CODE SMELLS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

LEARNING OUTCOMES

Understand what a code smell is and some common categories of code smells.

Understand common ways to address code smells through refactoring.

WHAT IS A CODE SMELL?

These slides are based on

https://refactoring.guru/refactoring/smells

WHAT IS A CODE SMELL?

- When the refrigerator smells, the food might not have gone bad yet. But it will soon!
- When there is a code smell, the program isn't bad yet. But the more code you add, the more it will have to accommodate the "smelly" code, causing your program to be more and more difficult to handle.
- Some problems can be fixed any time:
  - an inconvenient variable name can be changed using "search and replace"
  - the file path leading to a csv file where you store data can be changed at any time
- Code smells should be fixed as soon as you see them, to avoid extra work later.

REFACTORING

- https://refactoring.guru/refactoring
- "Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design."

- https://refactoring.guru/refactoring/what-is-refactoring
- Clean Code is well formatted, well documented, contains good variable names, does not contain duplicate code, has nice small methods and reasonably sized classes, no magic numbers, and passes all its tests.

CATEGORIES OF CODE SMELLS

- Bloaters – too much code
- Object Orientation Abusers – can be improved by changing use of inheritance, composition, or by redistributing responsibilities
- Change Preventers – features that make it difficult to extend or update the code
- Dispensables – things that can be deleted or combined
- Couplers – the opposite of "lessening dependencies"
- Etc...

LONG METHOD: EXAMPLE OF A PROBLEM AND SOLUTIONS

- https://refactoring.guru/smells/long-method
- Signs and Symptoms
  - "A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions."
- Reasons for the Problem
  - Mentally, it’s often harder to create a new method than to add to an existing one: “But it’s just two lines, there’s no use in creating a whole method just for that...”
  - Which means that another line is added and then yet another, giving birth to a tangle of spaghetti code...
- Possible Fixes
  - **Extract Method** to make the original method shorter
  - **Replace Temp with Query**, **Introduce Parameter Object** or **Preserve Whole Object**, if variables or parameters make it difficult to extract method.
  - **Replace Method with Method Object** to encapsulate the method, if you cannot make it smaller.
  - If an if statement prevents such a move, use **Decompose Conditional**. If a loop prevents the move, use **Extract Method**.

NOTES ABOUT SMELLS – PART 1

- Switch statements
  - only a problem if they are complicated or use "instanceof"
- "Large class" can mean
  - A class that is inconveniently long to read and work with, or
  - a class that breaks the Single Responsibility Principle
- "Alternative Class with Different Interfaces”
  - happens when different people solve the same problem independently in different parts of the code.
  - This can be prevented or fixed quickly during code reviews, such as during a pull request, when you become aware of the problem

NOTES ABOUT SMELLS – PART 2

- "Divergent Change" and "Shotgun Surgery" sound similar, but are not the same problem
  - Divergent Change means...
  - Shotgun Surgery means...
- "Parallel Inheritance Hierarchies" are not just the existence of two inheritance hierarchies that look similar. To have this code smell, an addition of one class in one hierarchy forces a new class be added to the other inheritance hierarchy.
  - Example: each view class corresponds to a different presenter class
- Isn't an Input Data Object a Data Class?

NOTES ABOUT SMELLS – PART 3

- How do I know if a class is "lazy"?
- Isn't "Speculative Generality" the same as "I haven't finished my program yet"?
- "Duplicate code" can easily be avoided by not copying and pasting code (copy-pasta).
- Aren't "Message Chains" just methods calling other methods?
  - No.
  - Example: obj1.method1().method2().method3().method4(); Which class contains method4? What if you want to refactor it?

REFACTORING AND THE OPEN/CLOSED PRINCIPLE

- The Open/Closed Principle says that we should write code in a way that lets us change as few things as possible when adding new features.
- Refactoring amounts to making changes to the code. Doesn't that go against the principle?
- We use refactoring <u>now</u> to get the code to a point where we can avoid making too many changes when we add new features <u>later</u>.

WHEN TO REFACTOR?

- Whenever you see a place where your code could be improved, or when you identify a code smell.
- When you find yourself doing the same thing for the third time.
- When trying to understand someone else's code, but it is confusing.
- When trying to fix a bug, but the code is difficult to follow.
- During or immediately after a code review, when conducted with at least one of the authors of the code.

ADDING NEW FEATURES ≠ REFACTORING

- When you refactor, the functionality of your code should not change.
- Refactoring should change the readability and understandability of your code.
- If you refactor before adding new features, it should be easier to add the new features.
- Working "smart" now can save you hard work later.
- The goal is to make it <u>easier</u> to grow your program and <u>save time</u> for you and your team.

CATEGORIES OF CODE SMELL FIXES

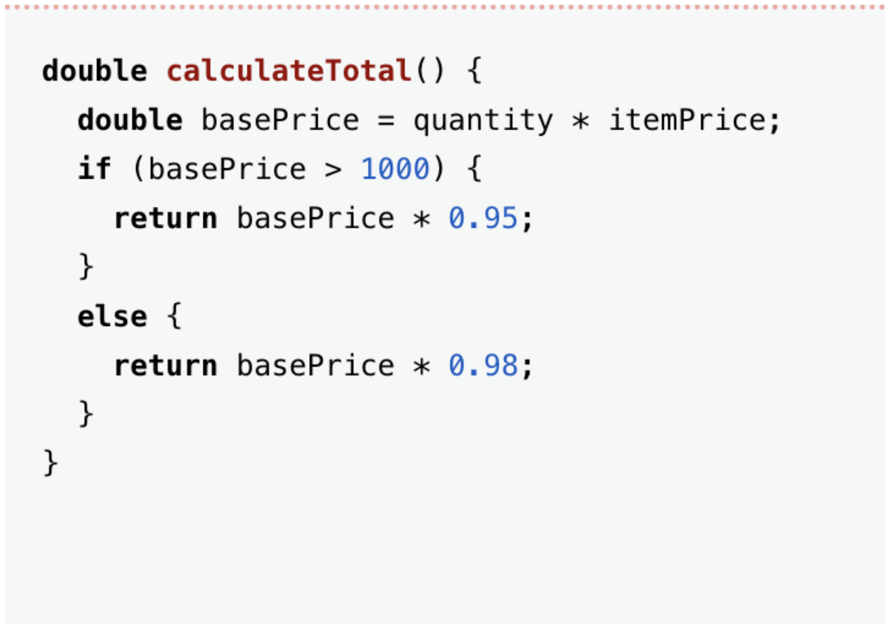
The solution to a code smell is almost always "refactor".

https://refactoring.guru/refactoring/techniques

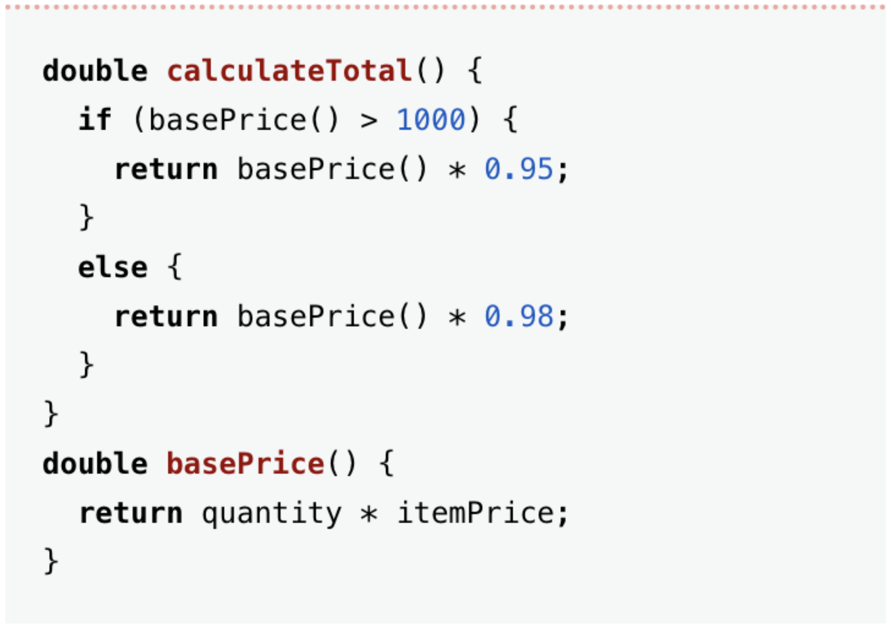
- Composing Methods
- Moving Features Between Objects
- Organizing Data
- Simplifying Conditional Expressions
- Simplifying Method Calls
- Dealing With Generalization

COMPOSING METHODS EXAMPLE

- "Replace Temp With Query"
- https://refactoring.guru/replace-temp-with-query
- You can encapsulate a temporary variable assignment by putting it in a separate method and querying that method.



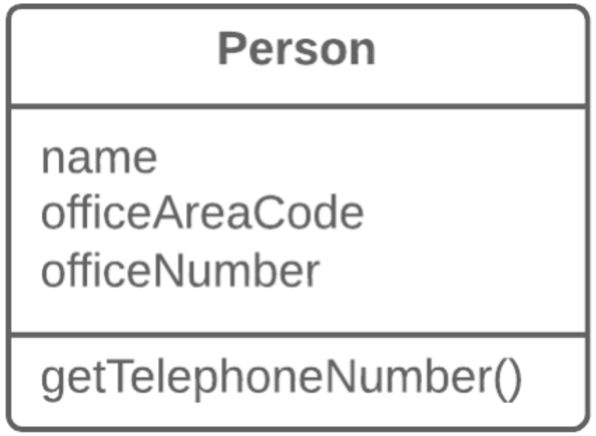
```
double calculateTotal() {
  double basePrice = quantity * itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
}
```

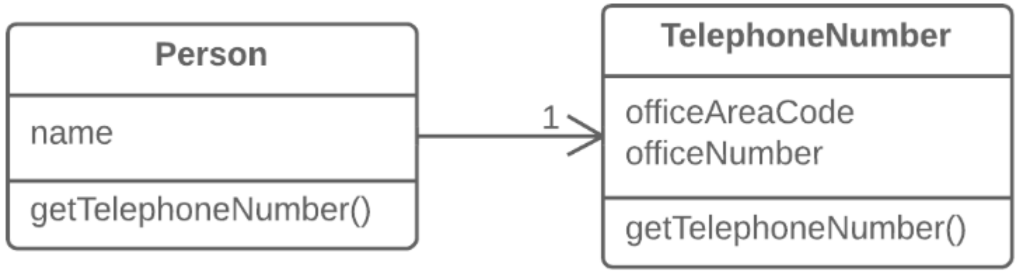


```
double calculateTotal() {
  if (basePrice() > 1000) {
    return basePrice() * 0.95;
  }
  else {
    return basePrice() * 0.98;
  }
}
double basePrice() {
  return quantity * itemPrice;
}
```

MOVING FEATURES BETWEEN OBJECTS EXAMPLE

- "Extract Class"
- https://refactoring.guru/extract-class
- When a class does too many things, you can take a subset of the functionality that belongs together to create a new class. The new class can become a variable inside the original.





ORGANIZING DATA EXAMPLE

- Replace Magic Number with Symbolic Constant
- https://refactoring.guru/replace-magic-number-with-symbolic-constant
- Try not to hardcode values in your methods. It is easier to find and change their values later, if they occur as constants.

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

```
static final double GRAVITATIONAL_CONSTANT = 9.81;

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

SIMPLIFYING CONDITIONAL EXPRESSIONS EXAMPLE

- Replace Conditional with Polymorphism
- https://refactoring.guru/replace-conditional-with-polymorphism
- If statements or switch statements based on "instanceof" make extension difficult.

We saw an example of this in the solid.pdf slides for the Open/Closed principle, with the AreaCalculator class requiring a list of Rectangles. We created a Shape super class and relocated the "area" method inside each subclass of Shape.

```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCocon
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}

// Somewhere in client code
speed = bird.getSpeed();
```

Computer Science
UNIVERSITY OF TORONTO

SIMPLIFYING METHOD CALLS EXAMPLE

- Preserve Whole Object
- https://refactoring.guru/preserve-whole-object
- If you must call multiple getters from an object and then pass those values on as arguments, don't. Just pass the original object.
- This sometimes conflicts with Clean Architecture. For this reason, we can have Input Data Objects and Output Data Objects.

```java
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```java
boolean withinPlan = plan.withinRange(daysTempRange);
```

DEALING WITH GENERALIZATION EXAMPLES

- Replace Inheritance with Delegation
- https://refactoring.guru/replace-inheritance-with-delegation
- Replace Delegation with Inheritance
- https://refactoring.guru/replace-delegation-with-inheritance
- If a subclass S does not use all of the inherited methods from its super class T, then S can contain a variable of type T instead.
- If a class S has nothing but simple methods that call methods from variable T, then you can consider making T a superclass of S.
- Do not do the previous refactoring if it makes more sense to have a Façade, or if there is no "is-a" relationship between S and T.

CODE SMELLS AND PROMPTING: AN EXAMPLE

- Imagine you want to create a nested structure representing the directories and files on your computer, including the sizes — the number of bytes each file or directory takes up.
    - The size of a directory is the sum of the sizes of all its contents (recursive) plus 100 bytes for the directory itself.
- What a great final exam question!
    - Too bad we already used it back in December 2019 …

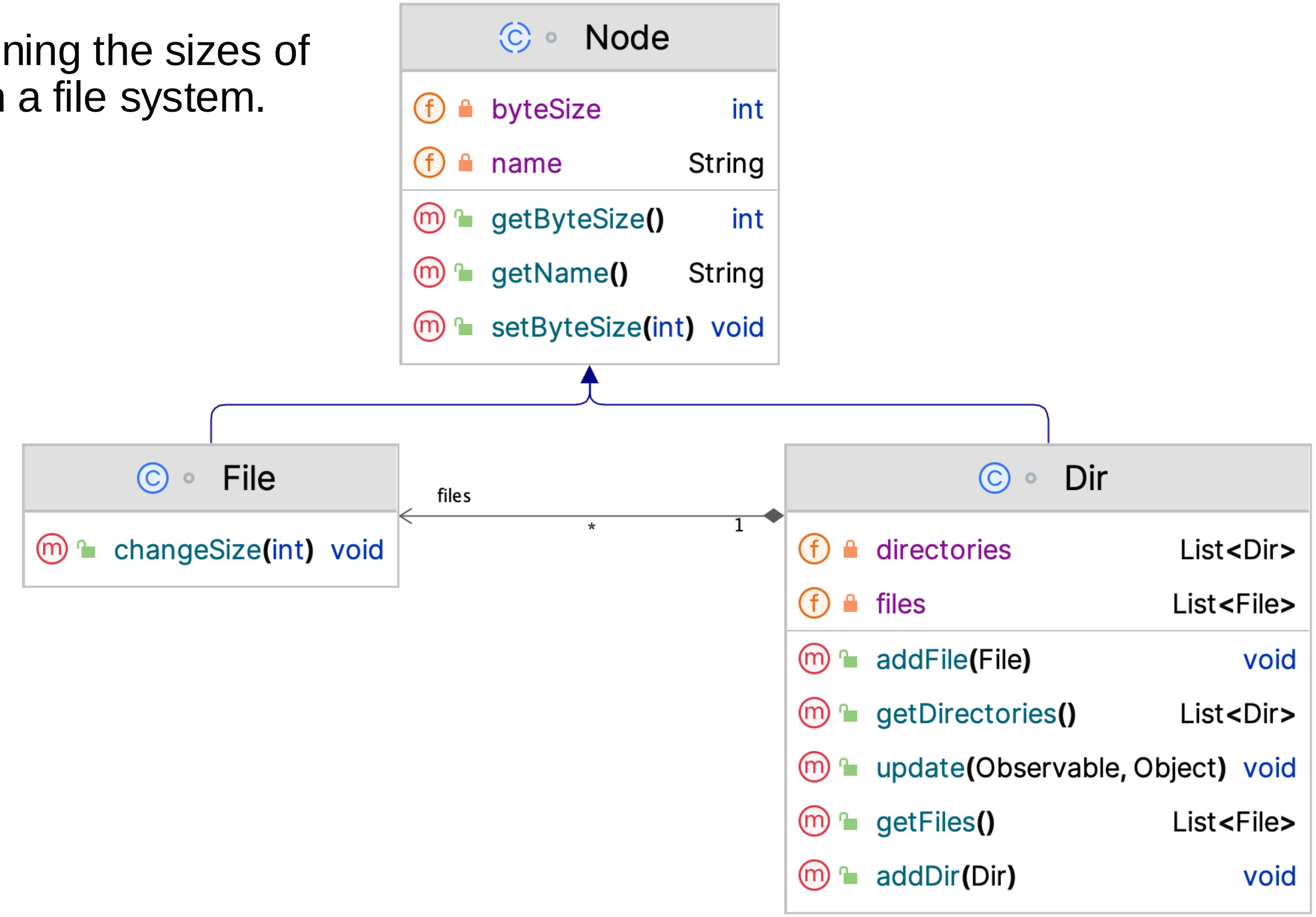
What is the size of the root directory?

```
Dir root = new Dir("root");
Dir child1 = new Dir("dir1");
Dir child2 = new Dir("dir2");
File child3 = new File("f1.txt", 10);
Dir grand1 = new Dir("dirdir");
File grand2 = new File("g1.txt", 20);

root.addDir(child1);
root.addDir(child2);
root.addFile(child3);
child2.addDir(grand1);
child2.addFile(grand2);
```

PROMPTING FOR CODE SMELLS

- Consider this code for maintaining the sizes of nodes (files and directories) in a file system. Analyze it for code smells.



PROMPTING FOR CODE SMELLS

```java
abstract class Node extends Observable {
    private String name;
    private int byteSize;

    public Node(
            String name, int byteSize) {
        this.name = name;
        this.byteSize = byteSize;
    }

    // getters omitted

    public void setByteSize(int byteSize) {
        int oldSize = this.byteSize;
        this.byteSize = byteSize;
        setChanged();
        notifyObservers(byteSize - oldSize);
    }
}
```

```java
class File extends Node {

    public File(String name, int size) {
        super(name, size);
    }


    public void changeSize(int newSize) {
        setByteSize(newSize);
    }

    @Override
    public void update(
            Observable o, Object arg) {
        // File objects do not observe anything
        // so this should never be reached.
        throw
            new UnsupportedOperationException();
    }
}
```

PROMPTING FOR CODE SMELLS

```
class Dir extends Node implements Observer {

    private List<Dir> directories =
        new ArrayList<>();
    private List<File> files = new ArrayList<>();

    public Dir(String name) {
        super(name, 100);
    }

    // getters omitted

    public void addDir(Dir info) {
        info.addObserver(this);
        directories.add(info);
        setByteSize(
            getByteSize() + info.getByteSize());
    }

    public void addFile(File info) {
        info.addObserver(this);
        files.add(info);
        setByteSize(
            getByteSize() + info.getByteSize());
    }

    @Override
    public void update(
            Observable o, Object sizeDelta) {
        setByteSize(
            getByteSize() + (Integer) sizeDelta);
    }

} // end of Node class
```

PROMPTING FOR CODE SMELLS: COPILOT OUTPUT

- Duplicated Code
  - The addDir and addFile methods in the Dir class have similar logic. This duplication could be refactored into a single method to reduce redundancy.
- Large Class
  - The Dir class has multiple responsibilities, including managing directories, files, and observing changes. This violates the Single Responsibility Principle (SRP). Consider breaking it down into smaller classes.
- Inappropriate Intimacy
  - The Dir class directly manipulates the internal state of Node objects by calling setByteSize. This tight coupling can make the code harder to maintain and test.
- Primitive Obsession
  - The use of List without generics (e.g., List directories = new ArrayList&lt;&gt;();) can lead to type safety issues. Use parameterized types to avoid this.
- Violation of Liskov Substitution Principle (LSP):
  - The File class's update method throws an UnsupportedOperationException. This violates the LSP as subclasses should be substitutable for their base classes without altering the correctness of the program.
- Magic Numbers
  - The Dir constructor initializes the size to 100 without explanation. This magic number should be replaced with a named constant to improve code readability.

PROMPTING FOR CODE SMELLS

- Observer Pattern Misuse:
  - The Node class implements Observable, and Dir implements Observer. This can lead to confusion and tightly coupled code. Consider separating these concerns.
- Lack of Encapsulation:
  - The Dir class exposes its internal lists through getDirectories and getFiles methods. This can lead to external code modifying these lists directly, breaking encapsulation.
- Inconsistent Naming:
  - The method names could be more descriptive. For example, addDir and addFile could be renamed to addDirectory and addFile for consistency and clarity.
- Lack of Type Safety:
  - The List fields in the Dir class should use generics to ensure type safety, e.g., private List<Dir> directories = new ArrayList<>();.