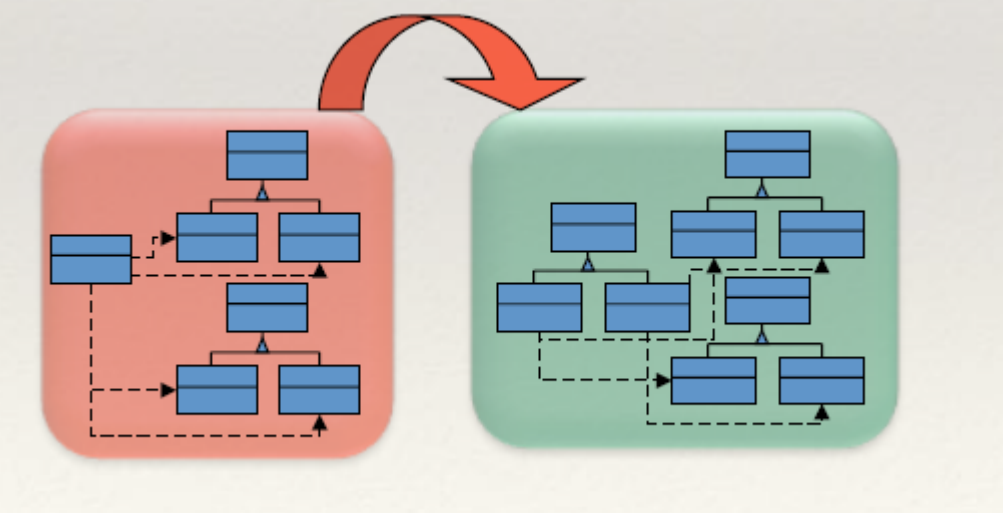
COMP2511
Refactoring

Refactoring

The process of restructuring (changing the internal structure of software) software to make it *easier to understand* and *cheaper to modify* without changing its *external, observable behaviour*



Why should you refactor?

- Refactoring improves design of software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster
- Refactoring helps you to conform to design principles and avoid design smells

When should you refactor?

Tip: *When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature*

Refactor when:

- You add a function (swap hats between adding a function and refactoring)
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Common Bad Code Smells

- **Duplicated Code**
  - Same code structure in more than one place or
  - Same expression in two sibling classes
- **Long Method**
- **Large Class** (when a class is trying to do too much, it often shows up as too many instance variables)
- **Long Parameter List**
- **Divergent Change** (when one class is commonly changed in different ways for different reasons)
- **Shotgun Surgery** (The opposite of divergent change, when you have to make a lot of little changes to a lot of different classes

The Video Rental Example

What is wrong with the design?

Is it wrong to write a quick and dirty solution OR is it an aesthetic judgment (dislike of ugly code) …

- Overly long `statement()` method , poorly designed that does far too much, tasks that should be done by other classes (Code Smell: Long Method)
- What if customer wanted to generate a statement in HTML? - Impossible to reuse any of the behaviour of the current statement method for an HTML statement. (Code Smell: Duplicated code)
- What about changes?
  - What happens when "charging rules" change?
  - what if the user wanted to change the way the movie was classified
- The code is a maintenance night-mare (Design smell: Rigidity)

Improving the design

Apply a series of fundamental refactoring techniques:

Technique #1: Extract Method

- Find a logical clump of code and use Extract Method.

  Which is the obvious place? the `switch` statement

- Scan the fragment for any variables that are local in scope to the method we are looking at

  (`Rental r` and `thisAmount`)

- Identify the changing and non-changing local variables
- Non-changing variable can be passed as a parameter
- Any variable that is modified needs more care, if there is only one, you could simply do a return

Improving the design

Technique #2: Rename variable

- Is renaming worth the effort? Absolutely
- Good code should communicate what it is doing clearly, and variable names are a key to clear code. Never be afraid to change the names of things to improve clarity.

Tip

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Improving the design

#3: Move method

- Re-examine method `calculateRental()` in class `Customer`
- Method uses the `Rental` object and not the `Customer` object
- Method is on the wrong object

Tip

Generally, a method should be on the object whose data it uses

Improving the design

What OO principles do Extract Method and Move Method use?

They make code reusable through Encapsulation and Delegation

But, isn't encapsulation about keeping your data private?

The basic idea about encapsulation is to protect information in one part of your application from other parts of the application, so

- You can protect data
- You can protect behaviour – when you break the behaviour out from a class, you can change the behaviour without the class having to change

And what is delegation?

- The act of one object forwarding an operation to another object to be performed on behalf of the first object

Improving the design

#4: Replace Temp With Query

- A technique to remove unnecessary local and temporary variables
- Temporary variables are particularly insidious in long methods and you can loose track of what they are needed for
- Sometimes, there is a performance price to pay

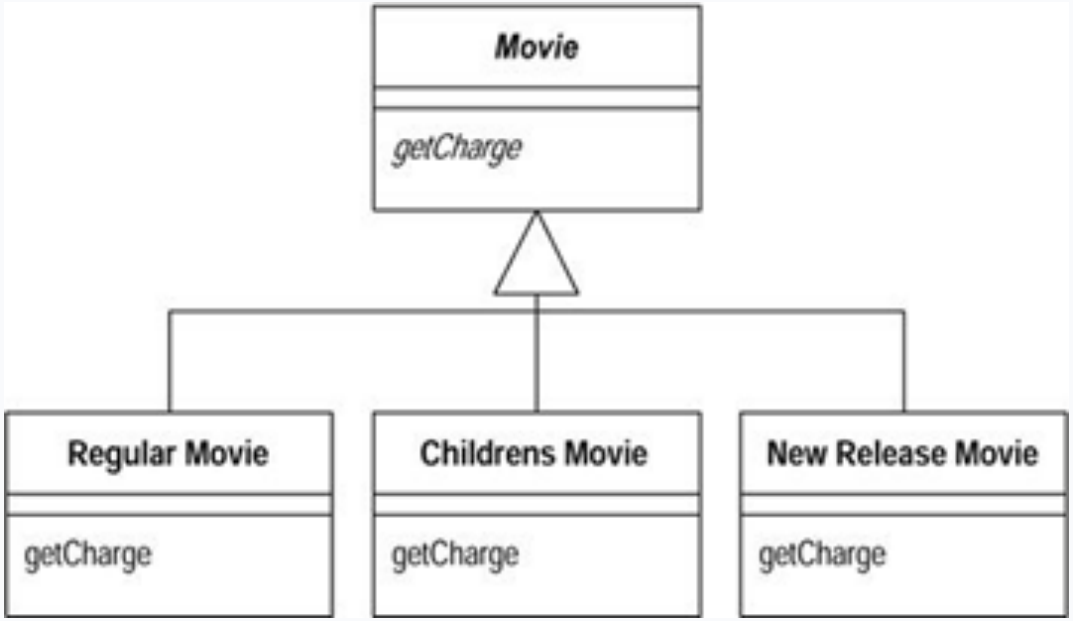
Improving the design

#5: Replacing conditional logic with Polymorphism

- The switch statement – an obvious problem, with two issues
- class `Rental` Is tightly coupled with class `Movie` - a switch statement based on the data of another object – not a good design
- There are several types of movies with its own type of charge, hmm… sounds like inheritance

Improving the design

- A `base` class `Movie` class with method `getPrice()` and sub-classes `NewRelease`, `ChildrenMovie` and `Regular`
- This allows us to replace `switch` statement with polymorphism



- Sadly, it has one flaw...a movie can change its classification during its life-time

So, what options are there besides inheritance ?

- Composition – reuse behaviour using one or more classes with composition
- Delegation: delegate the functionality to another class

…this is the second time, this week we have said, we need something more than inheritance

So, next ...

- **Design Principle:** Favour composition over inheritance
- **More refactoring techniques to solve our "switch" problem**
  - Replace type code with Strategy/State Pattern
  - Move Method
  - Replace conditional code with polymorphism