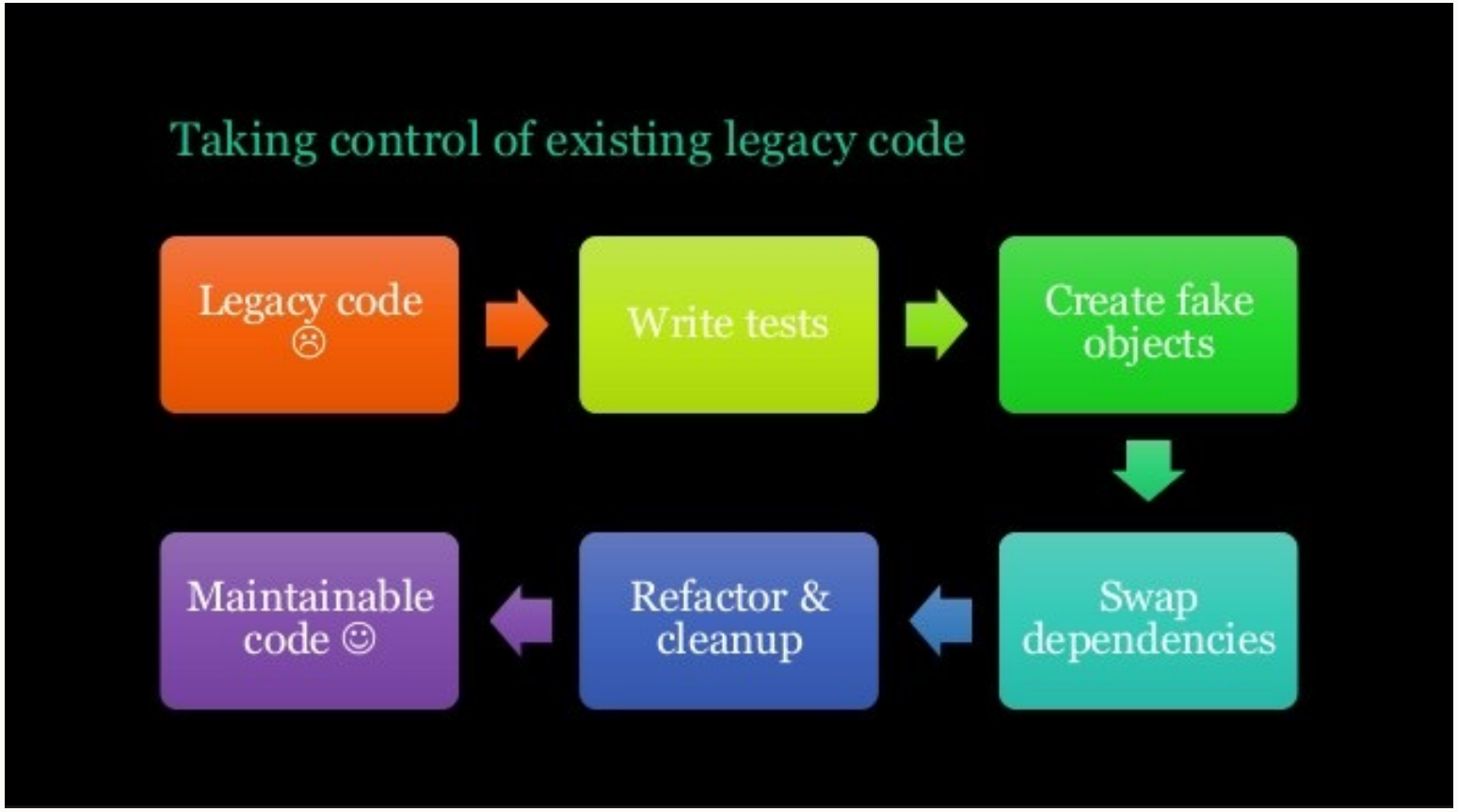
"I've just copied my code from stack overflow…its bound to work"



"Can I exceed the word limit on CWK1?

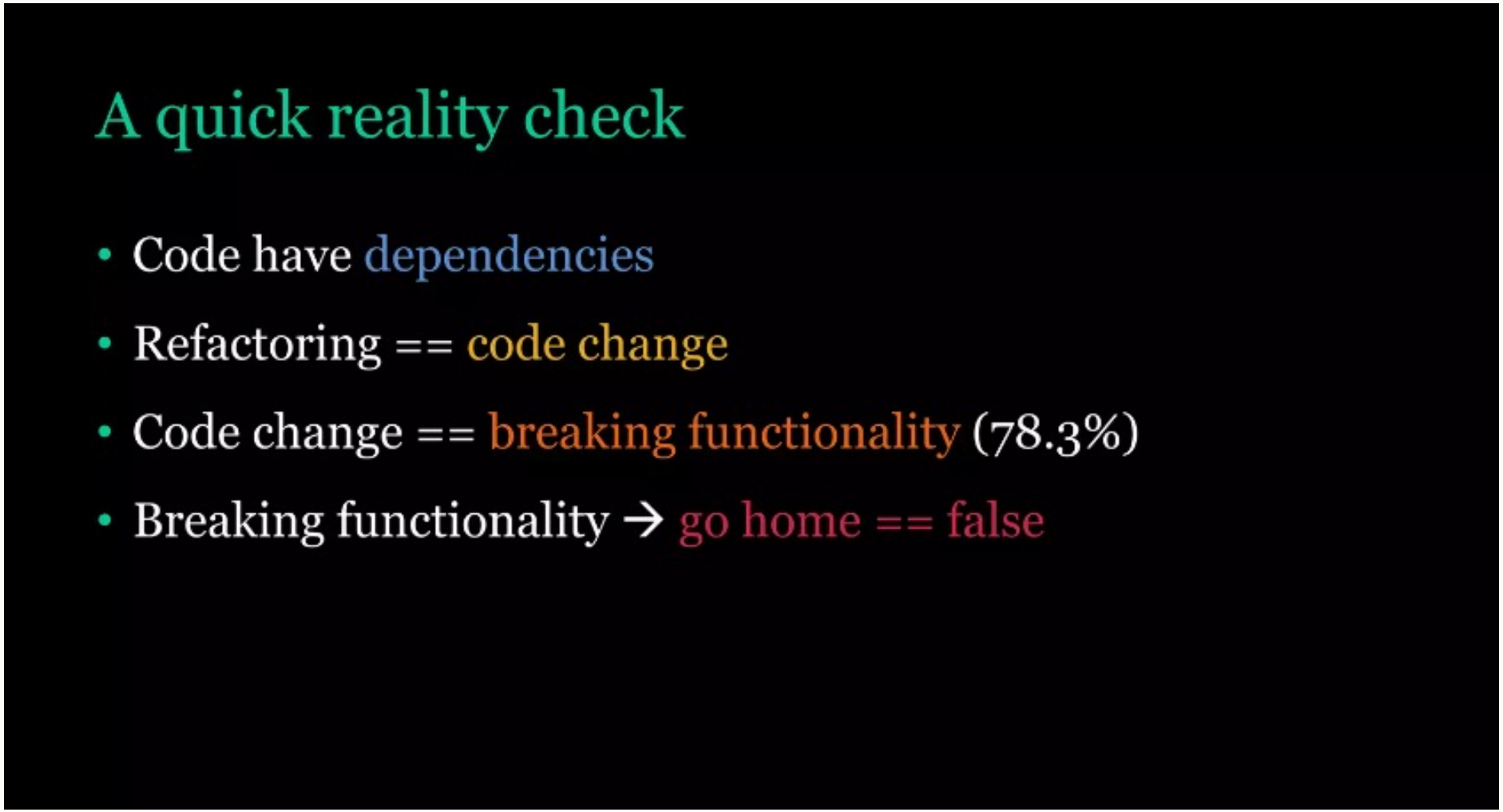
Legacy Code Maintenance The Core Of An SE Role

The Process of Working With Legacy Code



Still confused? Try this: https://www.slideshare.net/dhelper/working-with-c-legacy-code

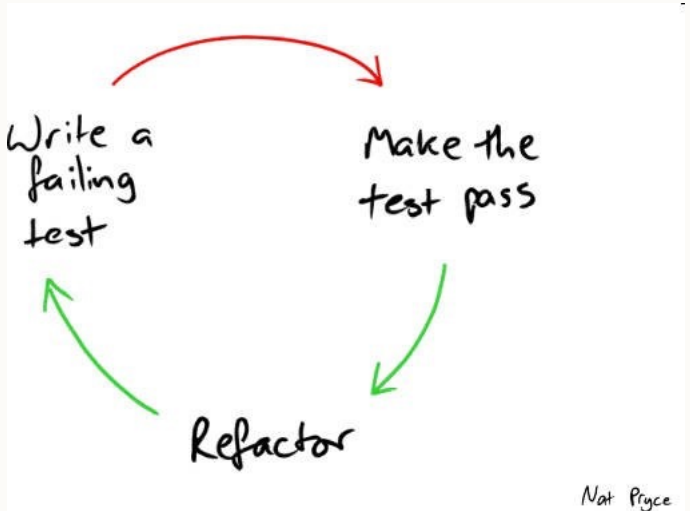
Reality Reality check!

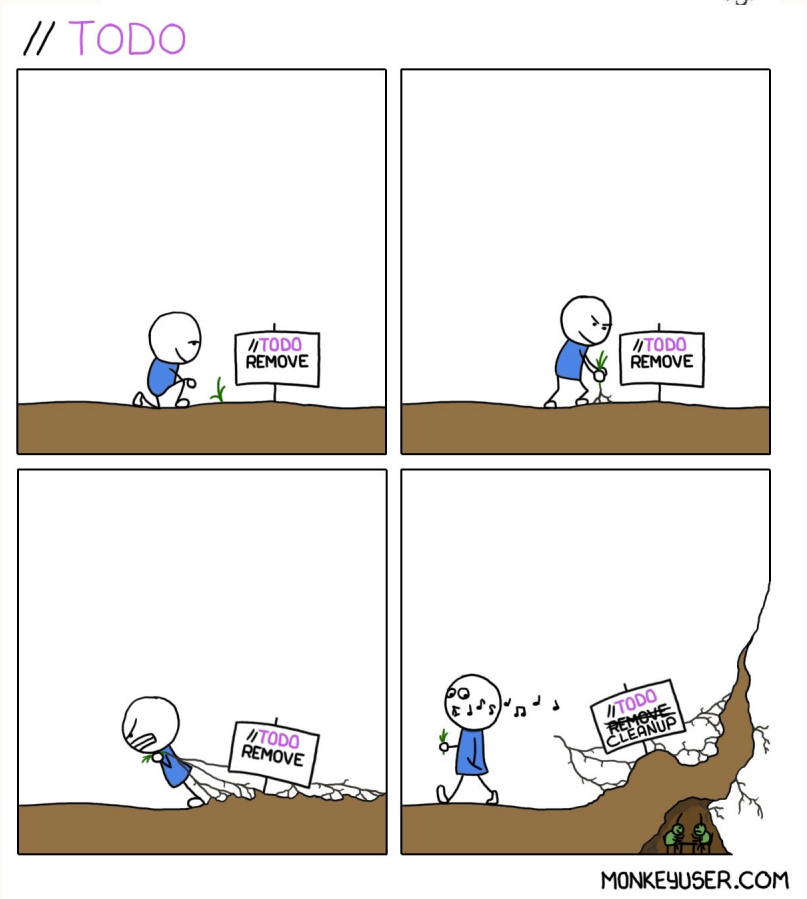


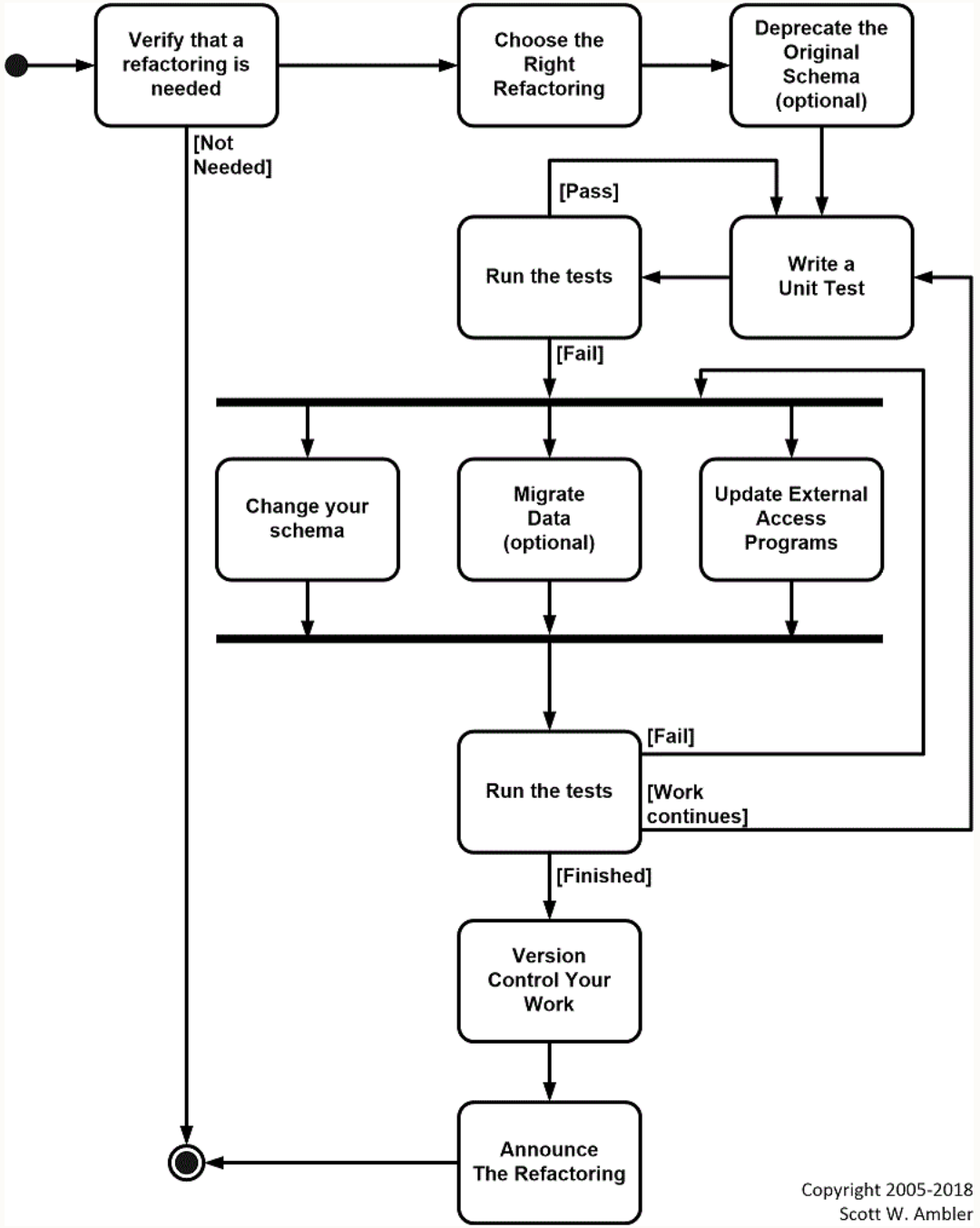
Source : https://www.slideshare.net/dhelper/working-with-c-legacy-code

Refactoring

Easy in theory







Quotes about refactoring from the authorities

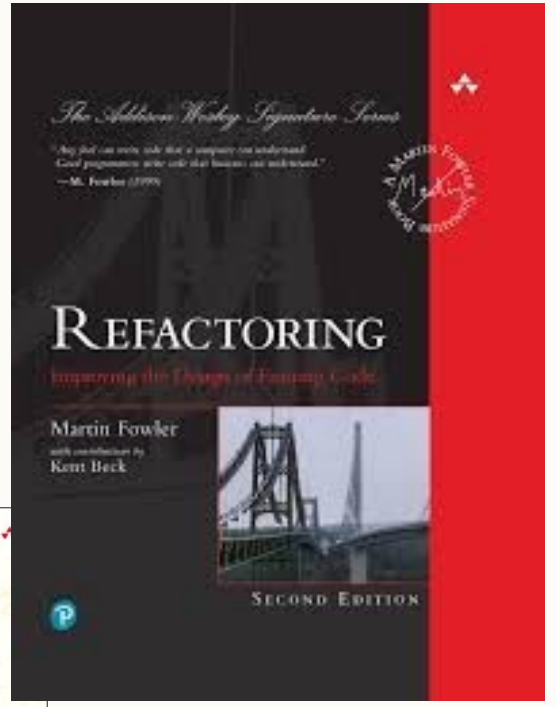
- *"A series of small decisions and actions all made through the filter of a set of values and the desire to make something excellent"* – Chad Fowler, Ruby Central

- *"Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure"* – Martin Fowler

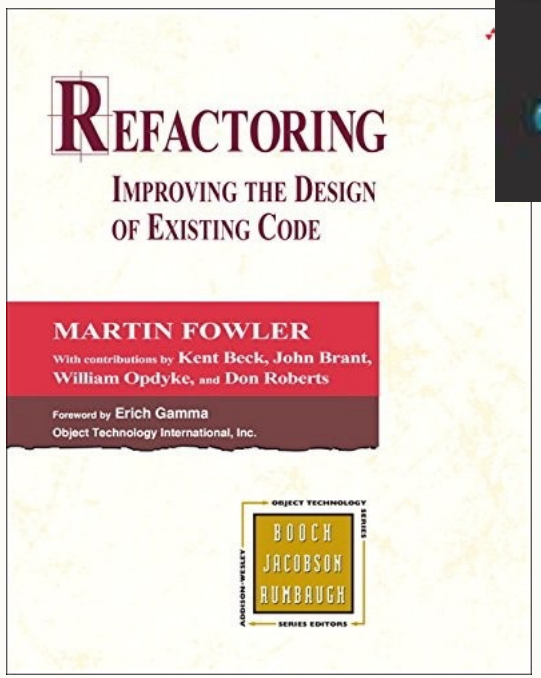
- *"The XP philosophy is to start where you are now and move towards the ideal. From where you are now, could you improve a little bit?"* - Kent Beck

- *"Any fool can write code that a computer can understand. Good programmers can write code that humans can understand"* – Martin Fowler

Martin Fowler
Refactoring Godfather



Kent Beck
Agile Guru &
Code Smeller



Refactoring improves existing code

- The process of changing a software system to not alter the external behavior of the code yet improves its internal structure in an incremental fashion
- A disciplined way to clean up code to minimize the chance of introducing bugs
- Improving the design of code after it has been written
  - Can be code you wrote yesterday or code written decades ago
- Explores the tradeoff space between clarity and performance
- Refactoring is not optimizing code to make it run faster
  - It is about making code make more sense and increasing its robustness
  - Making code open for modification
  - Application of SOLID principles and design pattern

Kent Beck Says...

"The majority of the cost of software is incurred after the software has been first deployed. Thinking about my experience of modifying code, I see that I spend much more time reading the existing code than I do writing new code. If I want to make my code cheap, therefore, I should make it easy to read."

```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

A Trivial HTML Example Of Using Whitespace in Code

```
<html> <head> <title>Make your HTML code readable by using white space</title> </head> <body> <p>Do you want an easy way to understand your code? If yes, use white space to organize your code. You can use white space as much as possible to make your code readable. You will be surprised how easy it is to follow your code when you use appropriate amount of white space. If you do not use white space, your HTML code will look messy. So: </p> <ul> <li>group related tags and separate them by indenting</li> <li>use white space</li> <li>use comments when necessary to explain what your code does</li> </ul> </body> </html>
```

```
<html>
  <head>
    <title>
      Make your HTML code readable by using white space
    </title>
  </head>
  <body>
    <p>
      Do you want an easy way to understand your code? If yes, use white space to organize your code. You can use white space as much as possible to make your code readable. You will be surprised how easy it is to follow your code when you use appropriate amount of white space. If you do not use white space, your HTML code will look messy. So:
    </p>
    <ul>
      <li>group related tags and separate them by indenting</li>
      <li>use white space</li>
      <li>use comments when necessary to explain what your code does</li>
    </ul>
  </body>
</html>
```

Who is involved in Refactoring?

- Refactoring is a multidisciplinary activity involving:
  - Programmers/Developers
  - Senior designers
  - Management
  - System Architects
- Each of these roles will adapt the principles of refactoring to a specific project or workspace
- Refactoring occurs in different forms across a project

Refactoring Roadmap

Quick Wins

- Remove dead code
- Remove code duplicates
- Enhance identifier naming
- Reduce method size

Divide & Conquer

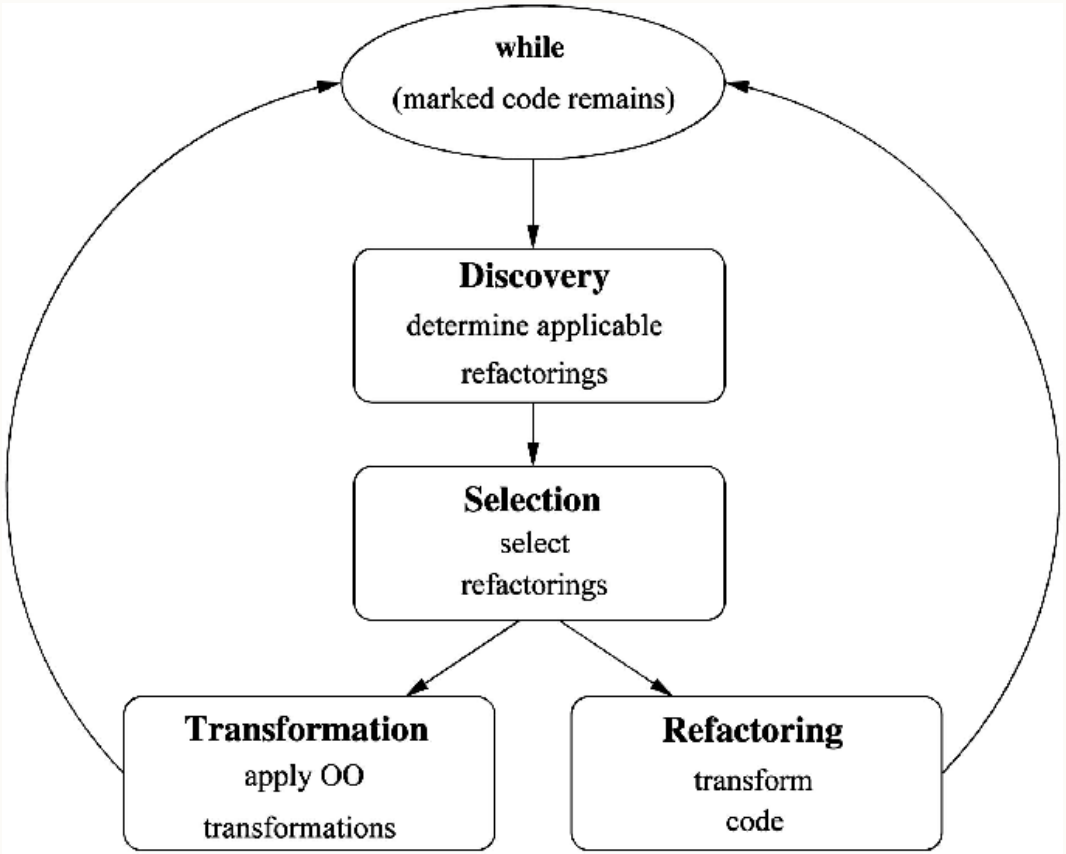
- Discover & split code into components
- Enhance component encapsulation
- Reduce coupling

Inject Quality In

- Cover components with automated tests
- Enhance components internal design

Continuous Review

Sustainable Refactoring Roadmap - v1.2



Build Your Test Scaffold Before You Touch Anything

- The key to successful refactoring is to AVOID breaking anything while you make your changes
- This requires that you use the existing or develop a new test scaffold
  - Regression testing, right?
- Run the test scaffold after every instance of change, not after every feature refactored
- Legacy code may not include any test cases, in which case you need to add them before you start ‘tinkering’ i.e. making incremental changes
- Substitute ‘live’ components with dummy ones i.e. use ‘Mock Objects’
- Once you have done this you are ready to go
- Still Confused? Try this: https://martinfowler.com/articles/mocksArentStubs.html

Identify Code Smells

Decide what you are going to refactor

Code Smells Because Its Hard to Understand (not because it is incorrect)

- Duplicated Code:
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- Long Methods:
  - long methods are more difficult to understand
  - Note: performance concerns with respect to lots of short methods are largely unfounded
- Large Classes:
  - classes try to do too much, which reduces cohesion (violates single responsibility, “The Blob”
- Long Parameter Lists:
  - hard to understand, can become inconsistent
- Divergent Changes:
  - related to cohesion where one type of change requires changing one subset of methods; another type of change requires changing another subset

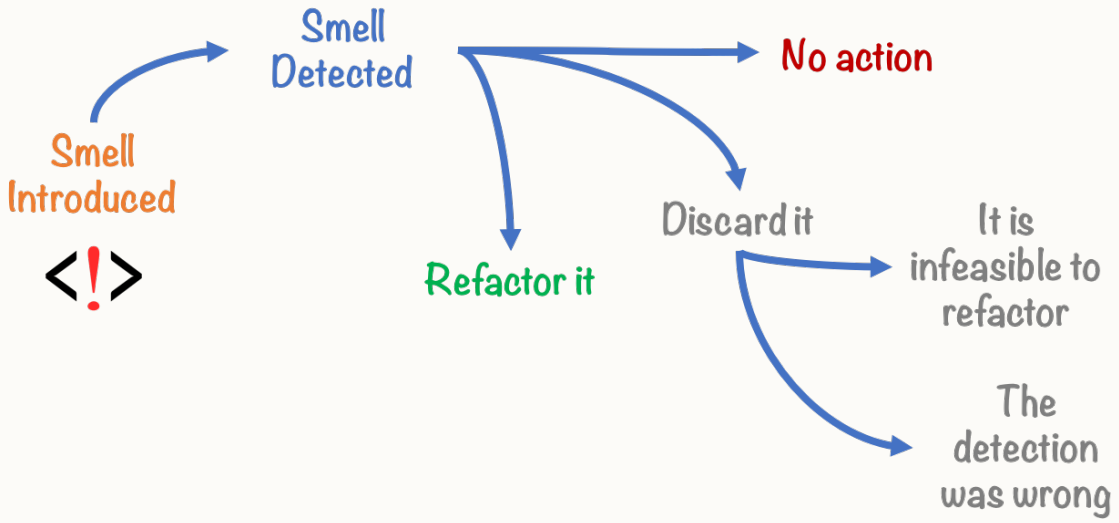
More code smells examples

- Lazy Classes:
  - A class that no longer “pays its way”, part formed functionality
- Speculative Generality:
  - “Oh I think we need the ability to do this kind of thing someday”
- Temporary Field:
  - An attribute of an object is only set in certain circumstances; but an object should need all of its attributes
- Data Class

  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- Refused Bequest
  - A subclass ignores most of the functionality provided by its superclass where the subclass may not pass the “IS-A” test (Liskov Substitution Violation)
- Comment-based coverup
  - Comments are sometimes used to hide bad code, symptomatic of large blocks of comments, not using automatic comment generation

Still Confused? Try This:

- Many more smells at: http://c2.com/cgi/wiki?CodeSmell
- Matching up a smell to its refactoring counterpart: http://wiki.java.net/bin/view/People/SmellsToRefactorings
- Taxonomy of code smells: http://www.tusharma.in/smells/
- An easy read: https://medium.com/@tusharma/how-to-track-code-smells-effectively-48dbf5ba659d



Refactoring Code

Shamelessly Inspired By Refactoring (Fowler), Chapters 6-10

Fowler Defined Different Categories of Refactoring

- *Composing Methods:* refactoring within a method or within an existing class
- *Moving Features Between Objects:* refactoring changing the responsibility of a class
- *Organizing Data:* refactoring to improve data structures and object linking
- *Simplifying Conditional Expressions:* encapsulation of conditions by replacing with polymorphism
- *Making Method Calls Simpler:* refactorings that make interfaces simpler
- *Dealing with Generalization:* moving methods up and down the hierarchy of inheritance by (often) applying the Factory or Template design pattern
- *Big Refactorings:* architectural refactoring to promote loose coupling or to realise a redesign via object orientation (Note: this is extremely difficult for most projects).

Refactoring reduces repetition and helps focus on objects

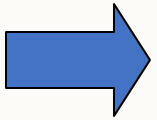
- Adding a new animal type, such as Amphibian, does not require revising and recompiling existing code
  - We have provided good extensibility which is less likely to break future code
- Mammals, birds, and reptiles are likely to differ in other ways, and we've already separated them out
  - Therefore we won't need more switchstatements in future
- Eliminated the 'flags' we needed to tell one kind of animal from another i.e. elegantly removed MAMMAL=0 etc.
- We now use Objects the way they were meant to be used and give our code better structure

Encapsulate Fields to retain private variables

- Un-encapsulated data is a violation of key object-oriented principles
- Use property get and set procedures to provide public access to private (encapsulated) member variables

```
public class Course
{
 public List students;

}
```

```
int classSize =
course.students.size();
```

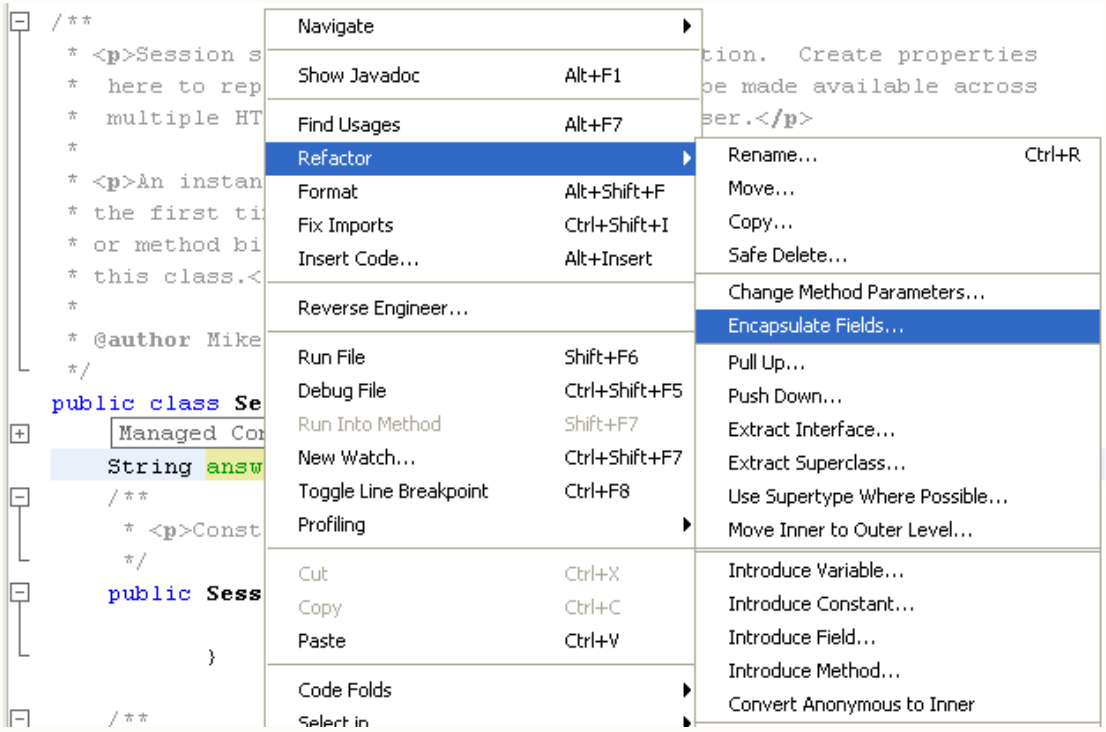


```
public class Course
{
 private List students;
 public List getStudents()
 {
    return students;
 }
 public void setStudents(List s)
 {
     students = s;
  }
}
```

```
int classSize = course.getStudents().size();
```

Encapsulating Fields Automatically With IDEs

- I have a class with 10 fields, its quite an overhead to manage and is a pain to have to go through to add get_() and set_() for each one
- Refactoring Tools
  - See NetBeans/Eclipse/Visual Studio/ Monodevelop's refactoring tools
- Also allows for automating two other refactoring tasks
  - Rename Method: cascading the new name of the method
  - Change Method Parameters: auto update when parameters are modified



Extract method from a larger block of code

- If a method is too long or needs excessive code to describe its function, then you can probably extract a new method out of the code
- Short, well named methods are easier to maintain
- Obeys the single responsibility principle
- How big is too big?
  - Used to be around 125 LOC in Fortran
  - Then we have 80 lines of code (from the size of the UNIX terminal)
- Length is not the issue – its down to “the semantic distance between a method name and the method body

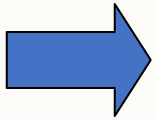
Create a new method to perform the extract method

1. Create a new method and name it after the *intention* of the method
   – If you can't derive this name, you probably should not be extracting the method!
2. Copy the extracted code from the source method into the new method
3. Scan the extracted code for references to local and temporary variables
4. Check if any of the local variables are modified by the extracted code
   – See if this can be transformed into a query instead using the Replace Temp with Query refactor (more on this later)
5. Pass into new method as parameters any required local scope variables
6. Replace the extracted code with a method call to the new method, checking the use of temp vars
7. Run your regression tests

Using Extract Method is a quick win!

- What are the code smells? See how the comment becomes the method name…

```
public class Customer
{
  void int foo()
  {
     …
     // Compute score
    score = a*b+c;
    score *= xfactor;
  }
}
```

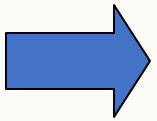


```
public class Customer
{
  void int foo()
  {
     …
    score = ComputeScore(a,b,c,xfactor);
  }

  int ComputeScore(int a, int b, int c, int x)
  {
     return (a*b+c)*x;
  }
}
```

Replace Parameter with Explicit Method

- You have a method that runs different code depending on the values of an enumerated parameter. Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
  if (name.equals("height")) {
   height = value;
   return;
  }
  if (name.equals("width")) {
    width = value;
    return;
  }
  Assert.shouldNeverReachHere();
}
```

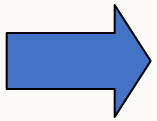


```
void setHeight(int arg)
{
    height = arg;
}

void setWidth (int arg)
{
    width = arg;
}
```

Inline Method to reduce indirection

- The method does in the body precisely the method intention

```
class PizzaDelivery
{
          // ...
    int getRating(){
        return moreThanFiveLateDeliveries() ? 2 : 1;
    }
    boolean moreThanFiveLateDeliveries(){
        return numberOfLateDeliveries > 5;
    }
}
```



```
class PizzaDelivery
{
 // ...
    int getRating() {
        return numberOfLateDeliveries
                    > 5 ? 2 : 1;
    }
}
```

Replace Temp with Query

- You place the result of an expression in a local variable for later use in your code??
- Move the entire expression to a separate method and return the result from it.
- Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.

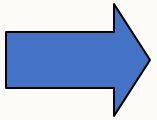
```
double calculateTotal(){
   double basePrice = quantity * itemPrice;
   if (basePrice > 1000){
      return basePrice * 0.95;
   }
   else
      return basePrice * 0.98;
}
```

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

Rename Variable or Method To Self Document

- Perhaps one of the simplest, but one of the most useful that bears repeating: If the name of a method or variable does not reveal its purpose then change the name of the method or variable.

```
public class Customer
{
   public double getinvcdtlmt();
}
```



```
public class Customer
{
   public double getInvoiceCreditLimit();
}
```

Remove assignments to parameters

- Some value is assigned to a parameter inside method's body, therefore use a local variable instead of a parameter

- If a parameter is passed by reference, then after the parameter value is changed inside the method, this value is passed to the argument that requested calling this methods
  - This occurs accidentally and leads to unfortunate effects

  - Even if parameters are usually passed by value (and not by reference) in your programming language, this coding quirk may alienate those who are unaccustomed to it.
- Also, multiple assignments of different values to a single parameter make it difficult for you to know what data should be contained in the parameter at any particular point in time, makes testing harder too

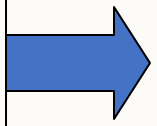
Remove assignments to parameters

- Each element of the program should be responsible for only one thing. This makes code maintenance much easier going forward, since you can safely replace code without any side effects
- This refactoring helps to extract repetitive code to separate methods

Remove assignments to parameters

- Create a local variable and assign the initial value of your parameter
- In all method code that follows this line, replace the parameter with your new local variable

```
int discount (int inputVal, int quantity)
{
        if (inputVal > 50)
        {
                inputVal -= 2;
        }
 // ...
}
```

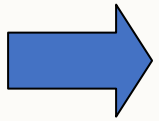


```
int discount (int inputVal, int quantity)
{
        int result = inputVal;
        if (inputVal > 50)
        {
                result -= 2;
        }
 // ...
}
```

Replace Magic Number with Symbolic Content

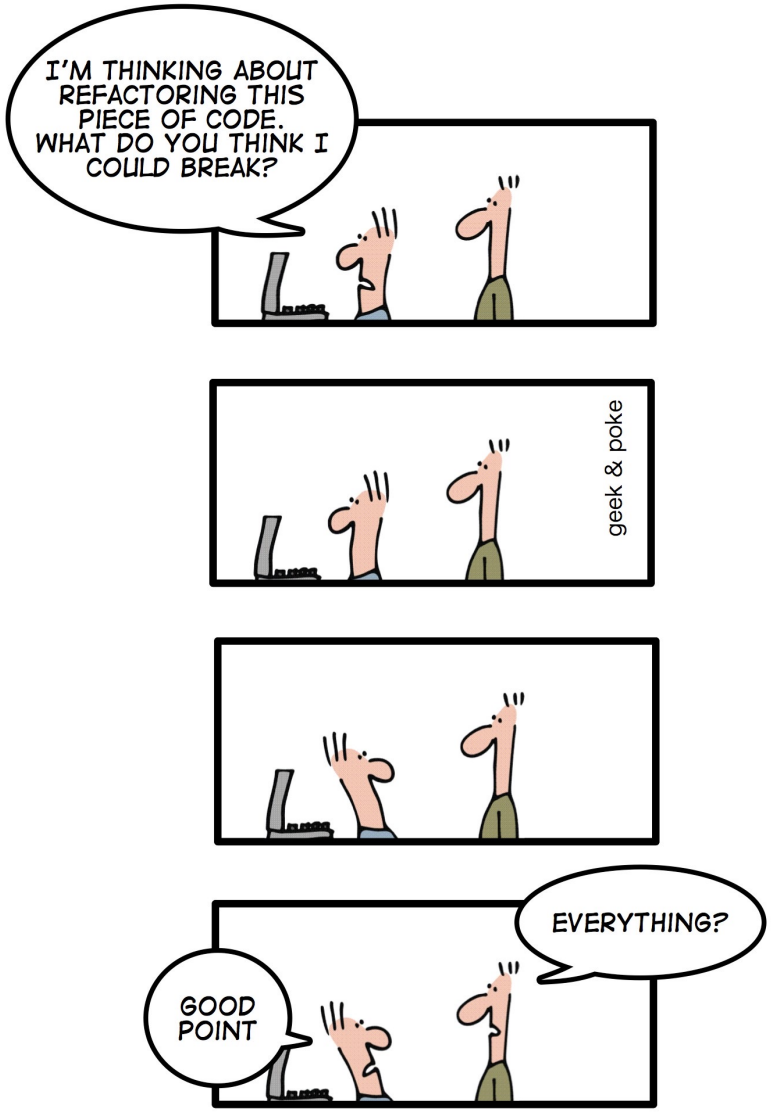
- Used if you have a literal number with a particular meaning
- Create a constant, name it after the meaning and replace the number with it
- This works if this is a universal constant in your code! (global variables are bad)

```
double energy (double mass, double height)
{
        return mass*height*9.81;
 // ...
}
```



```
static final double GRAVITY = 9.81;
double potentialEnergy(double mass,
double height) {
        // …
        return mass * height * GRAVITY;
 }
```

Still confused? Try this: https://www.youtube.com/watch?v=owFFVQYW1p8


I'M THINKING ABOUT REFACTORING THIS PIECE OF CODE. WHAT DO YOU THINK I COULD BREAK?
geek & poke
EVERYTHING?
GOOD POINT


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

Refactoring at a higher Level

Designing classes and really refactoring to be clean and object oriented

Replace type code with polymorphism

- **Switch** statements are very rare in properly designed object-oriented code, much more common in procedural programs
  - 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
  - Lengthy to test all of the test cases
- There are several well-defined refactorings for this case
  - The simplest is the creation of subclasses

Replace type code with polymorphism

```
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 “skin";
        }
    }
}
```

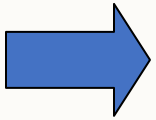
Improvement over switch using `extends`


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

Single Responsibility with Extract Class

- When one class does the job of two, it promotes tight coupling
- Break one class into two, e.g. Having the phone details as part of the Customer class is not a realistic OO model, and also breaks the Single Responsibility design principle. We can refactor this into two separate classes, each with the appropriate responsibility.

```
public class Customer
{
 private String name;
 private String workPhoneAreaCode;
 private String workPhoneNumber;
}
```



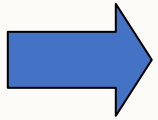
```
public class Customer
{
 private String name;
 Private Phone workPhone;
}

public class Phone
{
 private String areaCode;
 private String number;
}
```

Generalization with Extract Interface

- Extract an interface from a class. Some clients may need to know a Customer's name, while others may only need to know that certain objects can be serialized to XML. Having toXml() as part of the Customer interface breaks the Interface Segregation design principle which tells us that it's better to have more specialized interfaces than to have one multi-purpose interface.
- How to do this:
  1. Create an empty interface.
  2. Declare common operations in the interface.
  3. Declare the necessary classes as implementing the interface.
  4. Change type declarations in the client code to use the new interface.

Extract Interface

```
public class Customer
{
   private String name;

   public String getName(){ return name;}

   public void setName(String string)
   { name = string; }

   public String toXML()
   { return "<Customer><Name>" +
       name + "</Name></Customer>";
   }
}
```



```
public class Customer implements SerXML
{
   private String name;

   public String getName(){ return name; }

   public void setName(String string)
   { name = string; }

   public String toXML()
   { return "<Customer><Name>" +
       name + "</Name></Customer>";
   }
}
```

```
public interface SerXml {
   public abstract String toXML();
}
```

Push Down/Pull Up of methods

- Eliminating duplicate behavior is a core value of refactoring
  - “breeding ground” for bugs
  - Whenever there is duplication there is the risk that you will alter one instance and not the other one
- ‘Pull Up’ when you have methods of the same composition
  - Often after you have refactored a bit to reduce the temp variables
  - Use when you have a subclass which overrides a superclass method yet does the same thing or after applying Substitute Algorithm to make them identical
  - If you have two methods which are similar but not identical use ‘Form Template’ method
- Also applies to the refactoring of a module hierarchy where a method is duplicated on two or more classes, moving the method up the hierarchy
- Push down is the opposite, to specialize a class to stop a method implemented in a superclass which is only used by one or a small number of subclasses

Extras: Move Method - Before

- If a method on one class uses (or is used by) another class more than the class on which its defined, move it to the other class

```
public class Student
{
  public boolean isTaking(Course course)
  {
     return (course.getStudents().contains(this));
   }
}

public class Course
{
  private List students;  public
  List getStudents()
  {
    return students;
  }
}
```

Extras: Move Method - Refactored

- The student class now no longer needs to know about the Course interface, and the isTaking() method is closer to the data on which it relies - making the design of Course more cohesive and the overall design more loosely coupled

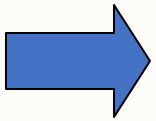
```
public class Student
{
}

public class Course
{
  private List students;
  public boolean isTaking(Student student)
  {
     return students.contains(student);
  }
}
```

Extras: Introduce Null Object

- If relying on null for default behavior, use inheritance instead

```
public class User
{
  Plan getPlan()
  {
      return plan;
  }
}
```

```
if (user == null)
    plan = Plan.basic();
else
   plan = user.getPlan();
```

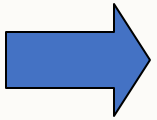


```
public class User
{
  Plan getPlan()
  {
    return plan;
  }
}

public class NullUser extends User
{
  Plan getPlan()
  {
     return Plan.basic();
  }
}
```

Extras: Replace Error Code with Exception

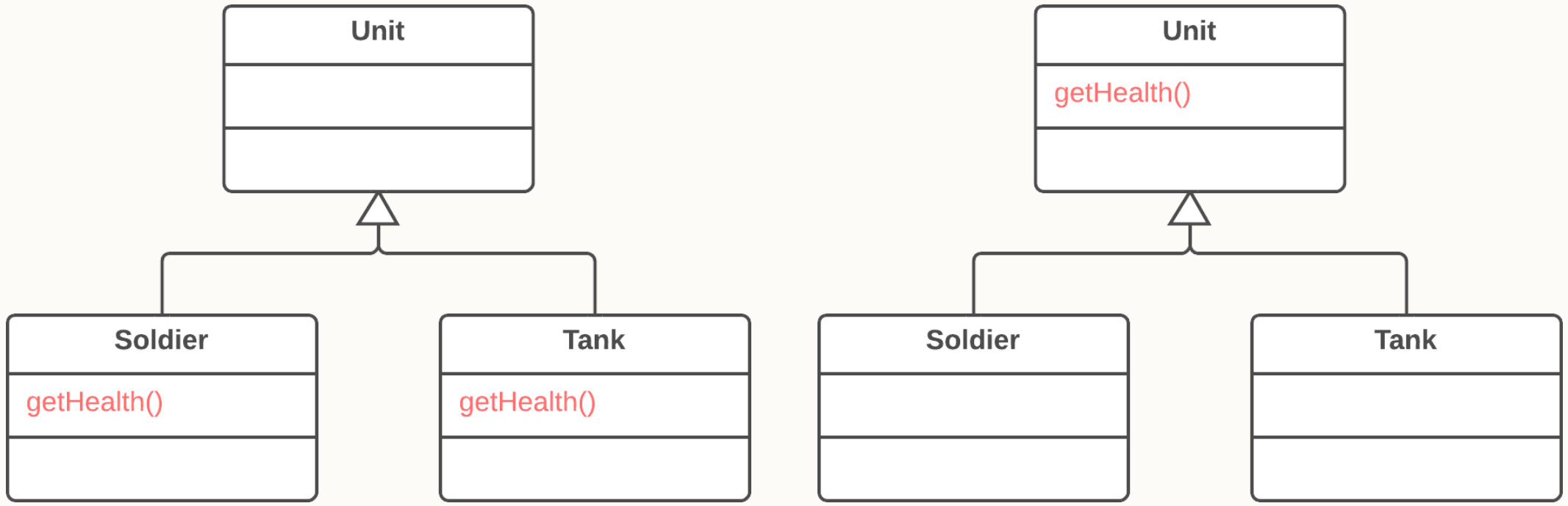
- A method returns a special code to indicate an error is better accomplished with an Exception.

```
int withdraw(int amount)
{
  if (amount > balance)
                return -1;
  else {
                balance -= amount;
                return 0;
        }
}
```

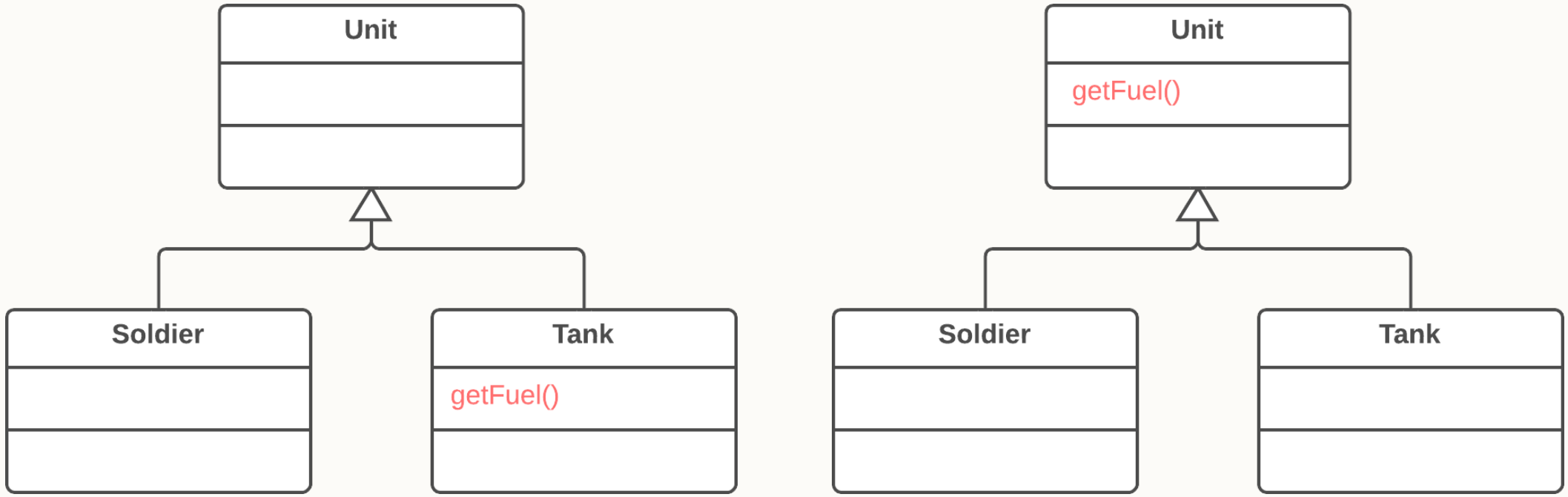


```
void withdraw(int amount)
   throws BalanceException
{
  if (amount > balance)
  {
                throw new BalanceException();
   }
  balance -= amount;
}
```

Pull Up

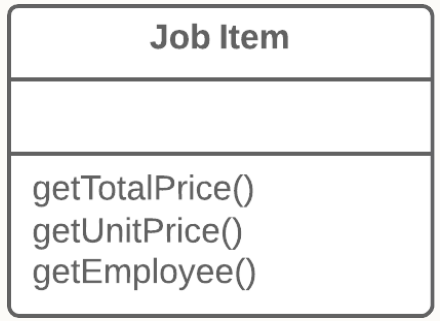

Unit
Soldier
getHealth()
Tank
getHealth()
Unit
getHealth()
Soldier
Tank

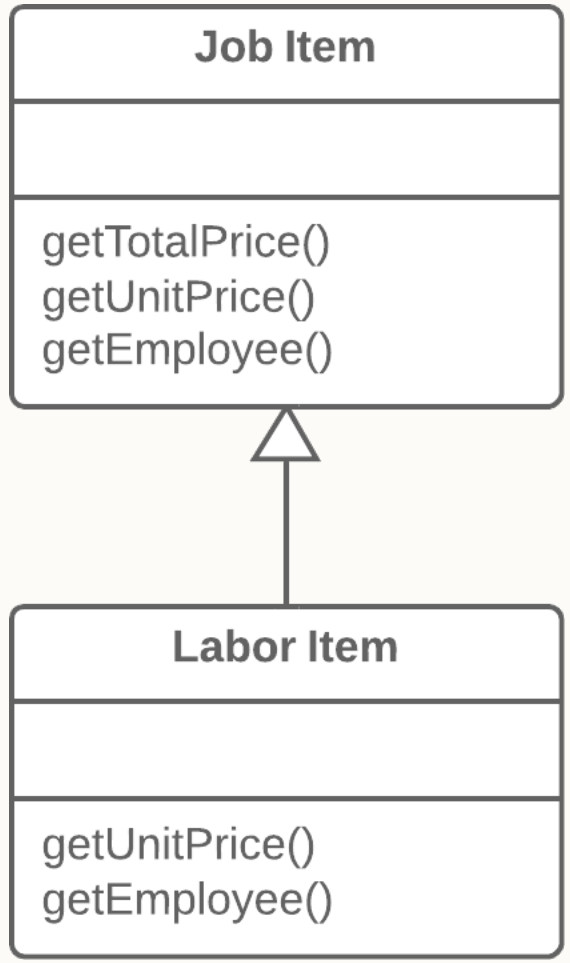
Push Down



You can inherit if you need to with Extract Subclass

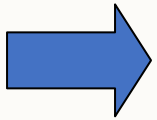
- You have found a class has features (attributes and methods) that would only be useful in specialized instances, we can create a specialization of that class and give it those features
- This makes the original class less specialized (i.e., more abstract), and good design is about binding to abstractions wherever possible
- Caution – too much inheritance can promote tight coupling, like making a large collection of subclasses you may want to avoid





Extract subclass example

```
public class Person
{
 private String name;
 private String jobTitle;
}
```

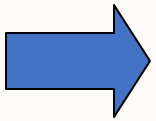


```
public class Person
{
 protected String name;
}

public class Employee extends Person
{
 private String jobTitle;
}
```

You may also want to Extract Super Classes

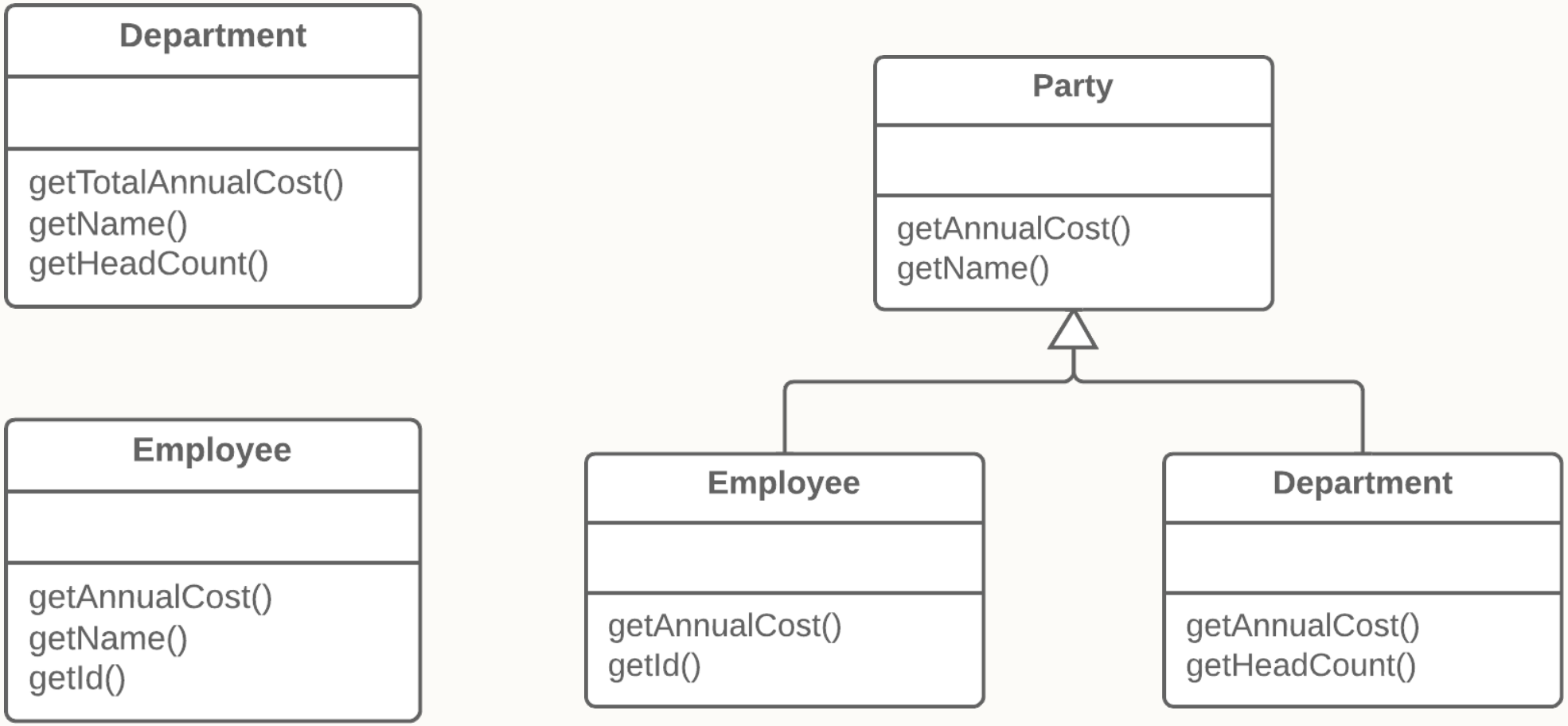
- When you find two or more classes that share common features, consider abstracting those shared features into a super-class
- This makes it easier to produce an abstraction, and removes duplicate code from the original classes
- Not used if superclass already exists

```
public class Employee
{
  private String name;
  private String jobTitle;
}

public class Student
{
  private String name;
  private Course course;
}
```



```
public abstract class Person
{
  protected String name;
}

public class Employee extends Person
{
  private String jobTitle;
}

public class Student extends Person
{
  private Course course;
}
```

Calculating annual cost examples



Applying a design pattern in Refactoring

The Form Template Method

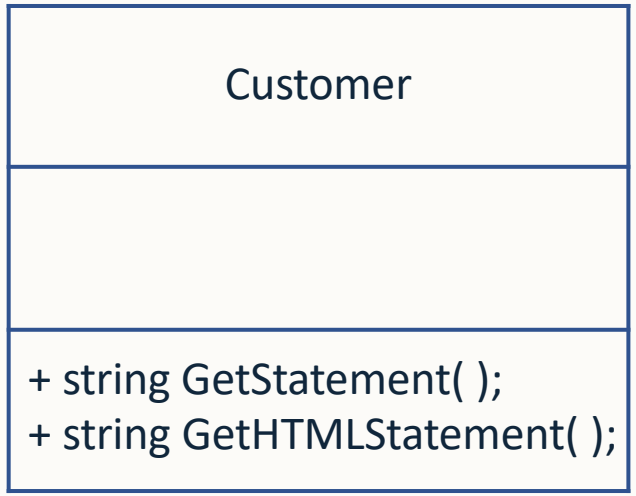
Form Template Method adds consistencies to methods so that 'pull up method' can be performed

- When you find two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class
- Subclasses are developed in parallel, sometimes by different people, which leads to code duplication, errors, and difficulties in code maintenance
- Each change must be made in all subclasses – *why is this problematic?*
- Code duplication doesn't always refer to cases of simple copy/paste
  - Often duplication occurs at a higher level, such as when you have a method for sorting numbers and a method for sorting object collections that are differentiated only by the comparison of elements
- Creating a template method eliminates this duplication by merging the shared algorithm steps in a superclass and leaving just the differences in the subclasses

Fowler's Famous Video Rental Example

- In this explanation we make use of the classic example described in the first chapter of Fowler's refactoring [available in Java, Ruby and Javascript]
- Describes a management system for a video rental store –*yes I know that such place don't exist any more… indeed Fowler has changed his recent examples accordingly to involve a theatre company management system.*
- It contains a Movie class, a Customer class and a Rental class, and prints out statements
- Performs a series of refactoring tasks on this original codebase
- Still confused? Here is Martin Fowler himself talking through his example in full using Javascript: https://martinfowler.com/articles/refactoring-video-store-js/
- Even more confused? Here's Fowler's lecture: https://www.youtube.com/watch?v=6wDoopbtEqk&t=1263s

Customer Rental System Before Refactoring




```
public string GetStatement()
{
        var result = new StringBuilder();
        result.AppendLine("Rental record: " +Name);
        foreach (var rental in Rentals)
                result.AppendLine("\t" + rental.Movie.Title);
        result.AppendLine("Owed:" + TotalCharge.ToString());
        result.AppendLine("Customer earned:" +
TotalFrequentRenterPoints.ToString()+ "renter points" );
        return result.ToString();
}
```

GetHTMLStatement() Is Surprisingly Similar…

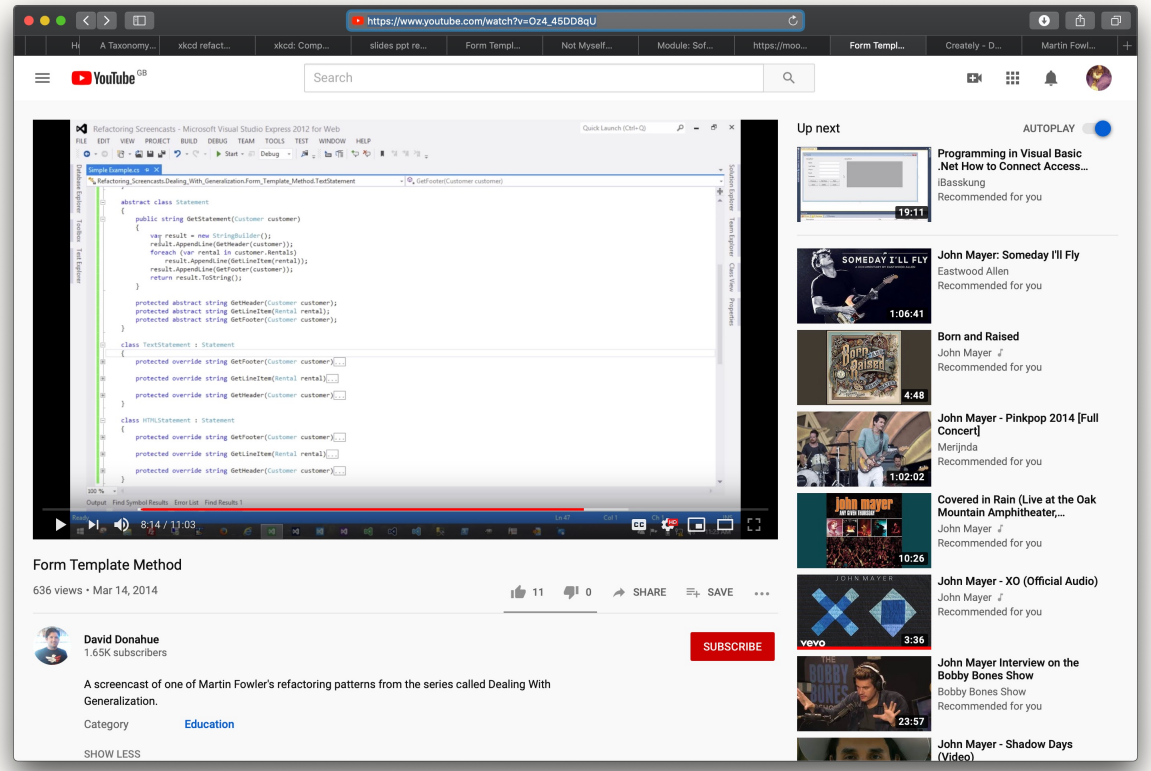
```
public string GetHTMLStatement()
{
        var result = new StringBuilder();
        result.AppendLine("<h1>Rental record:<em> " + Name + "</em></h1>");
        foreach (var rental in Rentals)
                result.AppendLine(rental.Movie.Title + "<br />");
        result.AppendLine("<p> Owed: <em>" + TotalCharge.ToString() + "<em></p>");
        result.AppendLine("<p> Customer earned: <em> " +
                        TotalFrequentRenterPoints.ToString()+ "</em>renter
points</p>");
        return result.ToString();
}
```

Mechanics of the Form Template Method

1. Decompose the methods so that the extracted methods are identical
2. Apply Pull Up of the identical methods into the the superclass (if using inheritance) or the class that extends the module
3. Test after each pull up/ extraction
4. Use the same set of method calls but the subclasses/modules handle the method calls differently
5. Test again
6. Apply Pull Up to the original method
7. Test again then remove the extraneous methods, test after each removal

See how it is done, by David Donahue

```
abstract class Statement
{
    public string GetStatement(Customer customer)
    {
        var result = new StringBuilder();
        result.AppendLine(GetHeader(customer));
        foreach (var rental in customer.Rentals)
            result.AppendLine(GetLineItem(rental));
        result.AppendLine(GetFooter(customer));
        return result.ToString();
    }

    protected abstract string GetHeader(Customer customer);
    protected abstract string GetLineItem(Rental rental);
    protected abstract string GetFooter(Customer customer);
}

class TextStatement : Statement
{
    protected override string GetFooter(Customer customer)...

    protected override string GetLineItem(Rental rental)...

    protected override string GetHeader(Customer customer)...
}

class HTMLStatement : Statement
{
    protected override string GetFooter(Customer customer)...

    protected override string GetLineItem(Rental rental)...

    protected override string GetHeader(Customer customer)...
}
```



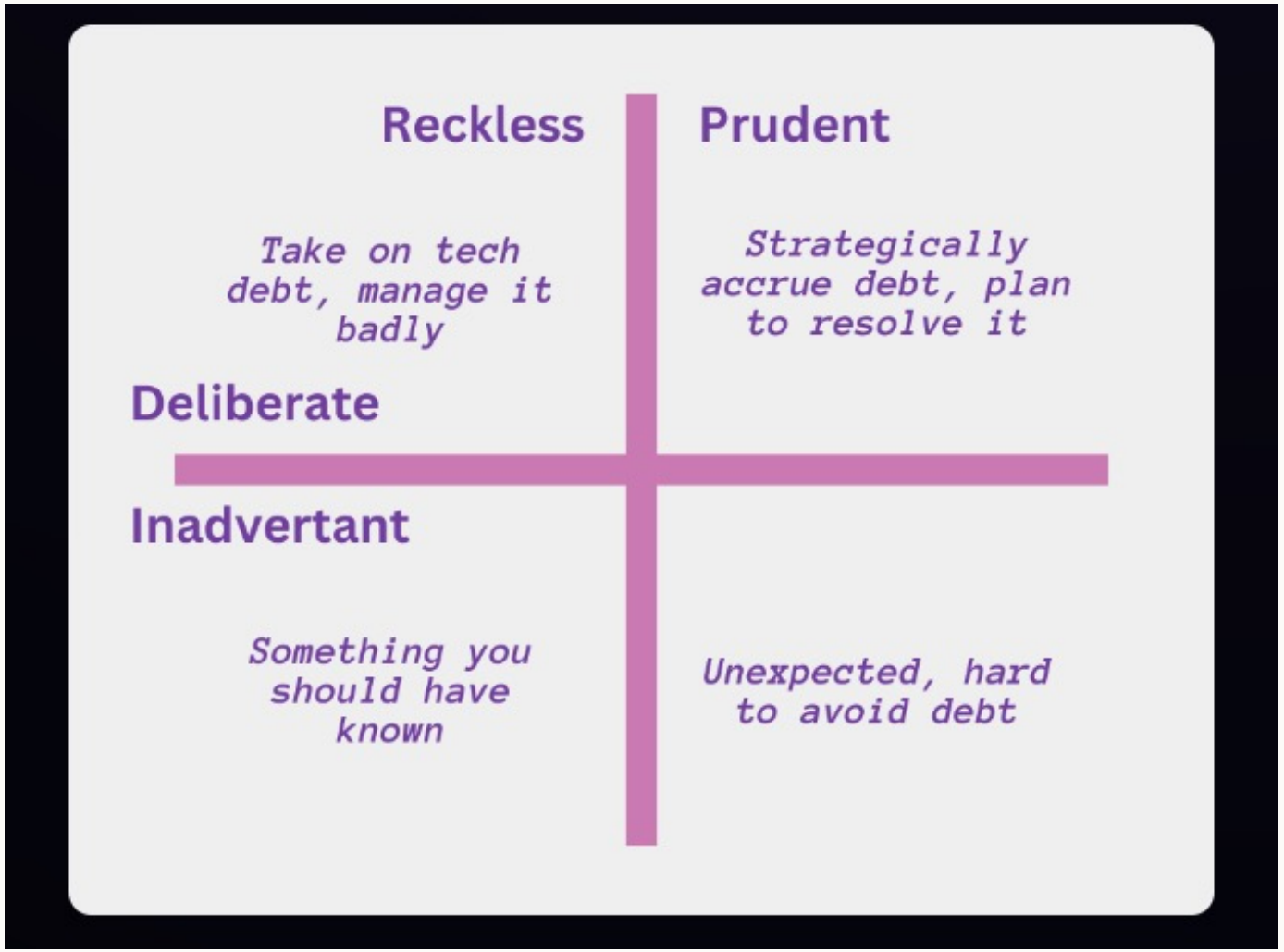
A note on ‘Big Refactoring’

- Little refactoring is about making many almost imperceptible changes to code
- Big refactoring is a team wide effort involving major redesigns
- Four Big Refactorings:
    1. Tease apart inheritance to promote encapsulation, separating out functionality
    2. Convert procedural design features to object oriented (beyond variables) by applying design patterns or SOLID principles
    3. Separate domain from presentation by for example using the MVC
    4. Extract a hierarchy by creating a hierarchy of classes with each subclass representing a specific case
- Massive risk of breaking an entire codebase if careful testing is not applied during these challenging processes – but the payoff in reducing future maintenance costs is huge: *remember, software engineering is about making money from code*

And there are many more to explore

- If you can afford it, buy a copy of Fowler's book – 1st Edition Java, 2nd Ed Javascript
- Check out Refactoring Guru where all of the techniques in the book are explained one by one: https://refactoring.guru/refactoring
- Make sure you watch the video on the Template Method

Technical Debt

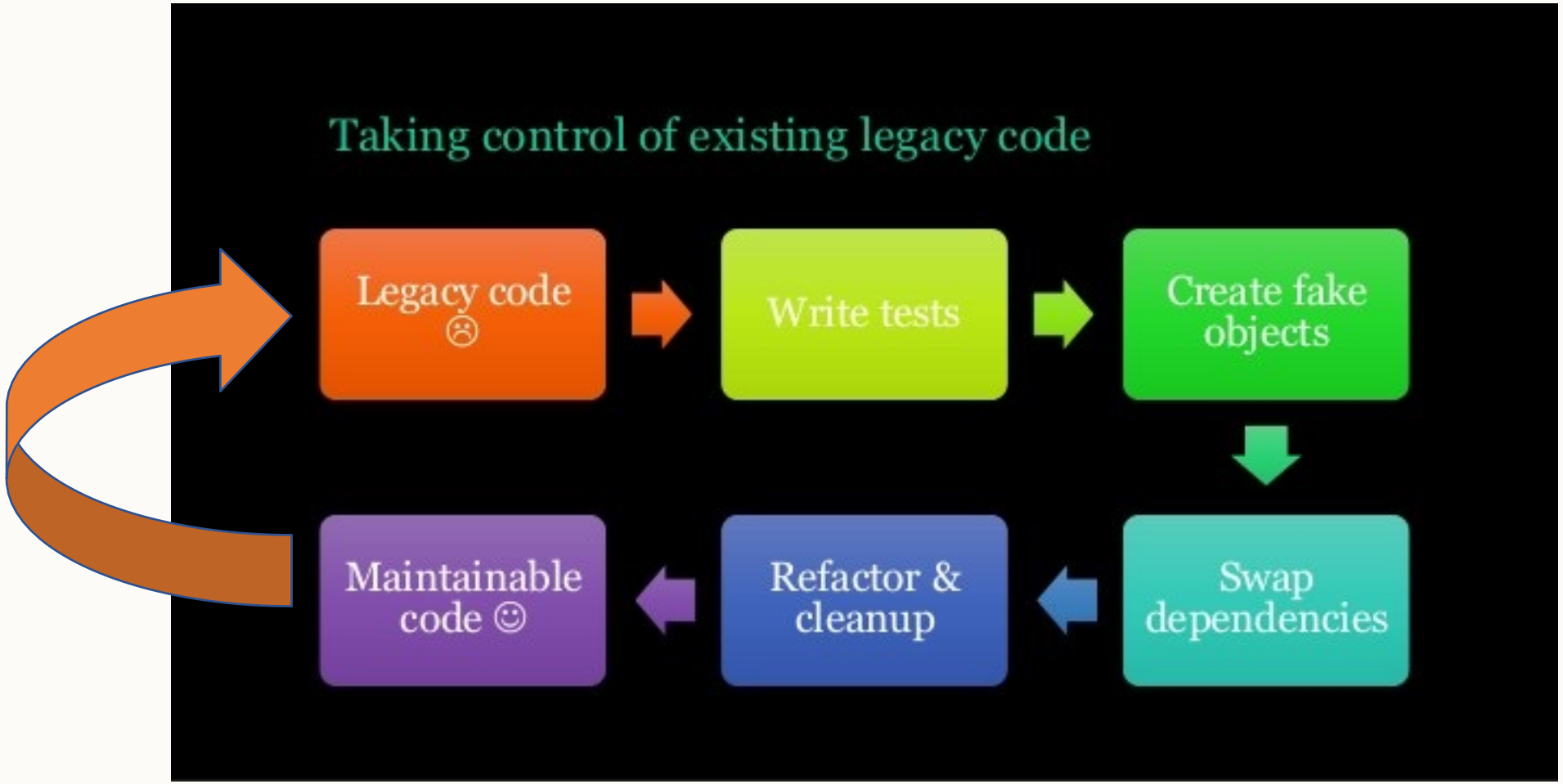


Source : https://www.stepsize.com/blog/types-of-tech-debt-with-examples-and-fixes

We are doing this to avoid technical debt

- No one sets out to write ‘bad’ code – though the agile principles of minimal design up front and release early and often teach us to focus on delivery rather than clean code
- Technical debt is the same as consumer debt, not following the practice of testing and refactoring over time means that the blob and spaghetti code build up
- You can speed up development without explicit testing but this has consequences
  - Business pressure to release new features all the time
  - Monolithic code rather than decomposed modules
  - Lack of testing and documentation
  - Lack of interaction between team members
  - Lack of compliance monitoring

As a recap, this is an iterative process



Self Study Quick Questions


Legacy code is …


Code smells are …


Refactoring is …


Testing works with refactoring because…


The extract method works by…


Replace magic numbers with…


The Form Template Method uses…

Enjoy Refactoring Week

- Refactoring and legacy code
- Principles and motivation for refactoring
- Martin Fowler and Kent Beck
- Small scale refactoring
- Object orientation and SOLID
- Applying patterns
- Many more refactoring techniques….

Today: Lecture on Refactoring

Friday 24th November : Mileston1
Coursework deadline, 5pm