COM S 362
Object-Oriented Analysis & Design

Refactoring and Code Smells
Bottom-Up Design

Reading

Fowler, Martin. Refactoring: Improving the Design of Existing Code, 2018.

Read Chapter 3 sections: the introduction, "Duplicate Code", "Long Function", "Long Parameter List", "Primitive Obsession", "Large Class" and "Data Class". pp. 71-74, 78-79, 82-83.

Refactoring

- "***Refactoring*** *is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior*." –Martin Fowler
- The goal of refactoring is to improve the design (structure) without changing the behavior
- Refactoring is making incrementation changes to existing code
- Refactoring ≠ rewriting (throwing away and staring fresh) the code

Common Refactorings

- Fowler has created a catalog of common refactorings
- Example: Introduce Parameter Object

```
double amountInvoiced(Date startDate, Date endDate) { … }
```

```
doube amountInvoiced(DateRange dateRange) { … }
```

  - Design principle: the refactoring creates a new *abstraction* so that the method caller does not need to know as many details

Code Smells

- Code smells are indications that code could be improved
- There are not precise metrics (e.g., all methods over 10 lines are bad)
- Good judgment required, the code smell only means something to look into
- For each code smell Fowler suggests common refactorings

Long Function

- A long function is difficult to understand
- Small functions are easy to name and therefore self documenting
- Long functions often violate *separation of concerns*
- *"Functions should do one thing. They should do it well. They should do it only."* —Robert C. Martin, Clean Code

```
private void exec() throws FileNotFoundException {
        long seed;
        int length;
        long startTime;
        long endTime;
        int[] masterData;

        do {
            // input data source
            prompt("Data source? (0 = file, 1 = generated)");
            if (getInput() == 0L) {
                // get data from file
                // process data from file
                // input file name
                prompt("File name? ");
                String fname = getFileName();
                masterData = readFile(fname);
            } else {
                // get experiment parameters
                prompt("How many values?");
                length = getInput();
                prompt("What seed? 0 = default ");
                seed = getInput();

                // generate random data
                masterData = new int[length];
                setRandomSeed(seed);
                for (int i = 0; i < length; i++) {
                    masterData[i] = nextRandomInt();
                }

            }
            runSorts(masterData);
            prompt("Another experiment? (0 = no, 1 = yes) ");
        } while (getInput() == 1L);
}
```

exec( ) is a super generic name, hard to be more specific because of how many things it does

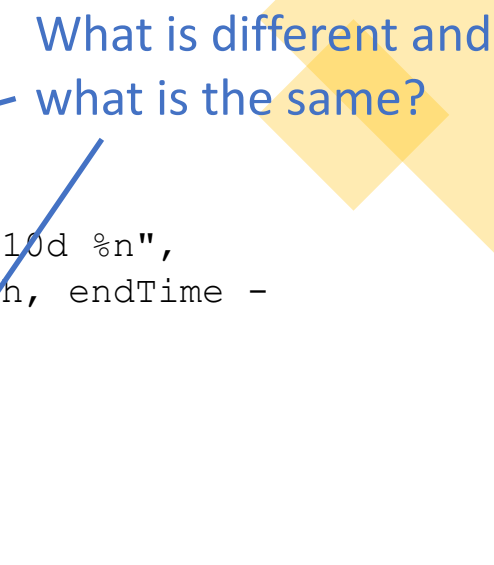
So many concerns in one function: user I/O, reading a file, generating random data, all packed into a control loop

Duplicate Code

- Duplicated code (at any level, e.g., block, function, class, module) is a sign of lacking *abstraction*
- It hurts understandability and maintainability of the code
- When you read duplicated code you have to be careful to check for intentional or unintentional differences

```
// perform selection sort
localData = makeCopy(masterData);
startTime = getTime();
selectionSort(localData);
endTime = getTime();
System.out.format("%20.20s %10d %10d %n",
"selection Sort", localData.length, endTime -
startTime);

// perform merge sort
localData = makeCopy(masterData);
startTime = getTime();
mergeSort(localData);
endTime = getTime();
System.out.format("%20.20s %10d %10d %n", "merge
Sort", localData.length, endTime - startTime);
```



Long Parameter List

- If you need to provide a function with a lot of arguments it may be that
  - The function is not following the rule of “do one thing” and suffers from *separation of concerns*, or
  - The parameters should be part of an *abstraction*

```
calculateStatistics(customer, null,
    true, false, 100);
```

```
calculateStatistics(customer,
    statOptions);
```

Primitive Obsession

- Using primitives makes change more difficult
- We use them for efficiency, but we should use *abstractions* to hide them

```
String userId = “98749382”;
int userRole = 1;
```

Does not make it easy to change in future

Large Class

- Sign of violating *separation of concerns*
- A class should not be assigned to many different responsibilities
- May also be opportunity to introduce a hierarchy, separate code into multiple subclasses

Data Class

- A class with only fields and getters and setters
- May be a sign of poor *encapsulation*