Lecture 11:
Refactoring (and TDD)

Dietmar Pfahl
email: dietmar.pfahl@ut.ee
Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure I
Week 06: Development Infrastructure II
Week 07: Architecture and Design
Week 08: Verification and Validation I
Week 09: Verification and Validation II

Week 10: Continuous Development and Integration
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: no lecture
Week 14: Software Craftsmanship
Week 15: Course wrap-up, review and exam preparation
Structure of Lecture 09

- What is Refactoring and why do it?
- Examples
  - Code Smells
  - Refactorings
What is Refactoring?
What is Refactoring?
Beck’s Definition

“A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality – simplicity, flexibility, understandability, performance”

(Kent Beck, *Extreme Programming Explained*, page 179)
Fowler’s Definition

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”

(Martin Fowler, *Refactoring*, page 53)
Refactoring – Comprehensive Definition

Changes made to the system that

• Do not change observable behavior (all the tests still pass)
• Remove duplication or needless complexity
• Enhance software quality:
  • Make the code simpler and easier to understand
  • Make the code more flexible
  • Make the code easier to change
A Trivial Example

- Consolidate Duplicate Conditional Fragments (page 243); This

```java
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
```

- becomes this

```java
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

(Martin Fowler, *Refactoring*)
A Trivial Example

Consolidate Duplicate Conditional Fragments (page 243); This

```java
if (isSpecialDeal()) {
    total = price * 0.95;
    send()
} else {
    total = price * 0.98;
    send()
}
```

becomes this

```java
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Or:

```java
total = price * 0.98;
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

(Martin Fowler, *Refactoring*)
Context – Test-Driven Development (TDD)

User Story  
(new functionality)

Write a failing test.

Clean up code.

Make the test pass.
Test-Driven Development (TDD) – Example

Story1: There is a cash discount of 2%

Test1:
price = 1;
assert_equal(total, 0.98);

⇒ Test1 fails

Code1:
total = price*0.98;
send();

⇒ Test1 passes

Nothing to refactor
Test-Driven Development (TDD) – Example

Story2: If there is a special deal, cash discount is 5%

Test2:
price = 1;
if (isSpecialDeal()) {assert_equal(total, 0.95);}
Test 2 → fails

Code 2:
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
→ Test2 passes (and also Test1 passes)

Refactoring → Code3:
total = price * 0.98;
if (isSpecialDeal()) {
    total = price * 0.95;
} send();
→ Test1 & Test2 pass

User Story
(new functionality)

Write a failing test.

Clean up code.

Make the test pass.
Why Refactor?

- Prevent “design decay”
- Clean up messes in the code
- Simplify the code
- Increase readability and understandability
- Find bugs (easier)
- Reduce debugging time
- Build in learning we do about the application
- Redoing things is fundamental to every creative process
When to Refactor?

- “All the time”
- When you duplicate code ➔ Rule of Three
  - Rule of thumb to decide when a duplicated piece of code should be replaced by a new method: code can be copied once, but when the same code is used three times, it should be extracted into a new method.
- When you add functionality
- When you learn something about the code
- When you fix a bug
- When the code smells
When **not to Refactor?**

- **When the tests aren’t passing**
- **When you should just rewrite the code**
- **When you have impending deadlines**
  - Downside: unfinished refactoring becomes a technical debt

**Potential problems:**

Taken too far, refactoring can lead to incessant tinkering with the code, trying to make it perfect.

Refactoring code when the tests don’t work or tests when the application doesn’t work leads to potentially dangerous situations.

Databases can be difficult to refactor

Refactoring published interfaces can cause problems for the code that uses those interfaces
How to Refactor?

- Make sure your tests pass
- Find some code that “smells”
- Determine how to simplify this code
- Make the simplifications
- Run tests to ensure things still work correctly
- Repeat the simplify/test cycle until the smell is gone
Structure of Lecture 09

- What is Refactoring and why do it?
- Examples
  - Code Smells
  - Refactorings
Code Smells

- Indicators that something may be wrong in the code
- Can occur both in production code and test code

```java
class AbstractCollection implements Collection {
    public void addAll(Collection c) {
        if (c instanceof Set) {
            Set s = (Set)c;
            for (int i=0; i < s.size(); i++) {
                if (s.contains(s.get(i))) {
                    add(s.get(i));
                }
            }
        } else if (c instanceof List) {
            List l = (List)c;
            for (int i=0; i < l.size(); i++) {
                if (l.contains(l.get(i))) {
                    add(l.get(i));
                }
            }
        } else if (c instanceof Map) {
            Map m = (Map)c;
            for (int i=0; i < m.size(); i++) {
                add(m.key[i], m.value[i]);
            }
        }
    }
}
```
Code Smells (Fowler)

Alternative Classes with Different Interfaces
Comments
Data Class
Data Clumps
Divergent Change
Duplicated Code
Feature Envy
Inappropriate Intimacy
Incomplete Library Class
Large Class
Lazy Class

Long Method
Long Parameter List
Message Chains
Middle Man
Parallel Inheritance Hierarchies
Primitive Obsession
Refused Bequest
Shotgun Surgery
Speculative Generality
Switch Statements
Temporary Field
Code Smells (Fowler)

- **Duplicated Code**
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!

- **Long Method**
  - long methods are more difficult to understand
    - performance concerns with respect to lots of short methods are largely obsolete
Code Smells (Fowler)

- Large Class
  - Large classes try to do too much, which reduces cohesion
- Long Parameter List
  - hard to understand, can become inconsistent
- Divergent Change
  - Related to cohesion
  - symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
Code Smells (Fowler)

- **Shotgun Surgery**
  - A change requires lots of little changes in a lot of different classes

- **Feature Envy**
  - A method requires lots of information from some other class
    - move it closer!

- **Data Clumps**
  - Attributes that clump together (are used together) but are not part of the same class
Shotgun Surgery – Example:
Add logging functionality & then change it

```c
void MyFunc() {
    ... 
}

void MyFunc2() {
    ... 
}

... 

void MyFuncN() {
    ... 
}

void MyFunc() {
    printf("Entering MyFunc()\n");
    ... 
}

void MyFunc2() {
    printf("Entering MyFunc2()\n");
    ... 
}

... 

void MyFuncN() {
    printf("Entering MyFuncN()\n");
    ... 
}
```

Change the logging text (e.g., print the code line)
Code Smells (Fowler)

- **Primitive Obsession**
  - characterized by a reluctance to use classes instead of primitive data types

- **Switch Statements**
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)

- **Parallel Inheritance Hierarchies**
  - Similar to Shotgun Surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies
  - Note: some design patterns encourage the creation of parallel inheritance hierarchies (so they are not always bad!)

  (e.g., Façade design pattern → used to simplify an interface)
Code Smells (Fowler)

- **Lazy Class**
  - A class that no longer “pays its way”
  - e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out

- **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
Code Smells (Fowler)

- **Message Chains**
  - A client asks an object for another object and then asks that object for another object etc. Bad because client depends on the structure of the navigation.

- **Middle Man**
  - If a class is delegating more than half of its responsibilities to another class, do you really need it? (involves trade-offs, some design patterns encourage this (e.g. Decorator))

- **Inappropriate Intimacy**
  - Pairs of classes that know too much about each other’s private details (loss of encapsulation; change one class, the other has to change)
Code Smells (Fowler)

- **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
  - Subclass may not pass the "IS-A" test

- **Comments (!)**
  - Comments are sometimes used to hide bad code
    - "...comments often are used as a deodorant" (!)
Test Code Smells (van Deursen et al.)

- Mystery Guest
- Resource Optimism
- Test Run War
- General Fixture
- Eager Test
- Lazy Test
- Assertion Roulette
- Indirect Testing
- For Testers Only
- Sensitive Equality
- Test Code Duplication

“Refactoring Test Code” by Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok, published in *Extreme Programming Perspectives*
Structure of Lecture 09

- What is Refactoring and why do it?
- Examples
  - Code Smells
  - Refactorings
Signs of Good OO Design

- Short methods
  - Simple method logic

- Few instance variables

- Clear object responsibilities
  - State the purpose of the class in one sentence
  - No super-intelligent objects
The Catalogue of Refactorings (1)

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Duplicate Observed Data
- Dynamic Method Definition
- Eagerly Initialized Attribute
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Module
- Extract Subclass
- Extract Superclass
- Extract Surrounding Method
- Extract Variable
- Form Template Method
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- Inline Module
- Inline Temp
- Introduce Assertion
- Introduce Class Annotation
- Introduce Expression Builder
- Introduce Foreign Method
- Introduce Gateway
- Introduce Local Extension
- Introduce Named Parameter
- Introduce Null Object
- Introduce Parameter Object
- Isolate Dynamic Receptor
- Lazily Initialized Attribute
- Move Eval from Runtime to Parse Time
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object

http://refactoring.com/catalog/
The Catalogue of Refactorings (2)

- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Recompose Conditional
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Middle Man
- Remove Named Parameter
- Remove Parameter
- Remove Setting Method
- Remove Unused Default Parameter
- Rename Method
- Replace Abstract Superclass with Module
- Replace Array with Object
- Replace Conditional with Polymorphism
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation With Hierarchy
- Replace Delegation with Inheritance
- Replace Dynamic Receptor with Dynamic Method Definition

- Replace Error Code with Exception
- Replace Exception with Test
- Replace Hash with Object
- Replace Inheritance with Delegation
- Replace Loop with Collection Closure Method
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional with GuardClauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Subclass with Fields
- Replace Temp with Chain
- Replace Temp with Query
- Replace Type Code with Class
- Replace Type Code with Module Extension
- Replace Type Code With Polymorphism
- Replace Type Code with State/Strategy
- Replace Type Code with Subclasses
- Self Encapsulate Field
- Separate Query from Modifier
- Split Temporary Variable
- Substitute Algorithm
### Code Smells & Refactoring

- Full document posted on course wiki

---

#### Smells to Refactorings

**Quick Reference Guide**

<table>
<thead>
<tr>
<th>Smell</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative Classes with Different Interfaces:</td>
<td>Rename Interface with Adapter [K.347]</td>
</tr>
<tr>
<td></td>
<td>Rename Method [F.273]</td>
</tr>
<tr>
<td></td>
<td>Move Method [F.142]</td>
</tr>
<tr>
<td>Combinatorial Explosion:</td>
<td>Replace Implicit Language with Interpreter [K.269]</td>
</tr>
<tr>
<td>Comments (i.e., Deodorants):</td>
<td>Rename Method [F.273]</td>
</tr>
<tr>
<td></td>
<td>Extract Method [F.110]</td>
</tr>
<tr>
<td></td>
<td>Introduce Assertion [F.267]</td>
</tr>
<tr>
<td>Conditional Complexity:</td>
<td>Introduce Null Object [F.260, K.321]</td>
</tr>
<tr>
<td></td>
<td>Move Embellishment to Decorator [K.144]</td>
</tr>
<tr>
<td></td>
<td>Replace Conditional with Strategy [K.129]</td>
</tr>
<tr>
<td></td>
<td>Replace Class-Altering with State [K.166]</td>
</tr>
<tr>
<td>Data Class: Classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes.</td>
<td>Move Method [F.142]</td>
</tr>
<tr>
<td></td>
<td>Encapsulate Field [F.205]</td>
</tr>
<tr>
<td></td>
<td>Encapsulate Collection [F.206]</td>
</tr>
<tr>
<td>Data Clumps: Bunches of data that hang around together really ought to be made into their own object. A good test is to consider deleting one of the data values. If you did this, would the others make any sense? If they don't, it's a sure sign that you have an object that's dying to be born.</td>
<td>Extract Class [F.149]</td>
</tr>
<tr>
<td></td>
<td>Preserve Whole Object [F.258]</td>
</tr>
<tr>
<td></td>
<td>Introduce Parameter Object [F.256]</td>
</tr>
<tr>
<td>Divergent Change: Occurs when one class is commonly changed in different ways for different reasons. Separating these divergent responsibilities decreases the chance that one change could affect another and lower maintenance costs.</td>
<td>Extract Class [F.149]</td>
</tr>
<tr>
<td>Duplicated Code: Duplicated code is the most pervasive and pungent smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet essentially the same.</td>
<td>Extract Composite [K.214]</td>
</tr>
<tr>
<td></td>
<td>Extract Method [F.110]</td>
</tr>
<tr>
<td></td>
<td>Extract Class [F.149]</td>
</tr>
<tr>
<td></td>
<td>Introduce Null Object [F.260, K.321]</td>
</tr>
<tr>
<td></td>
<td>Introduce Polymorphic Creation with Factory Method [K.85]</td>
</tr>
<tr>
<td></td>
<td>Pull Up Method [F.322]</td>
</tr>
<tr>
<td></td>
<td>Pull Up Field [F.320]</td>
</tr>
<tr>
<td></td>
<td>Replace Unity/Many Distinctions with Composite [K.224]</td>
</tr>
<tr>
<td></td>
<td>Substitute Algorithm [F.139]</td>
</tr>
<tr>
<td></td>
<td>Unify Interfaces with Adapter [K.247]</td>
</tr>
</tbody>
</table>

(list shown here is not complete)
Typical Refactorings

<table>
<thead>
<tr>
<th>Class Refactorings</th>
<th>Method Refactorings</th>
<th>Attribute Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (sub)class to hierarchy</td>
<td>add method to class</td>
<td>add variable to class</td>
</tr>
<tr>
<td>rename class</td>
<td>rename method</td>
<td>rename variable</td>
</tr>
<tr>
<td>remove class</td>
<td>remove method</td>
<td>remove variable</td>
</tr>
<tr>
<td>extract class</td>
<td>push method down</td>
<td>push variable down</td>
</tr>
<tr>
<td></td>
<td>push method up</td>
<td>pull variable up</td>
</tr>
<tr>
<td></td>
<td>add parameter to method</td>
<td>create accessors</td>
</tr>
<tr>
<td></td>
<td>move method</td>
<td>abstract variable</td>
</tr>
<tr>
<td></td>
<td>extract code in new method</td>
<td></td>
</tr>
<tr>
<td></td>
<td>replace parameter with method</td>
<td></td>
</tr>
</tbody>
</table>
Push Method Down

- Behavior on a superclass is relevant only for some of its subclasses.
- Move it to those subclasses.
• You have methods with identical results on subclasses.
• Move them to the superclass
Add Parameter to Method

- A method needs more information from its caller.
- Add a parameter for an object that can pass on this information.

`Customer` +`getContact()`

`Customer` +`getContact(Date)`
Replace **switch** Statement with Polymorphism

- **switch** statements are very rare in properly designed object-oriented code
  - Therefore, a **switch** statement is a simple and easily detected “bad smell”
  - Of course, not all uses of **switch** are bad
  - A switch statement should *not* be used to distinguish between various types of objects (object types) and *not* rely on data from other objects

- Refactoring options for this case exist
  - The simplest is the creation of subclasses (exploiting polymorphism)
Encapsulate Field

- Un-encapsulated data is a no-no in OO application design.
- Use properly get and set procedures to provide public access to private (encapsulated) member variables.

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

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

    int classSize = course.getStudents().size();
    int classSize = course.students.size();
```
Extract Class

• 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.

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

```java
public class Phone {
    private String areaCode;
    private String number;
}
```
Extract Method

• Sometimes we have methods that do too much. The more code in a single method, the harder it is to understand and get right. It also means that logic embedded in that method cannot be reused elsewhere.

• The Extract Method refactoring is one of the most useful for reducing the amount of duplication in code.

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

```java
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;
    }
}
```
Extract Sub-Class

• When 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.
Extract Super-Class

- When you find two or more classes that share common features, consider abstracting those shared features into a super-class.
- Again, this makes it easier to bind clients to an abstraction, and removes duplicate code from the original classes.

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

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

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

public class Employee extends Person {
    private String jobTitle;
}

public class Student extends Person {
    private Course course;
}
```
Move Method – Before

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

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

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

```java
public class Student {
}

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

- If relying on null for default behavior, use inheritance instead
- Replace the null value with a null object

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

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

```java
if (user == null)
    plan = Plan.basic();
else
    plan = user.getPlan();
```
Replace Error Code with Exception

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

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

```java
void withdraw(int amount) throws BalanceException {
    if (amount > balance)
    {
        throw new BalanceException();
    }
    balance -= amount;
}
```
Replace Exception with Test

• Conversely, if you are catching an exception that could be handled by an if-statement, use that instead.

```java
double getValueForPeriod (int periodNumber)
{
    try
    {
        return values[periodNumber];
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        return 0;
    }
}
```

```java
double getValueForPeriod (int periodNumber)
{
    if (periodNumber >= values.length) return 0;
    return values[periodNumber];
}
```
Nested Conditional with Guard

- A method has conditional behavior that does not make clear what the normal path of execution is. Use Guard Clauses for all the special cases.

```java
double getPayAmount() {
    double result;
    if (isDead) result = deadAmount();
    else {
        if (isSeparated) result = separatedAmount();
        else {
            if (isRetired) result = retiredAmount();
            else result = normalPayAmount();
        }
    }
    return result;
}
```

```java
double getPayAmount() {
    if (isDead) return deadAmount();
    if (isSeparated) return separatedAmount();
    if (isRetired) return retiredAmount();
    return normalPayAmount();
};
```
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.

```java
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;
}
```
Replace Temp with Query

- You are using a temporary variable to hold the result of an expression. Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods and allows for other refactorings.

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

```java
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;

... double basePrice() {
    return quantity * itemPrice;
}
Rename Variable or Method

- 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.

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

```java
public class Customer {
    public double getInvoiceCreditLimit();
}
```
More on Refactoring

• Refactoring Catalog
  • http://www.refactoring.com/catalog

• Java Refactoring Tools
  • NetBeans 4+ – Built In
  • JFactor – works with VisualAge and JBuilder
  • RefactorIt – plug-in tool for NetBeans, Forte, JBuilder and JDeveloper. Also works standalone.
  • JRefactory – for jEdit, NetBeans, JBuilder or standalone

• Eclipse & Visual Studio 2005+ & IntelliJ IDEA
  • Refactoring Built In
    • Encapsulate Field, Extract Method, Extract Interface, Reorder Parameters, Remove Parameter, Promote Local Var to Parameter, more.
Further Reading

- Martin Fowler: Refactoring, 2005

- http://refactoring.com/catalog/

New Edition in Dec. 2018
Hands-On Example

by

Martin Fowler

(separate PDF on course wiki)
Example: Program Spec

• Very simple program to calculate and print a statement of a customer's charges at a video store.

• The program is told which movies a customer rented and for how long.

• It then calculates the charges, which depend on how long the movie is rented and the type of the movie.

• There are three kinds of movies: regular, children's, and new releases.

• In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.
Intro Example (from M. Fowler’s example)

```java
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }

    public String getTitle() {
        return _title;
    }
}
```
Example, continued

public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }

    public String getTitle() {
        return _title;
    }
}

double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
    case Movie.REGULAR:
        result += 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        break;
    case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
    case Movie.CHILDREN:
        result += 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
}

Charge depends on number of days rented and type of movie.
Example,

It is a bad idea to do a switch based on an attribute of another object. If you must use a switch statement, it should be on your own data, not on someone else’s. This implies that the charge should move onto movie.

Here: switch uses 'days rented' → belongs to class 'Rental'.

Charge depends on number of days rented and type of movie.
Example, continued

Idea to get rid of switch: Subclassing and polymorphism!
Example, continued

Idea to get rid of switch: Subclassing and polymorphism!

Doesn’t work!

Why?

Because a movie can change its type (classification) over time (e.g., from ’New Release’ to ’Regular’) – an object cannot change its class.

Solution:
State (or Strategy) Pattern
Example, continued

Applying the State pattern:

To introduce the State pattern we will use three refactorings:
1. Move the type code behavior into the State pattern with Replace Type Code with State.
2. Use Move Method to move the switch statement into the class Price.
3. Use Replace Conditional with Polymorphism to eliminate the switch statement.
Example, continued

```java
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }

    public String getTitle() {
        return _title;
    }
}
```

Refactoring: **Self Encapsulate Field** to ensure that all uses of the type code go through getting and setting methods.
Example, continued

Refactoring: **Self Encapsulate Field** to ensure that all uses of the type code go through getting and setting methods.

```java
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        setPriceCode(priceCode);
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }

    public String getTitle() {
        return _title;
    }
}
```

Add the new Price class and sub-classes and provide the type code behavior in the Price object – using an abstract method on Price and concrete methods in the sub-classes.

```java
abstract class Price {
    abstract int getPriceCode()
}
class ChildrensPrice extends Price {
    int getPriceCode() {
        return Movie.CHILDREN;
    }
}
class NewReleasePrice extends Price {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    }
}
class RegularPrice extends Price {
    int getPriceCode() {
        return Movie.REGULAR;
    }
}
```
Example, continued

```java
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private Price _price;
    public Movie(String title, int priceCode) {
        _title = title;
        setPriceCode(priceCode);
    }

    public int getPriceCode() {
        return _price.getPriceCode();
    }

    public void setPriceCode(int arg) {
        switch (arg) {
            case Movie.REGULAR:
                _price = new RegularPrice();
                break;
            case Movie.CHILDREN:
                _price = new ChildrensPrice();
                break;
            case Movie.NEW_RELEASE:
                _price = new NewReleasePrice();
                break;
            default:
                throw new IllegalArgumentException("Incorrect Price Code");
        }
    }
}
```

Next change Movie’s accessors for the price code to use the new class.
Example, continued

```java
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private Price _price;
    public Movie(String title, int priceCode) {
        _title = title;
        setPriceCode(priceCode);
    }
    public int getPriceCode() {
        return _price.getPriceCode();
    }
    public void setPriceCode(int arg) {
        switch (arg) {
        case Movie.REGULAR:
            price = new RegularPrice();
            break;
        case Movie.CHILDREN:
            price = new ChildrensPrice();
            break;
        case Movie.NEW_RELEASE:
            price = new NewReleasePrice();
            break;
        default:
            throw new IllegalArgumentException("Incorrect Price Code");
        }
    }
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDREN:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
            default:
        }
        return result;
    }
}
```

Refactoring: **Move Method** is applied to method `getCharge()`
Example, continued

Refactoring: **Move Method** is applied ...

```java
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private Price _price;

    public Movie(String title, int priceCode) {
        _title = title;
        _price = priceCode;
    }

    public int getPriceCode() {
        return _price.getPriceCode();
    }

    public void setPriceCode(int arg) {
        switch (arg) {
            case Movie.REGULAR:
                _price = new RegularPrice();
                break;
            case Movie.CHILDREN:
                _price = new ChildrensPrice();
                break;
            case Movie.NEW_RELEASE:
                _price = new NewReleasePrice();
                break;
            default:
                throw new IllegalArgumentException("Incorrect Price Co");
        }
    }

    double getCharge(int daysRented) {
        return _price.getCharge(daysRented);
    }
}
```

```java
Class Movie ...

double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
}

Class Price ...

double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
                result += (daysRented - 2) * 1.5;
                break;
            }
        case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
        case Movie.CHILDREN:
            result += 1.5;
            if (daysRented > 3) {
                result += (daysRented - 3) * 1.5;
                break;
            }
    }
    return result;
}
```
**Example, continued**

Refactoring: **Replace Conditional with Polymorphism** is applied ...

Now we have removed all the methods that needed a price code. So we can get rid of the price code methods and data on both Movie and Price.
Example, continued

The gain is now that should we change any of price’s behavior, add new prices, or add extra price dependent behavior; it will be much easier to change. The rest of the application does not know about the use of the state pattern.

It is going to be much easier to change the classification structure of movies, and to alter the rules for charging.
How is this an improvement?

• Adding a new movie type, such as 3D-Movie, does not require revising and recompiling existing code

• Children, Regular, and NewRelease movies are likely to differ in other ways, and we’ve already separated them out (so we won’t need more switch statements)

• In the charge calculation, we’ve gotten rid of the flags we needed to tell one kind of movie from another

• We’re now using objects the way they were meant to be used
Next Lecture

• Date/Time:
  • Friday, 23-Nov, 10:15-12:00

• Topic:
  • Agile/Lean Methods

• Labs:
  • Complete and submit Homework 5
  • Go to next week’s labs: introduction to Homework 6