# SOEN 333 Software Testing
# Homework (Preparation for assignment)

## Background.

In this homework, you will implement a static analysis tool to detect bugs in source code. Static analysis tools are commonly used in the continuous software delivery process to ensure quality of your software.

You will implement this tool in Java using Spoon, an open-source tool for source code analysis and transformation. To understand the Spoon API, refer to the documentation linked below:

- Spoon, an open-source tool for source code analysis and transformation in Java.

To understand the API, you need to read about Spoon API, visitor pattern and abstract syntax tree.

- Link to documentation for Spoon: https://inria.hal.science/hal-01078532/document
- You do not need to read all the documentation. It is only here for reference.

Additionally, some of the patterns you work on will appear in an upcoming assignment. By completing this homework, you will cover some of the required work for that assignment.

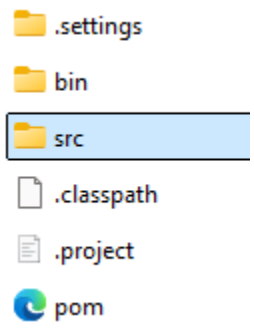
## Homework Description.

In this homework, you will detect various faults and bad design patterns. Note that bad design patterns refer to code that is logically flawed or inefficient, which may not necessarily cause runtime exceptions but can negatively affect the maintainability and readability of the code. Below are the patterns you will detect:

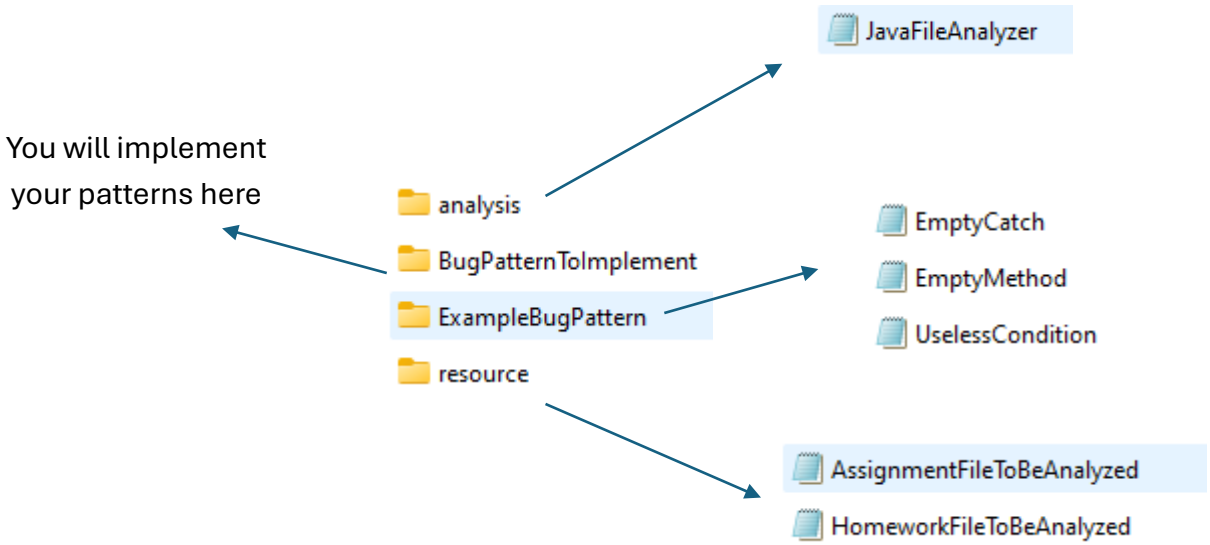
- Division by zero (causes runtime error)
- Empty catch block (bad design pattern)
- Empty if statement (bad design pattern)
- Empty method (bad design pattern)
- If statement always evaluates to true (may cause runtime error)
- Null pointer exception (causes runtime error)

Below, we will discuss the structure of the repository and explain the purpose of each file.

**Step 1: Navigate to the *analysis* file and examine the following directory structure.**




**Step 2: Go to *src* and you will see the following directory:**




- *Resource* directory contains a single file *HomeworkFileToBeAnalyzed.* This file includes the faults and bad design patterns (as discussed earlier) that you will need to detect using static analysis.
- *analysis* directory is where your main program resides. It will call file under *ExampleBugPattern* and start analyzing the *HomeworkFileToBeAnalyzed.*
- *ExampleBugPattern* contains three files:
    - *EmptyCatch* contains code to analyze *HomeworkFileToBeAnalyzed* and detect code containing empty catch block which is a bad design.
    - *EmptyMethod* contains code to analyze *HomeworkFileToBeAnalyzed* and detect code containing empty function which is a bad design.
    - *UselessCondition* contains code to analyze *HomeworkFileToBeAnalyzed* and detect code containing useless condition inside if statement, which may cause runtime error if unwanted code is always executed.

**Step 3: Open the *HomeworkFileToBeAnalyzed*:**

```java
import java.io.File;

public class HomeworkFileToBeAnalyzed {

    void method1() {
        int x = 3;
        try {
            x = x + 1;
        } catch (java.lang.Exception e) {
            // Bad Design: Empty Catch Block
        }
    }

    void method2() {
        if (true) { // Bad Design: Always evaluates to true
            // Bad Design: Empty If Statement
        }

        int x = 0;
        if (x < 3) {
            // Bad Design: Empty If Statement
        }
    }

    void method3() {
        // Bad Design: Empty Method
    }

    void method4() {
        int x = 10;
        x = x / 0; // Fault: DivisionByZeroException
    }

    void method5() {
        String text;
        int length = text.length(); // Fault: NullPointerException
    }

    void method6() {
        try {
            int[] numbers = new int[5];
            int result = 10 / 0; // Fault: Division by zero
        } catch (ArithmeticException e) {
            System.out.println("An exception occurred");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("An exception occurred");
        }
    }

}
```

- As you can see from the comments, the file contains bad design patterns and some faults. Your goal is to use the Spoon library to detect these patterns. As discussed in the lecture, you will convert this file into an Abstract Syntax Tree (AST), and then traverse the nodes to find these bug patterns.

**Step 4: Open the *JavaFileAnalyzer:***

```
1  package analysis;
2
3  import java.nio.file.Paths;
14
15 public class JavaFileAnalyzer {
16     //The Spoon Launcher (JavaDoc) is used to create the AST model of a project
17     final static Launcher launcher = new Launcher();
18     public static void main(String[] args) {
19         //Add java file to be parsed in spoon
20         launcher.addInputResource(Paths.get(System.getProperty("user.dir"),
21                 "src", "resource",
22                 "HomeworkFileToBeAnalyzed.java").toString());
23         // Parse the java file into AST
24         launcher.getEnvironment().setCommentEnabled(false);
25         launcher.getEnvironment().setIgnoreDuplicateDeclarations(true);
26         launcher.getEnvironment().setCopyResources(false);
27         launcher.getEnvironment().setIgnoreSyntaxErrors(true);
28         launcher.run();
29
30         // Create a rule
31         final EmptyCatch rule1 = new EmptyCatch();
32         final UselessCondition rule2 = new UselessCondition();
33         final EmptyMethod rule3 = new EmptyMethod();
34
35         // run the rules
36         addRuleToAnalyze(rule1);
37         addRuleToAnalyze(rule2);
38         addRuleToAnalyze(rule3);
39     }
40     /**
41      *
42      * @param rule: add your rule
43      */
44     public static void addRuleToAnalyze(AbstractProcessor rule) {
45         final Factory factory = launcher.getFactory();
46         final ProcessingManager processingManager = new QueueProcessingManager(factory);
47         processingManager.addProcessor(rule);
48         processingManager.process(factory.Class().getAll());
49     }
50
51 }
52
```

- ***Launcher*** (line 17) is the most fundamental API in spoon. It was used to create the AST model of the project. Conceptually, we are abstracting the source code into an AST, which serves as an intermediate representation for analysis.
- We then set some configuration in the ***Launcher***:
    - ***addInputResource*** (line 20)***:*** This API allows us to specify the file we are analyzing, which is ***HomeworkFileToBeAnalyzed***.
    - ***getEnvironment.setCommentEnabled(false)*** (line 24)**:** This API enables us to ignore comments while parsing the source code, meaning our abstract syntax tree will not include comments. We do this because comments are not relevant for our static analysis.

- ***getEnvironment.setIgnoreDuplicateDeclarations (true)*** (line 25)**:** This API enables us to ignore duplicate declaration. We do this because sometimes developers may duplicate method or variable declaration, which may lead to errors. You do not need to worry about this for the homework.
- ***getEnvironment.setCopyResources (false)*** (line 26)**:** This API enables us to save the analyzed model from launcher into cache. However, you do not need to worry about this for the purpose of the homework.
- ***getEnvironment.setIgnoreSyntaxErrors (false)*** (line 27)**:** This API enables us to ignore syntax error while parsing the source code. You do not need to worry about this for the purpose of the homework.

- ***Run*** (line 28): Executes the ***launcher*** and generates the AST.
- In lines 31 to 33, we instantiate the rules that we have implemented. For example, one of the design patterns we detect is the Empty Catch Block, where the catch block is empty and serves no purpose for error handling. The code for ***EmptyCatch*** is below:

```
1  package ExampleBugPattern;
2  import spoon.support.Level;
6
7  public class EmptyCatch extends AbstractProcessor<CtCatch> {
8      private final Logger LOGGER = Logger.getLogger(this.getClass().getSimpleName());
9
10     public void process(CtCatch element) {
11         if (element.getBody().getStatements().isEmpty()) {
12             LOGGER.warning("empty catch clause at " + element.getPosition().toString()+ ": \n" +
13                     element
14                     );
15
16         }
17     }
18 }
19
```

- When implementing a new detection rule, you create a new class, such as *EmptyCatch*, and extend *AbstractProcessor*. This is a base class for creating processors that operate on specific elements of the Java Abstract Syntax Tree (AST). In other words, this class handles the complex task of traversing the AST. It only requires three things in return:
  - You need to define the type of processor. For example, *AbstractProcessor<CtCatch>* indicates that it specifically processes CtCatch elements, which are catch blocks in the Java code.
  - Extending *AbstractProcessor* will require you to implement one method, which is *process*.
    - *Process:* This method traverses all nodes in the Abstract Syntax Tree (AST), allowing for detailed analysis of the code structure. In this instance, we focus on identifying CtCatch nodes with empty catch blocks. Specifically, for each CtCatch node encountered, we examine its body. If the body is empty, it indicates an empty catch block, and we log a warning message accordingly.

- When you execute the code, you will see the following output

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Sep 13, 2024 7:04:58 PM BugPatternToImplement.divisonByZero process
WARNING: Division by Zero  (/Users/djaek/Documents/SE333/analysis/src/resource/HomeworkFileToBeAnalyzed.java:41):
x / 0
```

- This shows that we found one division by zero bug pattern at line 41.

# Homework Goal.

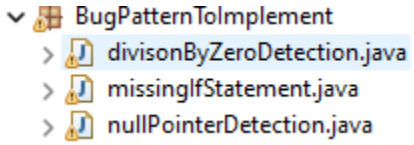
The goal of your homework is to detect three additional bug patterns:

- Null Pointer Exception Detection
- Division By Zero Detection
- Missing If Statement Body

**Step 1:** For example, when you open the ***HomeworkFileToBeAnalyzed*** you will see these two methods. The comment shows where the faults are. Your job is to detect these faults.

```
44     void method5() {
45         String text;
46         int length = text.length(); // Fault: NullPointerExceptior
47     }
48
49     void method6() {
50         try {
51             int[] numbers = new int[5];
52             int result = 10 / 0; // Fault: Division by zero
53         } catch (ArithmeticException e) {
54             System.out.println("An exception occurred");
55         } catch (ArrayIndexOutOfBoundsException e) {
56             System.out.println("An exception occurred");
57         }
58     }
```

**Step 2:** You should implement your detection in the code I already provided in the folder shown:



**Step 3:** When you open the file ***divisionByZeroDetection*** you will see below:

```
1  package BugPatternToImplement;
2  import java.util.ArrayList;
11
12 public class divisonByZeroDetection extends AbstractProcessor<CtElement> {
13     private final Logger LOGGER = Logger.getLogger(this.getClass().getSimpleName());
14
15     @Override
16     public void process(CtElement element) {
17         // TO IMPLEMENT!
18         LOGGER.warning("Division by Zero  " + element.getPosition().toString());
19     }
20 }
21
```

You need to implement the code in line 17, which will enable you to detect code that has division by zero.

Note that the current *AbstractProcessor<CtElement>* has a type of CtElement, meaning it will traverse through every node. You may want to specify a more specific node type for this. For example,

- To detect **division by zero**, you would likely focus on nodes representing expressions, especially binary operations (*CtBinaryOperator*). You can refine the processor to target those specifically.
- For detecting **empty catch blocks**, you can target *CtCatch* nodes.
- For detecting **empty methods**, you can target *CtMethod*.
- For **null pointer exception**, you would target *CtClass* nodes.
- For **if statements that always evaluate to true**, you would target *CtBlock* nodes.

You may need to review the Spoon documentation to determine which specific node types to focus on for detecting the bug patterns in *HomeworkFileToBeAnalyzed*.

I will list some of the AST node types in here for reference.

- **CtClass<T>**: Represents a class declaration. Useful for analyzing class definitions and their components.
- **CtMethod<T>**: Represents a method declaration. Useful for analyzing method definitions, including their bodies and signatures.
- **CtField<T>**: Represents a field declaration within a class. Useful for analyzing field definitions and their usage.
- **CtIf**: Represents an if statement. Useful for detecting conditional logic and analyzing conditions.
- **CtTry**: Represents a try block used in exception handling. Useful for analyzing exception handling and detecting empty catch blocks.
- **CtAssignment<T, A>**: Represents an assignment operation. Useful for detecting assignments, including potentially problematic ones like divisions by zero.
- **CtBinaryOperator<T>**: Represents binary operations (e.g., +, -, /). Useful for detecting issues with operations, including division by zero.
- **CtExpression<T>**: Represents an expression in the code. This can include a wide range of expressions, useful for general analysis of code logic.
- **CtLoop**: Represents loop constructs (e.g., for, while). Useful for analyzing loops, including detecting empty or unnecessary loops.
- **CtBlock<R>**: Represents a block of code, typically enclosed in braces {}. Useful for analyzing the scope and content of methods or control structures.
- **CtCatch**: Represents a catch block in exception handling. Useful for detecting empty catch blocks.
- **CtVariable<T>**: Represents a variable declaration. Useful for analyzing variable usage and initialization.
- **CtInvocation<T>**: Represents method or constructor invocations. Useful for analyzing method calls and their arguments.
- **CtAnnotation<A>**: Represents annotations in the code. Useful for detecting and analyzing annotations.
- **CtComment**: Represents comments in the code. While comments themselves are not typically analyzed for faults, they can provide context for the code.
- **CtLiteral<T>**: Represents literal values (e.g., integers, strings). Useful for analyzing constant values used in the code.
- **CtSwitch<S>**: Represents a switch statement. Useful for analyzing switch constructs and their cases.
- **CtType<T>**: Represents a type in the code. Useful for analyzing type declarations and references.
- **CtThrow**: Represents a throw statement used to throw exceptions. Useful for analyzing exception handling logic.

You can resort this file which list all of the node types.

Note that a Null Pointer Exception can occur many ways for different software systems, but you should not worry about all possible scenarios for this homework. Your task is to detect the faults in the file provided, *HomeworkFileToBeAnalyzed.*

**In summary you will detect the following faults below:**

- **Total Faults**: 3
    - Division by Zero: 2
    - Null Pointer Exception: 1
- **Total Bad Design Patterns**: 5
    - Empty Catch Block: 1
    - Empty If Statement: 2
    - Empty Method: 1
    - If Statement Always True: 1