

# Program Transformations

# Announcements

1. Gradescope is fixed!
  - a. All future assignments submitted on Gradescope
  - b. Including Lab2 today
  
2. Lab1 due last night
  - a. If you emailed it to me, I uploaded it to gradescope
  - b. If you don't see it there, upload directly
  
3. HW1 released... questions?

# Outline

- Java Bytecode review
- Program Transformations
  - Bytecode
  - AST
- Unit Testing and Coverage
  - Today's Lab Topic

# Java Bytecode - Slot reuse

xstore / xload load from or store to the local variable

- Where x is the type

Sometimes a slot is **reused** and the same local variable slot will be used for different purposes during different parts of a method's execution

**Variables with Different Scopes:** Local variables that have different scopes and do not overlap in execution time can be assigned the same slot

# Java Bytecode - Exceptions

- Each method has an **Exception Table**
  - Records regions of code that are enclosed in a try
- Each entry contains:
  - Start and end of the protected region
  - Type of Exception to catch

# Java Bytecode - Exceptions

When an exception occurs:

1. **Exception Object** is created and **pushed** on the stack
2. **Table Lookup:** The JVM searches the exception table to see if any of the entries match the type of the exception and the program counter (PC) at the time of the exception
3. If there is an entry in the exception table, **goto target**
4. else: throw the exception and halt execution

# Java Bytecode - Exceptions

```
public void div(int x) {  
    try {  
        int result = 10 / x;  
    } catch (ArithmeticException e) {  
        System.out.println(e);  
    }  
}
```

**public void div(int);**

Code:

```
0: bipush    10  
2: iload_1  
3: idiv  
4: istore_2  
5: goto      16  
8: astore_2  
9: getstatic #3    // Field java/lang/System.out:Ljava/io/PrintStream;  
12: aload_2  
13: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/Object;)V  
16: return
```

Exception Table

from	to	target	type
0	5	8	java/lang/ArithmeticException

Local Variable Table

this	x	e
------	---	---

Operand Stack


# Program Representations anecdote

Automated grading of homework assignments - consider how we would enforce the use of the following program constructs with different program representations.

1. Write a ***loop*** to print 0 to 100
2. Write a ***recursive method*** to compute factorial

**Enforcing certain constructs is difficult at a textual level**



# Program Transformations at a textual level

Example transformation: Turn all assert statements into junit assert

```
public class Div {  
    public int div(int x, int y) {  
        assert y != 0;  
        return x / y;  
    }  
}
```

# Program Transformations

# Why might we want to transform a program?

- Optimizations
- Obfuscations
- Debugging
- Performance Monitoring
- ...
- We'll learn more in this course!

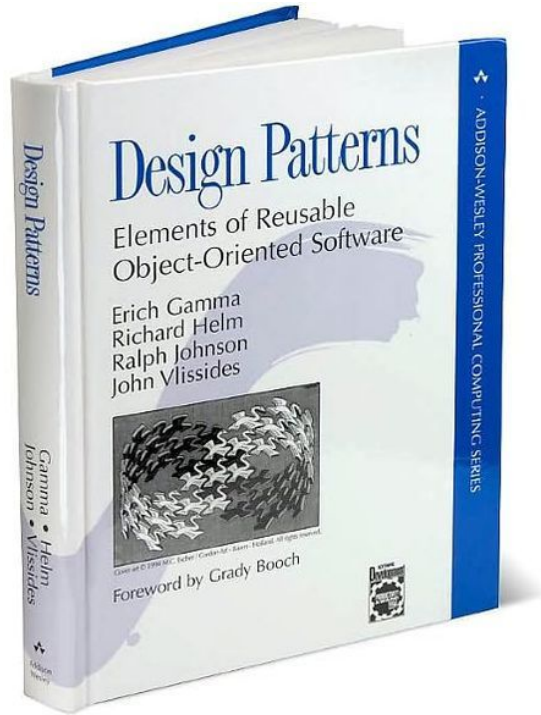
# AST Transformations

# AST Transformations

**Example:** inject performance logging for each method (print time taken for each method)

**Traversing the AST tends to be laborious and error prone. This is largely due to the process relying on recursion and frequent type checking to attain your goal**

# Design Patterns



- Reusable solutions to common software design problems
- Provide a way to structure your code in a way that is flexible, maintainable, and efficient
- *“All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways that I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects.”*

# Visitor Design Pattern

- **Visitor Design Pattern** commonly used for tree transformations
- Allows you to separate the operations / transformations from the traversal process
- Trees often have different types of nodes. The Visitor pattern makes it easy to define different operations for each node type by using method overloading in the visitor.
- Each operation is encapsulated in a "visitor," which visits different node types (like addition or multiplication) and processes them.

## JavaParser Library

- Allows you to interact with Java source code as an AST
- Provides mechanism to navigate the tree with *Visitor Support*
  - Allows you to write operations without needing to write traversal code



# A Simple Visitor

Let's create a visitor to print the name of each method in a program

You can also pass a value to the visitor!

- Useful to save or load state

# A Simple Modifying Visitor

Let's write a visitor to change variable names (declarations and references) to all caps in Method `decls`

```
public class Vars {  
  
    public void decls() {  
        int foo = 15;  
        int helloworld = 20;  
        int added = foo + helloworld;  
    }  
  
    public void dontchange() {  
        int lower = 15;  
        int stillLower = 10;  
        int lowercase = lower + stillLower;  
    }  
}
```

# JavaParser library

## Basic Workflow:

1. Parse the source into an AST
  - a. `JavaParser.parse(sourceCode);`
2. Transform the AST
  - a. Using a Visitor, traverse the AST and perform a transformation
3. Write the modified AST
  - a. Call `toString()` on the AST root (`CompilationUnit`) to get the source

# Bytecode Transformations

# ASM

Java Library for Bytecode manipulation

Basic Workflow:

1. Read the bytecode
  - a. `ClassReader`
2. Transform the bytecode
3. Write the modified bytecode
  - a. `ClassWriter`



# Bytecode Transformation

Increment every constant by 1 in the main method

```
public class Constants {  
  
    public static void dontModify(int q) {  
        int x = 100;  
        int y = 10;  
        int z = x + y + 1;  
    }  
  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
        int c = 30;  
  
        dontModify(100);  
    }  
}
```

```
public static void dontModify(int);
```

Code:

```
0: bipush      100  
2: istore_1  
3: bipush      10  
5: istore_2  
6: iload_1  
7: iload_2  
8: iadd  
9: iconst_1  
10: iadd  
11: istore_3  
12: return
```

```
public static void main(java.lang.String[]);
```

Code:

```
0: bipush      10  
2: istore_1  
3: bipush      20  
5: istore_2  
6: bipush      30  
8: istore_3  
9: bipush      100  
11: invokestatic #7    // Method dontModify:(I)V  
14: return
```

# Bytecode vs AST transformations

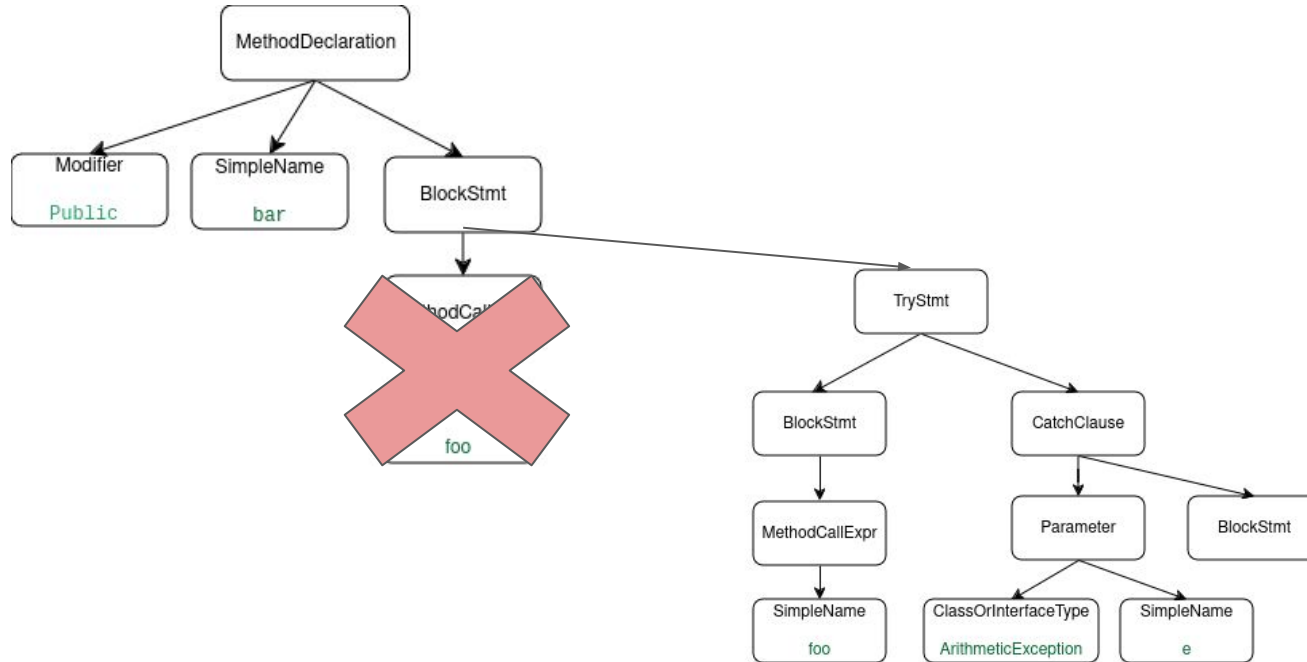
Transformation: Wrap all calls to a method `foo` in a try-catch block as precisely as possible.

Consider performing this transformation at both the **bytecode** and **AST** level:

- Come up with an example source and draw its representation
- Work through it by hand
- Consider error cases
- Which transformation do you prefer?

# AST try-catch Transformation

```
public void bar(int x) {  
    foo();  
}
```





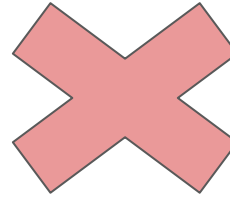
# AST try-catch Transformation

## Failure cases:

```
public void bar(int x) {  
    int z = foo();  
    z = z + 1;  
}
```



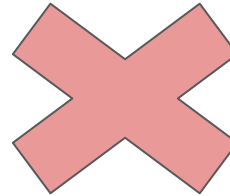
```
public void bar(int x) {  
    try {  
        int z = foo();  
    } catch (Exception e) {  
  
    }  
    z = z + 1;  
}
```



```
public void bar(int x) {  
    if (foo()) {  
        ...  
    }  
}
```



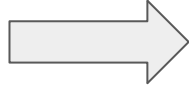
```
public void bar(int x) {  
    try {  
        if (foo()) {  
            ...  
        }  
    }  
}
```



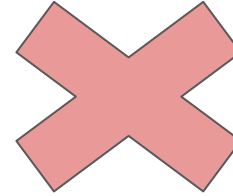
# AST try-catch Transformation

## Failure cases:

```
public void bar(int x) {  
    for (int i=0; i<foo(); i++)  
        ..  
}
```



```
public void bar(int x) {  
    try {  
        for (int i=0; i<foo(); i++)    ..  
    } catch (Exception e) {  
    }  
}
```



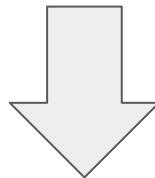
```
public void bar(int x) {  
    baz(foo());  
}
```

...

# Bytecode try-catch Transformation

1. Add statements after invokevirtual:
  - a. goto x
  - b. astore\_x //store exception in LVT
2. Add entry to exception table
3. modify labels on following statements

```
public void bar();  
Code:  
0: aload_0  
1: invokevirtual #2           // Method foo():V  
4: return
```



```
public void bar();  
Code:  
0: aload_0  
1: invokevirtual #2           // Method foo():V  
4: goto          8  
7: astore_2  
8: return
```

# Bytecode try-catch Transformation

## Failure cases of transformation on an AST:

- Output of `foo` saved in a variable
- `foo` is the condition in an if-statement
- `foo` is the stopping condition in a for-loop

Would these be problematic in bytecode transformations?

# Unit Testing

# Testing

Testing usually involves: **executing** a unit of code, observing its **resulting state**, **asserting** some property on it

## Why do we write tests?

1. To find faults
2. To provide documentation
3. To prevent regressions

# What makes a good test suite?

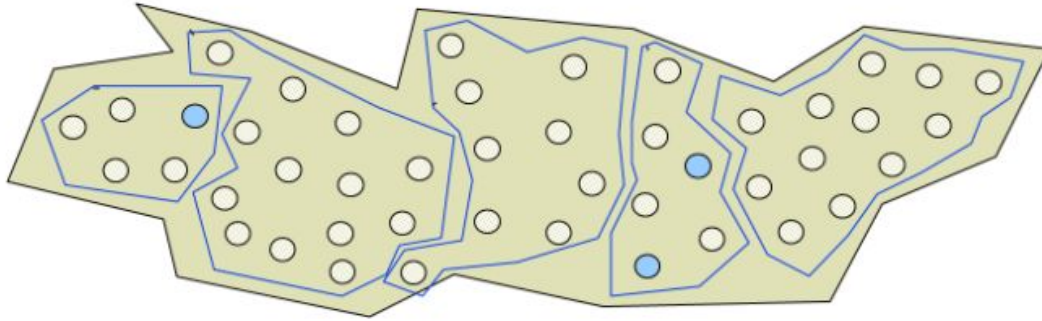
- Cover all “classes” of input cases
  - Happy cases
  - Invalid cases
  - Edge cases



# Equivalence Classes

How can we select inputs?

We can partition the space into “equivalence classes”



# What makes a good test suite?

- Cover all “classes” of input cases
  - Happy cases
  - Invalid cases
  - Edge cases
- Diverse
  - Both black-box and white-box criteria
- Executes Quickly
- Covers all portions of the code

# What makes a bad test suite?

- Only tests certain classes of inputs
  - “Add a loop that test about 10000 different positive number from 0 to 9999.”
- Redundant
- Slow




# Coverage: a metric for test suite quality

**Intuition:** untested parts of the code may still contain faults. If your test suite doesn't execute certain sections of the program, potential mistakes in those areas can go undetected

# Types of Coverage

1. Line Coverage
2. Statement Coverage
3. Branch Coverage
4. Path Coverage
5. Edge Coverage

# Measuring Coverage with JaCoCo

 jacoco >  com.example.jacoco >  Rectangle.java

## Rectangle.java

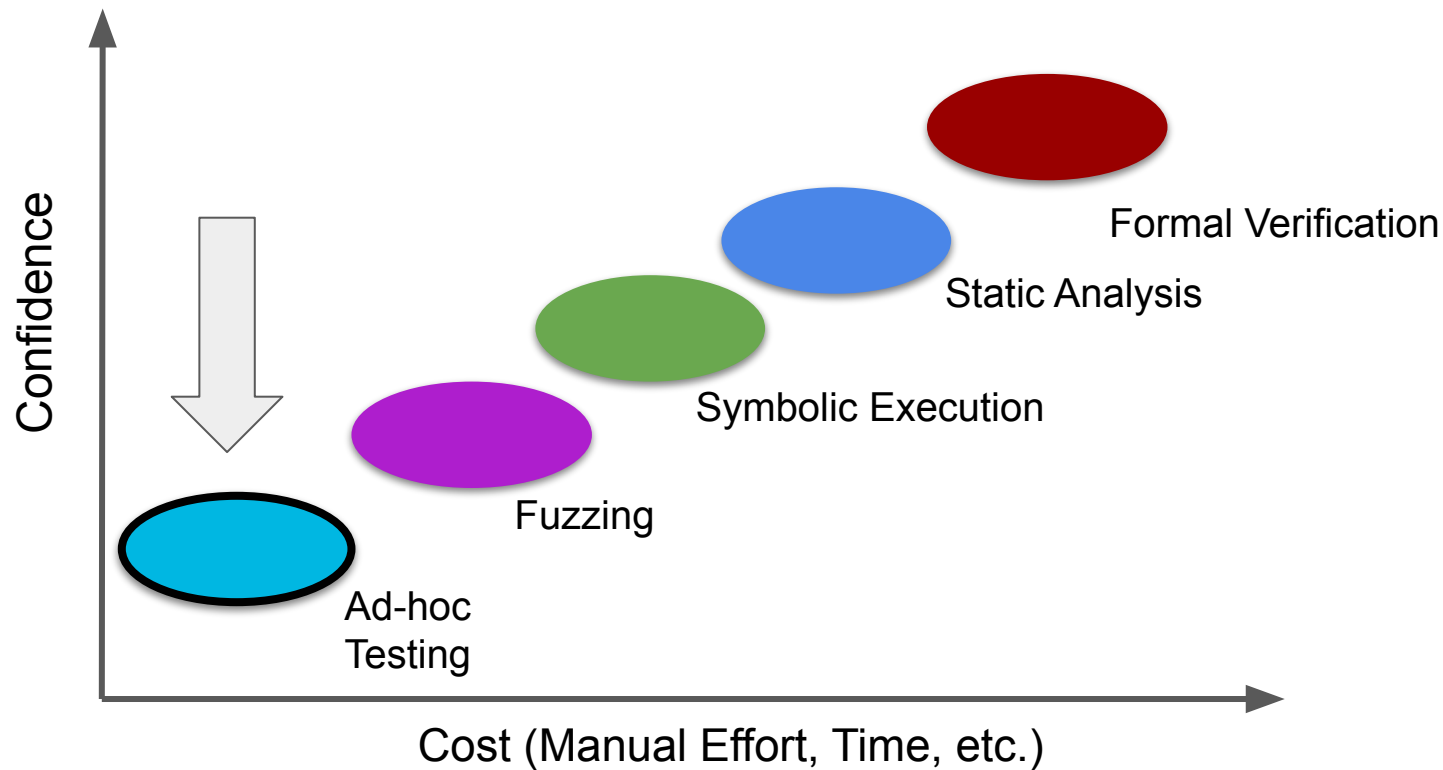
```
1. package com.example.jacoco;
2.
3. public class Rectangle {
4.     private int x;
5.     private int y;
6.     private int width;
7.     private int height;
8.
9.     public Rectangle(int x, int y, int width, int height) {
10.         if (width <= 0 || height <= 0)
11.             throw new IllegalArgumentException("Dimensions are not positive");
12.
13.         this.x = x;
14.         this.y = y;
15.         this.width = width;
16.         this.height = height;
17.     }
18.
19.     public boolean intersects(Rectangle other) {
20.         if (x + width <= other.x)
21.             return false;
22.         if (x >= other.x + other.width)
23.             return false;
24.         return (y + height > other.y && y < other.y + other.height);
25.     }
26. }
```

# How do you think JaCoCo works?

Coverage collection tools work either by:

1. Instrumenting the bytecode
2. Calculating coverage “on the fly”
  - a. Easier with a VM language

# Unit Testing has low confidence, but low cost





# Summary

- Transformations are dangerous over text
  - Use AST or bytecode instead
- Coverage: a metric for test suite quality
- HW1 (due next Wednesday Feb 12)
  - AST and Bytecode transformations
- Lab today: What makes a good test suite?