Informatics 1: Object Oriented Programming

Tutorial 06

Week 7: Refactoring

Volker Seeker (volker.seeker@ed.ac.uk)
Vidminas Vizgirda (s1750767@ed.ac.uk)

1 Introduction

All this time we have been writing code to deliver new functionality or fix what already exists but doesn't work quite as intended. There's more and more code, and with time it just gets messier. Perhaps you have even had the experience of writing a program and not wanting to ever touch the code again, as long as it works.

Well, today's tutorial is different - we will be re-writing and deleting code to make it pretty. Who says programming can't be a beautiful art? Refactoring sure can be.

1.1 Wait.. but what is refactoring actually?

Refactoring is transforming a mess into clean and simple code without changing behaviour (no new features, nor new bugs should be introduced; all previously created tests should still pass; existing bugs may be exposed by refactoring, but fixing them would be debugging)

1.2 And why would you refactor?

I mean, if the code works, why should you bother re-writing it (especially since this might accidentally introduce new bugs)..?

Imagine the feeling of opening a program that someone else wrote and smiling to yourself. The code is crisp, clear and easy to read. It's obvious what the classes are for and what every function does. There is minimal code duplication and everything is done in the simplest way possible. Adding new functionality would be a piece of cake and everything that the program does is thoroughly covered by unit tests. Turning code into such an experience is the ultimate goal of refactoring.

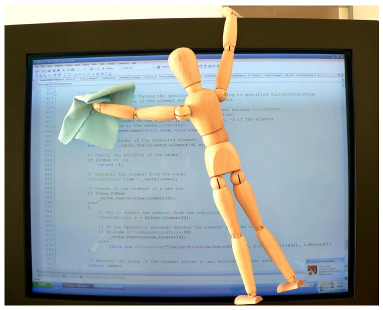
Figure 1: "Code washing" by Ezu is licensed under CC BY-NC-ND 2.0.

Code naturally accrues "technical debt". This metaphor is like if you get a loan from a bank in real life. It allows you to make purchases faster. But you pay extra for expediting the process - you don't just pay off the amount you borrowed, but also the additional interest on the loan. Needless to say, you can even rack up so much interest that the amount of interest exceeds your total income, making full repayment impossible.

The same thing can happen with code. You can temporarily speed up without refactoring code, but over time, adding new features or fixing bugs gets more and more difficult. At some point it becomes easier to just start over.

It might happen because of trying to push features quickly to meet deadlines, taking some shortcuts or ”dirty hacks” on the way. Or perhaps multiple people working on the project didn’t communicate well and different source files now all follow different conventions. Or even if there is a lack of tests - you can write code faster by ditching tests for new features, but this will gradually slow your progress every day (as various small bugs come back to haunt new code), until you eventually pay off the debt by writing tests.

1.3 When should you refactor?

If you try to write perfect code the first time in everything you do, it will be hard to get anywhere at all. Sometimes the requirements change, or you realise new things as the development progresses, which requires rethinking your approach. You might find yourself rewriting code over and over again, and at the end of the day, barely any progress is achieved.

One approach is called the rule of three:

1. The first time you have to do something, just get it done in the simplest way that you can think of
2. The second time you do the same thing, you may cringe or wince a little, but do it again
3. The third time it comes up, it’s time to refactor your code

Refactoring is also good to do before adding new features or when trying to find a well hidden bug, or whenever a module of your program is finished during review.

2 Let’s dig in(to some terrible code)

Task 1 - Something not too bad for starters ◁ **Task**

Discuss with your peers - what’s wrong with the piece of code below and how you might improve it? What the code actually does is not important, you can still make some small changes that simplify the code without changing the behaviour. Don’t worry about the missing variable and method declarations - assume they are defined elsewhere in the same file as the snippet.

```java
public class PizzaDelivery {

  // .. some code omitted ...

  public int getRating() {
    return moreThanFiveLateDeliveries() ? 1 : 2;
  }

  private boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }

  public boolean hasDiscount(Order order) {
    double basePrice = order.basePrice();
    return basePrice > 1000;
  }

  public void renderBanner() {
    if ((platform.toUpperCase().contains("MAC")) &&
         (browser.toUpperCase().contains("IE")) &&
          wasInitialized() && resize > 0 )
    {
      draw(getRating());
    }

    // ... some code omitted ...
  }

  // ... some code omitted ...
}
```

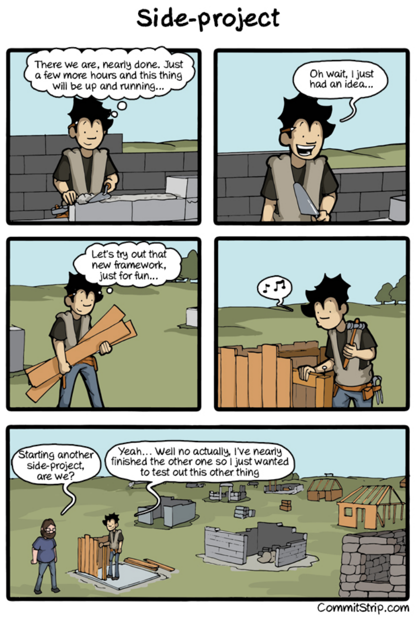
Side-project
There we are, nearly done. Just a few more hours and this thing will be up and running...
Oh wait, I just had an idea...
Let's try out that new framework, just for fun...
♪♫
Starting another side-project, are we?
Yeah... Well no actually, I've nearly finished the other one so I just wanted to test out this other thing
CommitStrip.com

Figure 2: "West Side-project story" by Commitstrip is available for non-commercial reuse under the Commitstrip licence.

Even though the functionality is nicely split across methods with meaningful names, this piece of code can be refactored in multiple ways. You can:

* Inline method to simplify `getRating`: (no need to have a separate method for something very simple, especially if the number of deliveries is hard coded as part of the name!)

```java
public int getRating() {
  return numberOfLateDeliveries > 5 ? 1 : 2;
}
```

* Inline temporary variable to simplify `hasDiscount`: (no need for a temporary variable which is only used once)

```java
public boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

* Extract variable to simplify `renderBanner`: (the if condition is much more readable this way)

```java
public void renderBanner() {
  final boolean IS_MAC_OS = platform.toUpperCase().contains("MAC");
  final boolean IS_IE = browser.toUpperCase().contains("IE");
  final boolean WAS_RESIZED = resize > 0;

  if (IS_MAC_OS && IS_IE && wasInitialized() && WAS_RESIZED) {
    draw(getRating());
  }
  // ... some code omitted ...
}
```

* Get rid of “magic numbers” by using easily modifiable constant values. You could do the same for the "MAC" and "IE" strings.

```java
private static final LOW_RATING = 1;
private static final HIGH_RATING = 2;
private static final DISCOUNT_START = 1000;
private static final LATE_DELIVERY_MAX = 5;

public int getRating() {
  return numberOfLateDeliveries > LATE_DELIVERY_MAX ? LOW_RATING : HIGH_RATING;
}

public boolean hasDiscount(Order order) {
  return order.basePrice() > DISCOUNT_START;
}
```

Task 2 - Another one to bite the dust

◁ Task

To the right is another piece of code. This one is also incomplete, but even just what we have here is more complicated than it needs to be. Can you identify how?

```java
public double getPayAmount() {
  double result;

  if (isDead) {
    result = deadAmount();
  }
  else {
    if (isSeparated) {
      result = disamt();
      result += separatedAmount();
    }
    else {
      if (isRetired) {
        result = disamt();
        result += retiredAmount();
      }
      else {
        result = disamt();
        result += normalPayAmount();
      }
    }
  }
  return result;
}

/**
 * This method computes the disability pay
     amount for a person.
 * @return disability pay amount for this
     instance
 */
private double disamt() {
  if (seniority < 2) {
    return 0;
  }
  if (monthsDisabled > 12) {
    return 0;
  }
  if (isPartTime) {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

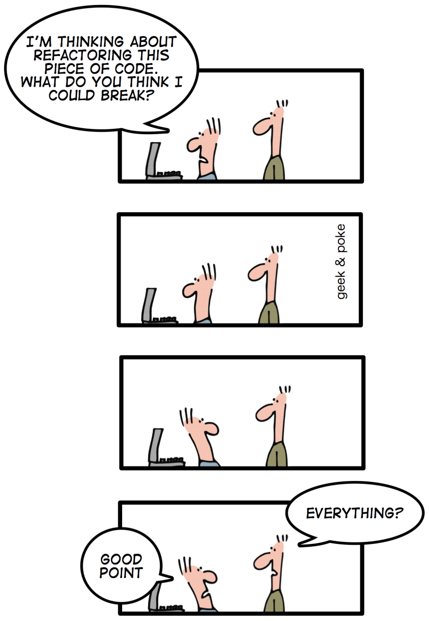
I'M THINKING ABOUT REFACTORING THIS PIECE OF CODE. WHAT DO YOU THINK I COULD BREAK?
geek & poke
EVERYTHING?
GOOD POINT

Figure 3: Take care when refactoring (unit tests are great at catching bugs that may pop up). “What could go wrong?” by Oliver Widder is licensed under CC BY 3.0.

This is a very similar exercise, just with different refactoring methods. Again, you could give the discussion a few minutes and then share the answers. Some ideas could be to:

* Replace "nested conditional from hell" with guard clauses in getPayAmount:

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

* Consolidate conditionals, extract method and rename method to simplify `disabilityAmount`:

```java
private double disabilityAmount() {
  if (isNotEligibleForDisability()) {
    return 0;
  }
  // ...
}

private boolean isNotEligibleForDisability() {
  return (seniority < 2 || monthsDisabled > 12 || isPartTime);
}
```

Task 3 - IntelliJ can help!

◁ **Task**

As you keep refactoring code, you may notice that similar patterns keep coming up again and again. That's why specific refactorings have names like "extract method" or "consolidate conditionals". Since these patterns are common, IDEs like IntelliJ can actually help by providing smart suggestions.

Start by opening IntelliJ's "Learn IDE features" tutorial. If you have a project open, you can find it in the status bar under `Help > Learn IDE Features`. You can also find it in the welcome menu as shown in Figure 4.

Follow the tutorial prompts to work through the exercises under 'Essential', 'Editor basics', 'Code completion', and 'Refactorings'. We won't be covering the rest, although feel free to complete them in your own time after the tutorial, if you wish.

Task 4 - A small test

◁ **Task**

Now let's try some of the IntelliJ features we just learned about with a little less guidance.

✎ Download the **auto-refactoring.zip** project from the Inf1B course materials page and open it in IntelliJ.

In each of the small classes, see the comments that show you how to make use of automatic refactoring. Try applying the suggested actions in IntelliJ and the comments. Do they make sense? Can you think of any examples where this might be useful in your assignment?

✎ IDEs are not all powerful – they will not be able to tell if your variable or method names are

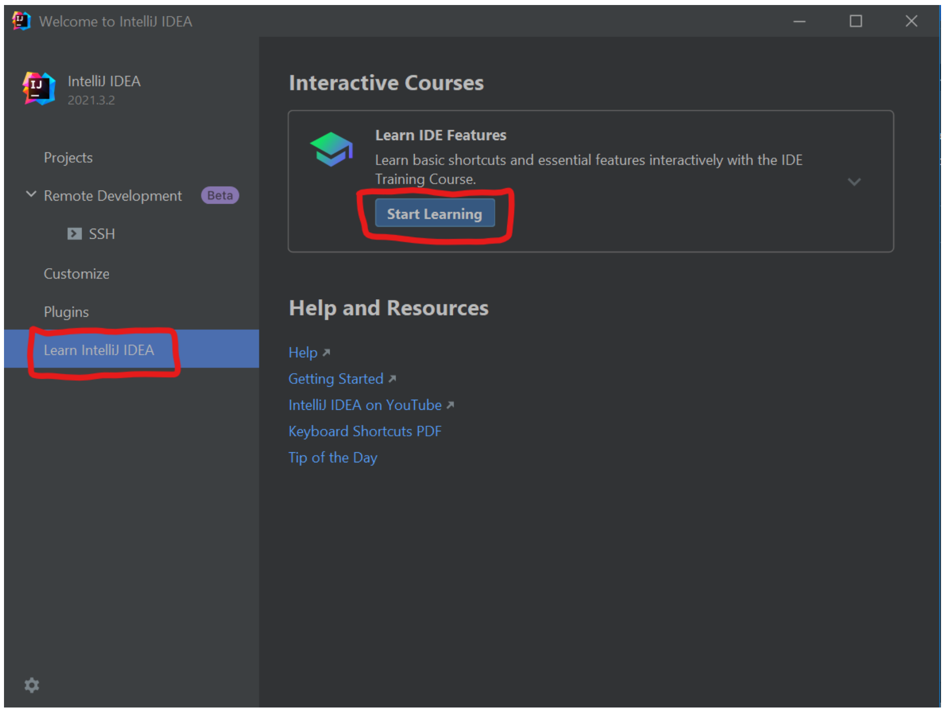

Figure 4: IntelliJ welcome menu option to launch the "Learn IDE Features" tutorial

meaningful, nor will they always pick up on code that can be refactored. While they can be immensely helpful, you still need to be aware of good code practices yourself.

✎ You can find the sample solution **auto-refactoring_solution.zip** on the Inf1B course materials page.

3 How am I supposed to know what good code is?

This is a fair question. What is good code? If we are to keep refactoring code, how can we know when to stop? Discuss with your peers to see what everyone thinks.

Since the readability and simplicity of code are somewhat subjective matters, there are many answers. The rules of thumb that are widely accepted by developers are called software engineering principles.

One such principle is DRY, which stands for Don't Repeat Yourself. In a nutshell, it means that redundant repetitiveness is bad. This applies to both code and documentation. The opposite of DRY is WET ("write everything twice" or "waste everyone's time").

DRY code uses abstractions and generalisations to make reusable methods and classes, instead of repeating them every time.

Task 5 - We can fix it!

◁ **Task**

Now that you are familiar with some refactorings, like extract variable or inline method, and hopefully know how to use IntelliJ to help you apply them, here is a challenge:

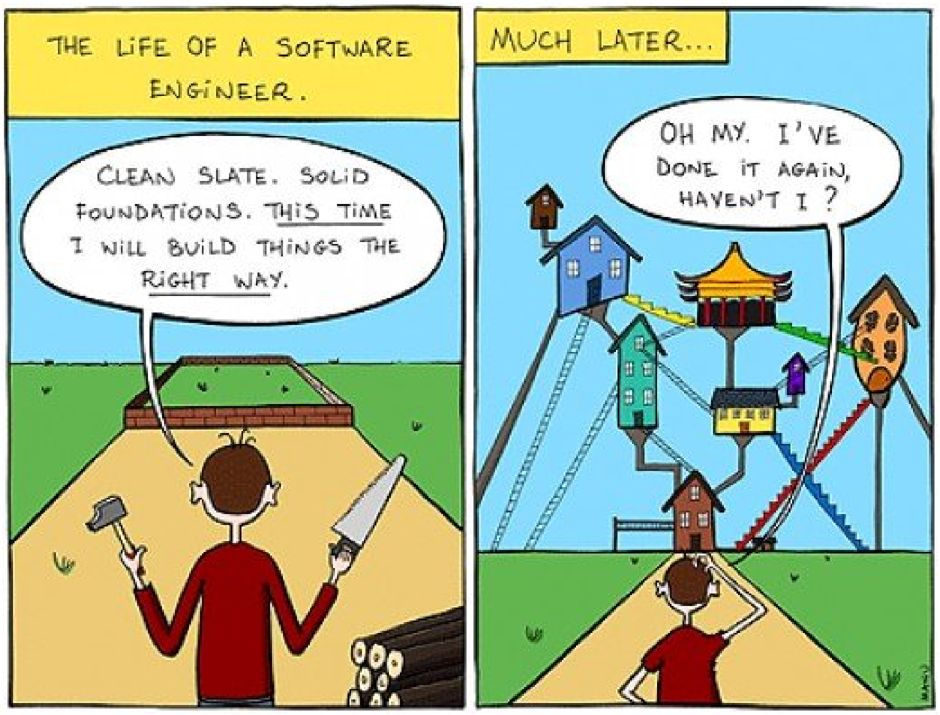
THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

Figure 5: "Life of a software engineer" by Manu Cornet © 2011 Manu Cornet

Find a partner and try to refactor this little but messy project as best you can in the next 10 minutes.

✎ Download the **refactoring-challenge.zip** project from the Inf1B course materials page and open it in IntelliJ.

In this code snippet, that estimates Pi, the first helpful thing to do is extract the code from main into a function called `estimatePi` or similar, so the code becomes more self-documenting. Bonus points for renaming the class from `Main` to something more appropriate too.

The next steps could be eliminating what is not necessary:

- the variable `y` is never used, so `int y = 0;` can go straight to the limbo
- the assignment `x = x` is useless, it can be ditched together with the whole else branch it is in
- Some further simplifications can be made that require some understanding of the code:
- the loop counter in `while (idx < num)` can be inlined by changing it to a for-loop
- the condition `x >= 0` is always true, so the ternary operator at the end is not necessary
- the variables `r, num, x, hits` can all be renamed to something more meaningful, e.g. `radius, numSamples, samplesInCircle, samples`

Finally, with better understanding of the code it can be simplified even more:

- `Math.random()* r` can be extracted to variables, since they represent x and y coordinates

- the `hits` (or `samples` if renamed) array is not necessary at all, it is only used for counting how many booleans are true, so it can be replaced by incrementing a counter on the spot rather than storing booleans and counting later

✎ You can find the full sample solution **RefactoringChallengeSolution.java** on the Inf1B course materials page.

4 Further learning

To learn more about software engineering principles, check out these Wikipedia summaries:

- `https://en.wikipedia.org/wiki/KISS_principle`
- `https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)`
- `https://en.wikipedia.org/wiki/SOLID`

There are loads more refactoring patterns that we haven't seen yet, you can learn about them here: `https://refactoring.guru/refactoring` (this website inspired the warmup exercises section of this tutorial too!)

The second year course ”INF2SEPP: Software Engineering and Professional Practice” teaches a lot about good software development practices, including refactoring and design patterns.

You can also learn about useful re-usable standard algorithms and classes in the second year ”Informatics 2 - Introduction to Algorithms and Data Structures”.