Contents

[1. OVERVIEW 1](#_Toc424725556)

[2. Types of Refactoring 2](#_Toc424725557)

[3. Refactoring Process 2](#_Toc424725558)

[4. Example 2](#_Toc424725559)

[4.1 Extract Method 2](#_Toc424725560)

[4.2 Replace Magic Number 2](#_Toc424725561)

[4.3 Encapsulate Field 2](#_Toc424725562)

[4.4 Pull Up Method 3](#_Toc424725563)

[4.5 Pull Down Method 3](#_Toc424725564)

[4.6 Inline Class 3](#_Toc424725565)

[5. List of Refactorings 4](#_Toc424725566)

[5.1 *Techniques that allow for more abstraction* 4](#_Toc424725567)

[5.2 *Techniques for breaking code apart into more logical pieces* 4](#_Toc424725568)

[5.3 *Techniques for integrating code that’s needlessly spread apart* 4](#_Toc424725569)

[5.4 *Techniques for improving names and location of code* 4](#_Toc424725570)

[6. Antipatterns (a way of coding that makes errors more likely) 5](#_Toc424725571)

[7. Fixing an Antipattern -- Exception as Control Flow 6](#_Toc424725572)

Code Refactor

# OVERVIEW

* **Local**: Refactoring should not affect unrelated parts of the program
* **Semantics-preserving**: The behaviour of the refactored code should be identical to the behaviour of the original code
* **Why Refactor**?
  + Change the design
  + Maintain good structure & prevent decay
  + Make code easier to change
  + Prevent future errors
  + Make it easy to understand

# Types of Refactoring

* Composing methods
* Moving features between objects
* Organizing data
* Simplifying conditionals
* Making method calls simpler
* Generalizations

# Refactoring Process

* Create unit tests (if needed)
* Run unit tests
* Make changes (refactor!)
* Re-run unit tests
* Evaluate results

# Example

# Extract Method

* + Useful when the code does the same thing many times
  + Methods should do one conceptual thing.

# Replace Magic Number

* + Using symbolic constants is often better coding practice than putting numbers directly into the code.
  + Easy to understand and update

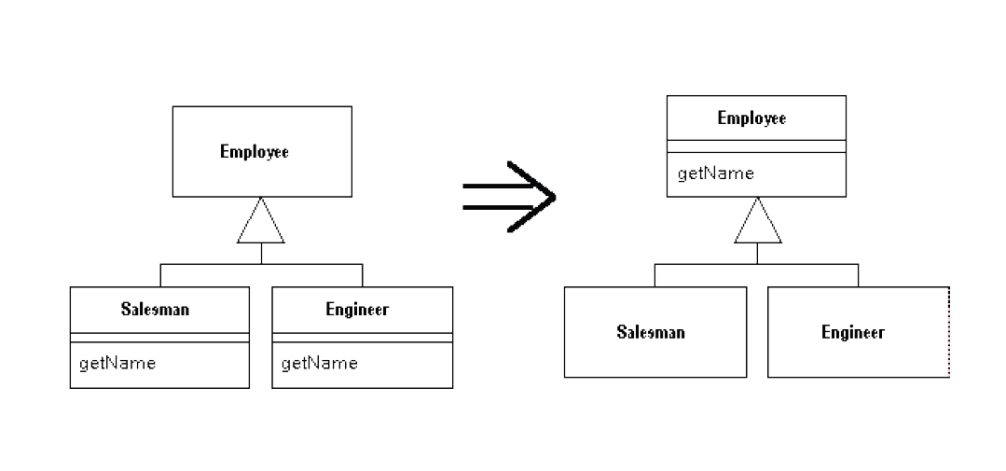
**Rename Variable and Methods** (for more clarity)

* + For more clarity

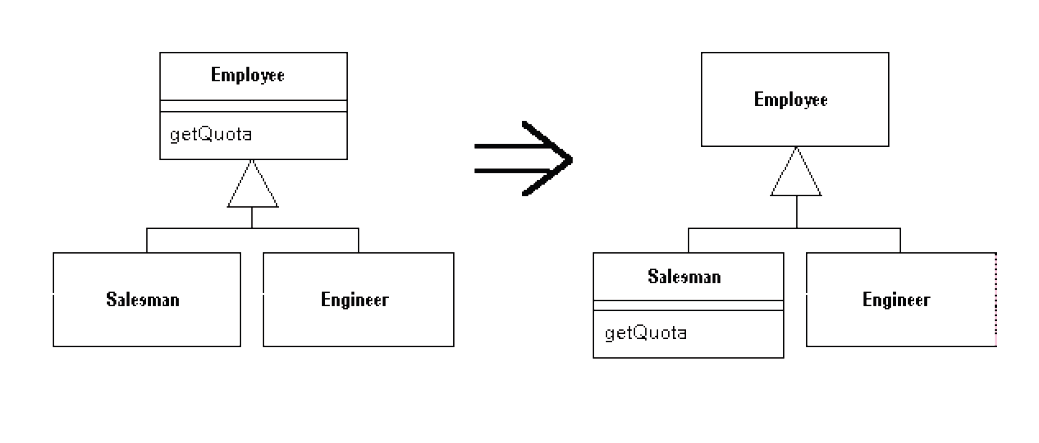
# Encapsulate Field

* + OOP programming practice: **private fields, public access methods**
  + Replace x.foo with x.getFoo() and x.setFoo()
  + Why? : Modularity, logging, flexibility…

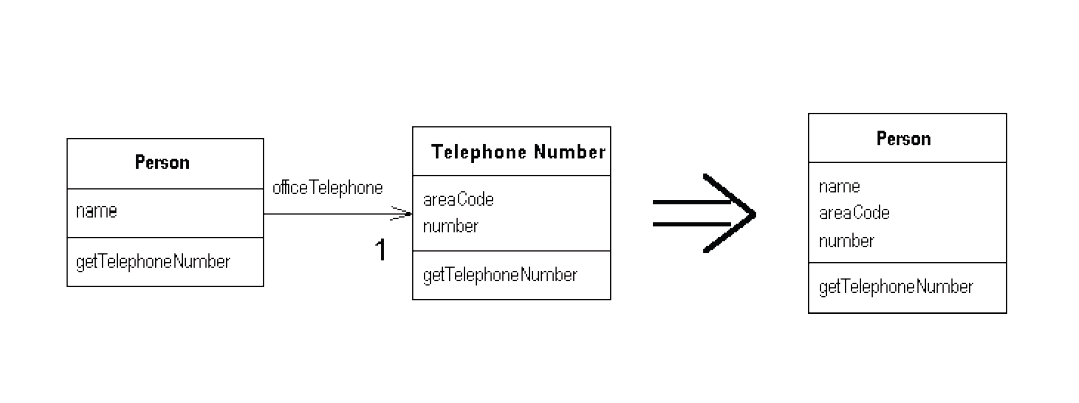
# Pull Up Method

****

# Pull Down Method

****

# Inline Class



# List of Refactorings

# *Techniques that allow for more abstraction*

* + 1. **Encapsulate Field** (force code to access the field with methods that encompass a meaning operation)
    2. Generalize Type (create more general types to allow for more code sharing)
    3. Replace conditional with polymorphism (use inheritance and virtual dispatch instead of a conditional)

# *Techniques for breaking code apart into more logical pieces*

* + 1. **Extract Method** (pull out part of a larger method into a new method)
    2. **Extract Class** (moves code from an existing class into a new class)

# *Techniques for integrating code that’s needlessly spread apart*

* + 1. Inline Method (integrate a copy of the body of a method into its calling method)
    2. Inline Class (put all of the fields and methods of a class into another class and erase the original)

# *Techniques for improving names and location of code*

* + 1. Move Method/Field (move to a more appropriate class or source file)
    2. **Rename Method/Field** (changing the name into a new one that better reveals its purpose)
    3. **Pull Up** (in OOP. Move to a superclass)
    4. **Push Down** (in OOP. Move to a subclass)

# Antipatterns (a way of coding that makes errors more likely)

* 1. **The Blob** 
     1. Idea: One object does basically everything, too much code/logic centralized in this one class
     2. *Solution: extract methods and classes so it is spread out.*
  2. **Lava Flow**
     1. Idea: Old dead (useless) code hanging around in the software, Time wasted maintaining.
     2. *Solution: delete it (restore from version control if needed)*
  3. **Functional Decomposition**
     1. Idea: Code resembles a structural language when using OOP, often caused by non-OOP programmers writing in Java/C#.
     2. *Solution: Extract classes and methods, pull up common code*
  4. **Copy-and-Paste**
     1. Idea: you copy and paste some code
     2. But: copies are different so bugs will not be solved if one is updated and the rest are not.
     3. *Solution: Extract class or method to replace all these copies.*
  5. Poltergeists
     1. Idea: classes with limited roles and effective life cycles. Objects hat pop in, do one thing. Then vanish.
     2. But: waste of resources and inefficient
     3. *Solution: inline their functionality to other classes*
  6. Golden Hammer
     1. Idea: applied familiar concept or architecture to everything
     2. But: it does not make sense sometime.
     3. *Solution: Refactor the code to more appropriate design.*
  7. **Exceptions as control flow**
     1. Idea: Use exceptions as control flow
     2. But: Exceptions should not be considered normal, exceptions are expensive to generate and handle
     3. *Solution: Refactor to avoid Exceptions where they are expected.*
  8. Spaghetti Code
     1. Idea: program without structure
     2. But: No structure means difficult to extend or change things
     3. Solution: refactor until the code has appropriate structure.

# Fixing an Antipattern -- Exception as Control Flow

AntiPattern:

try {

fileReader.readFile (fileSelector.getSelection( ).getFileName( ) );

}

catch ( NullPointerException npe ) {

showErrorDialog ( Error.NO\_FILE\_SELECTED ) ;

return;

}

Rewrite to avoid the Exception in the first place:

If ( fileSelector.getSelection( ) == null ) {

showErrorDialog ( Error.NO\_FILE\_SELECTED ) ;

return;

}

fileReader.readFile ( fileSelector.getSelection ( ).getFileName ( ) ) ;