Seminar 4 - Refactoring

PV260 Software Quality

March 8, 2015

1) Class has too many responsibilities - Motivation

revisions 0 . . . 4

Methods of RegexFractals:

```
public static void main(String[] args) {
private static void launch(String[] args) {
public static boolean isPowerOfTwo(String number) {
public static boolean isValidRegexPattern(String regex) {
```

Responsibilities:

- Expected: Control the generation process on high level
- Actual: Control the process and handle args parsing

1) Class has too many responsibilities - Refactoring

revisions 0 . . . 4

Mechanics:

- Create new class InputParser
- Move validation methods from RegexFractals to InputParser
- Update calls to validation methods in RegexFractals
- Move casting logic from RegexFractals to the InputParser

2) Methods do too much - Motivation

revisions 5,6

In the method

```
public int tryParseSize(String number) {
```

we do the following:

- try to cast the String input to integer
- check that the number is at least two
- check that the number is power of two

Each of these steps can be extracted into its own method.

2) Methods do too much - Refactoring

revisions 5,6

Mechanics:

- Apply the ExtractMethod to move chunks of logic into new methods
- Name these according to their responsibilities mentioned earlier
- After every method extraction run tests to be sure nothing broke

2) Methods do too much - Refactoring

revisions 5,6

Notes:

- Goal of this refactoring is to change internal structure of the method, its signature and behavior must stay unchanged
- When naming the extracted methods make sure their intent is really just one thing, otherwise either compose the submethods further or compose the original method differently

2) Methods do too much - Refactoring

revisions 5,6

Using the Compose Method technique we also refactor the code in

```
public static void launch(String[] args) {
```

to reflect the domain concepts:

- Colorize - takes grid of signatures and returns grid of colors
- Output - takes grid of colors and outputs some file containing the fractal (or any other image) built from these colors

3) Primitive Obsession - Motivation

revisions 7 . . . 9

Instead of working with abstraction of Grid we work with an array of arrays. Semantics of the array are not clear.
(There are actually two bugs in the code which cancel out, first see signatureOf in FractalGrid, then the colorization in main and then output in AsciiConverter)

```
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        Color color = Colorizer.colorFor ... Of(i, j));
        colored[j][i] = color;
    }
}
```

- Are we iterating by row or column?
- Is the grid saved as array of rows or array of columns?

3) Primitive Obsession - Refactoring

revisions 7 . . . 9

Mechanics:

- Create abstraction of the concept we want to model
- Use existing code to implement the abstraction
- Refactor occurences of the primitive type with the new abstraction

Notes:

- We must not make FractalGrid a subclass of Grid as it doesnt fulfill the contract of Grid (e.g. we cant set elements as they are all precomputed at construction time)
- Working with abstraction will later enable us to change source of data in the Grid without having to modify client code
- Try to make the abstraction as general as possible, e.g. we are not creating a Grid single type - Color, String etc. - but generic Grid to hold any type

4) Implement One Task in Multiple Ways - Motivation

revisions 10 . . . 12

The concept of Grid colorization (taking grid of signatures and turning it into grid of colors) should not be bound to FractalGrid

- making the Colorization depend on grid of signatures makes testing easier
- alternative implementations are possible (e.g. we create checkerboard independent of the signatures)

4) Implement One Task in Multiple Ways - Refactoring

revisions 10 . . . 12

Mechanics:

- Extract the GridColorizer interface
- Crete its implementation by extracting current code from RegexFractals
- Delegate to the new RegexGridColorizer from the original method call

4) Implement One Task in Multiple Ways - Motivation

revisions 10 . . . 12

Notes:

- To judge how well the abstraction is designed (methods, their signatures...) try to come up with other different implementations of the abstraction.
- E.g. we can implement the GridColorizer using the regex or we can draw a checkerboard instead etc.
- E.g. grid could also be implemented as a `Map<Position, T>` where Position holds x and y

5) Correct Use of Inheritance - Motivation

revision 13

The PngImageConverter extends the AsciiImageConverter to get access to its validation methods. However Outputing PngImage is not a special case of returning Ascii representation of the image.

Possible solutions:

- Copy the methods and change extends AsciiConverter to implements ImageConverter to break the offending iheritance, however we create duplication
- Extract the common functionality into a superclass common for AsciiConverter and PngConverter

5) Correct Use of Inheritance - Refactoring

revision 13

Mechanics:

- Identify methods and fields used by both implementations
- Extract these into a superclass and make the current root of the hierarchy (AsciiConverter) extend the new superclass instead of the original interface, the direct implementor of the interface is now the superclass
- Extend the superclass from the rest of the descendants (PngConverter in this case)

6) Use of Generics - Motivation

revisions 14, 15

- The concept of ImageConverter should not be constrained by String, there surely exist other perfectly good formats for image data we might wish to convert to.
- The PngImageConverter always returns null because of how the ImageConverter interface is designed, if we have generic interface we can return BufferedImage instead.
- We can eliminate the side effect of conversion in PngConverter which cleans up the responsibilities of the class (now only conversion, before conversion and output)

6) Use of Generics - Refactoring

revisions 14, 15

Mechanics:

- Change the interface to see all subclasses that require change
- Change subclasses to implement the interface using the original type (String in our case), that way no other code needs to be changed now
- Refactor subclasses which need to use different type one by one

Notes:

- It is safer to first update all subclasses to use the original type so that whole project can be compiled and runs before doing any functional changes

7) Reuse common algorithm core - Motivation

revisions 16 . . . 18

The two nested for loops in Ascii and Png Converters are identical, only their contents vary.

- The order of iteration by row is only important for the AsciiConverter, Png will work both ways. By establishing strategy which makes sense in all cases (iteration by row) we relieve others of the need to figure it out themselves later.
- The algorithm will be more readable as we provide named sections for it through this refactoring
- We reduce duplication, now one only has to read the superclass to understand any subclass quickly

7) Reuse common algorithm core - Refactoring

revisions 16 . . . 18

Mechanics:

- Find the variation of method to refactor which has the most steps different from other flavors of the method
- Extract these into new well named methods
- Either pass required data into each of these individually or cretae accessors for all data required in the new methods
- Pull up common core of the algorithm to the superclass and create abstract /empty declarations for the submethods called from core
- Refactor rest of the subclasses to implement these new methods, using logic from the original code
- If the other subclasses require some behavior not in the superclass consider adding it or refactor the code to not require it

7) Reuse common algorithm core - Refactoring

revisions 16 . . . 18

Notes:

- It is good practice to make the common core method final so that subclasses cant reimplement it completely
- If this template is not suitable for a subclass it should implement /extend another class instead
- By making the hoors abstract /empty you control whether these are mandatory or optional steps respectively

8) Moving Logic Closer to Data - Motivation

revisions 19, 20

In FractalGrid2DArray we are working with two boolean values isRightQuadrant and isTopQuadrant instead of actual Quadrant object. First we have to get rid of Primitive Obsession - create the Quadrant object - and then move all logic that is dependent only on the quadrant inside this new object.

Responsibilities of the Quadrant will be:

- How to change coordinate of the cell based on its quadrant
- What to add to the cell signature based on the quadrant

As there is clearly defined domain of possible instances of the Quadrant, we make it an enum.

8) Moving Logic Closer to Data - Refactoring

revisions 19, 20

Mechanics:

- Define the new class, for now we only need four enum constants for the four quadrants
- Find occurences of the isRight and isTop and while preserving the functionality turn checks on these to checks on Quadrant
- When Quadrant is used throughout the code start moving behavior inside it

The result is that we ask the Quadrant for answer instead of determining the answer based on what the quadrant is

Notes:

- It is possible to switch on enum, while it is better to have the logic directly inside it this is great for intermediate steps

8) Moving Logic Closer to Data - Refactoring

revisions 19, 20

```
if (isRightQuadrant && isTopQuadrant) {
    signatureAddend = '1';
} else if (!isRightQuadrant && isTopQuadrant) {
    signatureAddend = '2';
    ...
```

changes to

```
switch (quadrant) {
    case UPPER_RIGHT:
        signatureAddend = '1';
        break;
    case UPPER_LEFT:
        signatureAddend = '2';
        break;
    ...
```

9) Primitive Obsession in Signatures

revision 21

- The QuadrantSlice constructor looks like:

```
public QuadrantSlice(int size, int parentX, int parentY,
    String parentSignature, Quadrant quadrant){
```

- Except Quadrant those are actually about all of the parent's attributes, so we can pass the parent instead.
- Since we are working with a recursive datastructure we dont have to define getters, we can access the attributes on parent directly.
- Notice that the size is actually halved before the constructor is called, so we dont receive the parent size but our size.
- We have to first store the size in QuadrantSlice as a field and then halve it ourselves in the constructor.

10) Replacing flags by inheritance - Motivation

revisions 22 . . . 24

The flag isLeaf in QuadrantSlice effectively packs two distinct sets of behavior into one object.

- Slice which doesnt have any more children, always has size 1 and is at the bottom of the hierarchy
- Slice which always has 4 children, one for each further quadrant division and only serves as an intermediate step

Better than by a flag this distinction in responsibility can be represented by two actual classes - Leaf and Splittable

10) Replacing flags by inheritance - Refactoring

revisions 22 . . . 24

Mechanics:

- Create the new classes and make them extend the original
- In the constructors call the super for common part and extract the type specific (note that we still have the isLeaf flag for all the methods to work, set it in the new constructors)
- Create methods dependant on the isLeaf flag abstract and move implementation specific parts to respective subclasses
- Optionally repeat the steps if there is still some sort of behavior selection (in out case Root vs. Intermediate Fragments)