# Lecture 25 — Refactoring

Jeff Zarnett & Patrick Lam
jzarnett@uwaterloo.ca &p.lam@ece.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 14, 2015

# Refactoring

We have mentioned refactoring very frequently in the course.

Refactoring incrementally improves code quality using local, semantics-preserving changes to the code.

More restrictive definition: refactoring must make code clearer.

Ease of understanding is not our only goal, so we will use the more inclusive definition.

Constrast this against a performance optimization: makes code harder to understand, but faster.

Two small clarifications:

- **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
  (Nobody gets it 100% right on the first try).
- 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

1. Create unit tests (if needed)
2. Run unit tests
3. Make changes
4. Re-run unit tests
5. Evaluate results

Hopefully, you already have unit tests so you can skip step 1. If not - why not?

# Refactoring Example: Extract Method

```
// (1) make sure the code only runs on mac os x
boolean mrjVersionExists =
     System.getProperty(``mrj.version'') != null;
boolean osNameExists =
     System.getProperty(``os.name'')
           .startsWith(``Mac OS'');

if ( !mrjVersionExists || !osNameExists) {
  System.err.println(
       ``Not running on a Mac OS X system.'');
  System.exit(1);
}
```

# Refactoring Example: Extract Method

```
// (2, 3) do all the preferences stuff
// and get the default color
preferences = Preferences
          .userNodeForPackage(this.getClass());
int r = preferences.getInt(CURTAIN_R, 0);
int g = preferences.getInt(CURTAIN_G, 0);
int b = preferences.getInt(CURTAIN_B, 0);
int a = preferences.getInt(CURTAIN_A, 255);
currentColor = new Color(r,g,b,a);
```

# Refactoring Example: Extract Method

We can instead split this into three sub-methods.

```
dieIfNotRunningOnMacOsX();
connectToPreferences();
getDefaultColor();
```

# Refactoring: Extract Method

This is useful when the code does the same thing many times.

Replace all the clones with a call to a single method.

Update the code once to update all.

# Refactoring: Extract Method

Methods should do one conceptual thing.

A good test: can you come up with a name for it?

We are putting more design into the code.

Eclipse supports doing this in one click.

# Refactoring: Replace Magic Number

Using symbolic constants is often better coding practice than putting numbers directly into the code.

```
double potentialEnergy(double mass, double height) {
    return mass * height * 9.81;
}
```

# Refactoring: Replace Magic Number

Using symbolic constants is often better coding practice than putting numbers directly into the code.

```
static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
```

Easier to understand and easier to update.

# Refactoring: Rename Variable

One minor example of refactoring is renaming a variable.

```
DateTime creationTime;

public DateTime getCreationTime(){
    return creationTime;
}

public void setCreationTime(DateTime newTime) {
    this.creationTime = newCreationTime;
}
```

We change the variable and methods for more clarity.

# Refactoring: Rename Variable

One minor example of refactoring is renaming a variable.

```
DateTime creationDateTime;

public DateTime getCreationDateTime(){
    return creationDateTime;
}

public void setCreationDateTime(DateTime newDateTime)
    this.creationDateTime = newCreationDateTime;
}
```

We change the variable and methods for more clarity.

# Refactoring: Limitations

Diminishing returns as more is changed.

Changing interfaces might not be possible.

External systems (e.g., database) might constrain changes.

# Refactoring vs Performance

Refactoring aims to improve *non-functional properties* of the code; performance may get better, worse, or stay the same.

Our goal is easier to maintain code.

Example: split one loop into two separate loops.

# When to Refactor

- Continuously?
- Fixed Schedule?
- When Fixing a Bug?
- At the End of a Project?
- At the Start of a Project?

# List of Refactorings

Techniques that allow for more abstraction:

- *Encapsulate Field*: force code to access the field with getter and setter methods.
- *Generalize Type*: create more general types to allow for more code sharing.
- *Replace conditional with polymorphism*: use inheritance and virtual dispatch instead of a conditional.

# List of Refactorings

Techniques for breaking code apart into more logical pieces:

- *Extract Method*: as seen above, pull out part of a larger method into a new method.
- *Extract Class*: moves code from an existing class into a new class.

# List of Refactorings

Techniques for integrating code that's needlessly spread apart:

- *Inline Method*: integrate a copy of the body of a method into its calling method.
- *Inline Class*: put all of the fields and methods of a class into another class and erase the original.

# List of Refactorings

Techniques for improving names and location of code:

- *Move Method/Field*: move to a more appropriate class or source file.
- *Rename Method/Field*: changing the name into a new one that better reveals its purpose.
- *Pull Up*: in OOP, move to a superclass.
- *Push Down*: in OOP, move to a subclass.

# Antipatterns

Antipatterns are like antimatter – the opposite of how software should be designed.

In software, a way of coding that makes errors more likely.

Not only for software (e.g., management antipatterns).

# The Antipatterns Family

Some common antipatterns you might encounter:

- The Blob
- Lava Flow
- Functional Decomposition
- Copy-and-Paste Programming
- Poltergeists
- Golden Hammer
- Exceptions as Control Flow
- Spaghetti Code

# Antipatterns: The Blob

One object does basically everything.

From the horror movie: the blob just keeps growing.

Too much code/logic centralized in this one class.

Solution: extract methods and classes so it is spread out.

# Antipatterns: Lava Flow

Old dead (useless) code hanging around in the software.

Nobody changes/deletes it for fear of breaking something.

From the geological phenomenon: lava flows then solidifies.

Time wasted testing and refactoring this dead code.

Solution: delete it (restore from version control if needed).

# Antipatterns: Functional Decomposition

Code resembles a structural language when using OOP.

Often caused by non-OOP programmers writing in Java/C#.

Solution: Extract classes and methods, pull up common code.

# Antipatterns: Copy-and-Paste Programming

Code is copy-and-pasted; possibly modified slightly each time.

Copies can be slightly different so bugs will not be solved if one is updated and the rest are not.

Solution: extract class or method to replace all these copies.

# Antipatterns: Poltergeists

Poltergeist (English): a ghost that allegedly causes noise or destruction and then vanishes.

Poltergeists: classes with limited roles and effective life cycles.

Objects that pop in, do one thing, then vanish.

Waste of resources and inefficient.

Solution: Call the Ghostbusters!

> Actually, inline their functionality to other classes.

# Antipatterns: Golden Hammer

“When all you have is hammer, everything looks like a nail.”

Familiar concept or architecture applied to everything, whether that makes sense or not.

Solution: Refactor the code to more appropriate design.

# Antipatterns: Exceptions as Control Flow

"That error is supposed to happen."

Exceptions should not be considered normal.

Exceptions are expensive to generate and handle.

Solution: Refactor to avoid Exceptions where they are expected.

# Antipatterns: Spaghetti Code

Program with little or no structure.

Typically small number of objects with long methods.

No structure means difficult to extend or change things.

Solution: refactor until the code has appropriate structure.

We'll continue by talking about the refactorings in more detail; perhaps this will help you understand the goals of refactoring, so that you can do it in your own code.

# Refactoring: Encapsulate Field

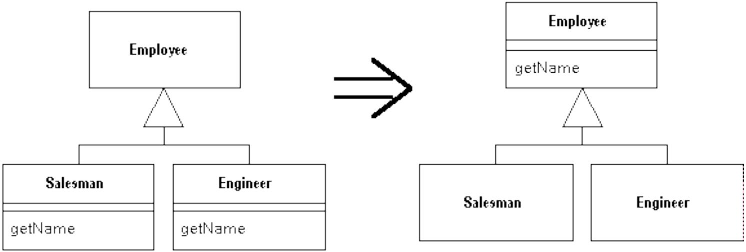
Java programming practice: private fields, public accessor methods.

Replace `x.foo` with `x.getFoo()` and `x.setFoo(y)`.

Why? Modularity, logging, flexibility, etc.

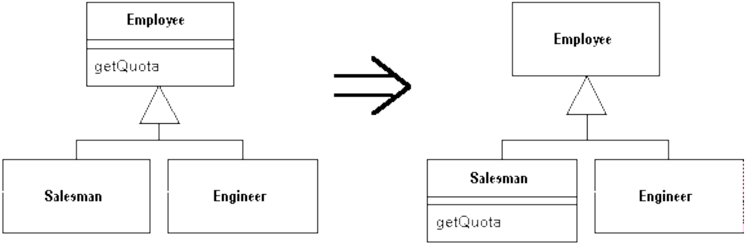
Should all fields be encapsulated?

This refactoring is an example of a generalization.



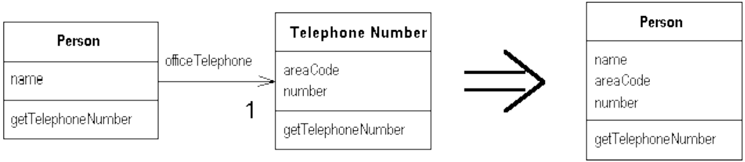
# Refactoring: Push Down Method

Sometimes we want to do the opposite.



# Refactoring: Inline Class

Here, we get rid of helper classes that aren't useful.



# Fixing an Antipattern: Exception as Control Flow

In this example, we are (ab)using an Exception and handler:

```
[language={Java}]
try {
   fileReader.readFile(
      fileSelector.getSelection().getFileName());
} catch (NullPointerException npe) {
   showErrorDialog(Error.NO_FILE_SELECTED);
   return;
}
```

# Fixing an Antipattern: Exception as Control Flow

Rewrite to avoid the Exception in the first place:

```
[language={Java}]
if (fileSelector.getSelection() == null) {
   showErrorDialog(Error.NO_FILE_SELECTED);
   return;
}
fileReader.readFile(
      fileSelector.getSelection().getFileName());
```

# Refactoring: Further Examples

Now we'll do some further examples on the board and/or Eclipse, if time allows.

- Generalize Type
- Replace Conditional with Polymorphism
- Extract Method / Variable
- ... and maybe more!