SOFTWARE DESIGN
COSC 4353/6353

Dr. Raj Singh

OUTLINE

What is Refactoring?

Code Smells

Why Refactoring?

Techniques

IDEs

WHAT IS REFACTORING?

- "Art of improving the design of existing code"
- Disciplined technique for restructuring an existing body of code.
- Altering internal structure without changing external behavior of code.
- Agile teams maintain and extend code a lot from iteration to iteration.
- Without continuous refactoring code tends to rot (bad code smell).

CODE HYGIENE

A popular metaphor for refactoring is cleaning the kitchen as you cook.

In any kitchen you will typically find that cleaning and reorganizing occur continuously.

Someone is responsible for keeping the dishes, the pots, the kitchen itself, the food.

The refrigerator is cleaned and organized from moment to moment.

Without this, continuous cooking would soon collapse.

WHAT IS CODE SMELL?

- Code smell is any symptom in the code that possibly indicates a deeper problem
- Code smells are usually not bugs or broken code
- They don't currently prevent the program from functioning
- They indicate weaknesses in design
- Disaster waiting to happen
- Slowing down development or increasing the risk of bugs or failures in the future

EXAMPLE – ADD NUMBERS

```
public class BadCodeExample {
    public static void main(String[] args) {
        String fileName = "C:/WorkSpace/numbers.txt";
        try {
            Scanner input = new Scanner(new File(fileName));
            ArrayList<Double> numbers = new ArrayList<Double>();
            while(input.hasNext()){
                numbers.add(input.nextDouble());
            }
            System.out.println("Total: "+numbers);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
```

WHAT BAD SMELL DO YOU NOTICE?

- Example deals with files
- What could go wrong if file doesn't exist?
- Bad filename
- File is locked while code executes
- Someone else needs access to the file
- Wait a minute … we didn't close the file …
- Only one method that performs all steps

COMMON CODE SMELLS

Repeating Code:	duplicated code
Coding Standards:	too many parameters, long method, large class, naming conventions
Feature envy	a class that uses methods of another class excessively
Dependency:	a class that has dependencies on implementation details of another class.
Lazy class / Freeloader:	a class that does too little.

COMMON CODE SMELLS

Contrived complexity:	forced usage of overly complicated design patterns where simpler design would suffice.
Excessively long or short identifiers:	not following coding standards
Excessive use of literals:	these should be coded as named constants
Ubercallback:	a callback that is trying to do everything
Complex conditionals	Complex if-else blocks

COMMON CAUSES OF CODE SMELLS

The habit of postponing code fixes	Insisting on a one-liner solution	Ignoring the warnings	Not my code / it's my code
Excessive use of design patterns without knowing the usage	Hard coded values	No unit testing	Not following standards

WHY REFACTORING?

Refactoring is usually motivated by noticing a code smell

Once recognized, such problems can be addressed by refactoring the source code

Or transform code into a new form that behaves the same as before but that no longer "smells"

Failure to perform refactoring can result in accumulating technical debt

Why fix what's not broken?

A software module

Should function its expected functionality

- It exists for this

It must be affordable to change

It will have to change over time, so it better be cost effective

Must be easier to understand

Developers unfamiliar with it must be able to read and understand it

BUT WHY?

BENEFITS

Maintainability:	It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.
Extensibility:	It is easier to extend add new features
Reusability:	Reuse of some functionality without too much coding.
Quality:	Good quality code that doesn't smell.
Optimization	Maintain high standards and simplicity

BEFORE WE REFACTOR

A solid set of automatic unit tests is needed

The tests are used to demonstrate that the behavior of the module is correct before the refactoring

If a test fails, then it's generally best to fix the test first

Understand the impact of refactoring, dependencies, and impact on other parts of the system

AFTER REFACTORING

The tests are run again to verify the refactoring didn't break the tests

Of course, the tests can never prove that there are no bugs, but the important point is that this process can be cost-effective

Good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough

Do regression test of complete application to make sure other parts are still working.

The process is an iterative cycle of making a small program transformation.

Testing it to ensure correctness, and making another small transformation.

If at any point a test fails, the last small change is undone and repeated in a different way.

Through many small steps the program moves from where it was to where you want it to be.

In order for this very iterative process to be practical, the tests have to run very fast.

SO HOW DO WE DO IT?

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 type

checking code with State/Strategy

Simplify Conditions

replace conditional with polymorphism

TECHNIQUES - MORE ABSTRACTION

TECHNIQUES - MORE LOGICAL PIECES

Componentization — break code down into reusable semantic units

Extract Class — move parts of the code from an existing class into a new class

Extract Method — turn part of a larger method into a new method

Break down code — smaller code is more easily understandable

TECHNIQUES – MOVE CODE

Move Method or Move Field — move to a more appropriate class or method

Rename Method or Rename 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

REFACTORING AUTOMATION IN IDES

- Refactoring is much easier to do automatically than it is to do by hand
- More Integrated Development Environments (IDEs) are building in automated refactoring support
- Select the code you want to refactor, pull down the specific refactoring you need from a menu, and the IDE does the rest
- You are prompted appropriately by dialog boxes for new names for things that need naming, and for similar input.
- You can then immediately rerun your tests to make sure that the change didn't break anything.
- If anything is broken, you can easily undo the refactoring and investigate

CODE REVIEW

Code review is systematic examination (often known as peer review) of source code.

It is intended to find and fix mistakes overlooked in the initial development phase.

It improves both the overall quality of software and the developers' skills.

Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.

TYPES OF REVIEW

Pair programming

Two developers code together and review each others code and provide feedback

Formal code review

Involves a careful and detailed process with multiple participants and multiple phases

Lightweight code review

Conducted as part of the normal development process

LIGHTWEIGHT CODE REVIEW

Over-the-shoulder

One developer looks over the author's shoulder as the latter walks through the code.

Email pass-around

Email code to reviewers automatically after code is committed.

Tool-assisted code review

Authors and reviewers use specialized tools designed for peer code review.

```java
public class CodeRefactoringExample {
        String fileName = "C:/WorkSpace/numbers.txt";
        Scanner input;

        public static void main(String[] args) {
            // TODO Auto-generated method stub
            CodeRefactoringExample cre = new CodeRefactoringExample();
            cre.openFile(cre.fileName);
            try {
                System.out.println("Total: "+cre.addNumbers());
            } catch (Exception e) {
                e.printStackTrace();
            }finally{
                cre.closeFile();
            }

        }
```

REVISITING THE EXAMPLE

```
    public void openFile(String filename){
        try {
            input = new Scanner(new File(fileName));
            System.out.println("File opened for processing!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public double addNumbers(){
        double total = 0d;
        while(input.hasNext()){
            total += input.nextDouble();
        }
        return total;
    }

    public void closeFile(){
        if(input != null)
            input.close();
        System.out.println("File closed!");
    }
}
```

REVISITING THE EXAMPLE

TO RE-FACTOR OR NOT TO RE-FACTOR?

To

- Anytime you can cleanup the code
- To make it readable, understandable, simpler
- You are convinced about the change
- Before adding a feature or fixing a bug
- After adding a feature or fixing a bug

Not to

- Not for the sake of refactoring
- When the change will affect too many things
- When change may render application unusable
- In the middle of adding a feature or fixing a bug
- You don't have unit tests to support your change

HOMEWORK

Review class notes.

Additional reading:

Examples of UML diagrams

Start a discussion on Google Groups to clarify your doubts.