UNIVERSITÄT BERN

Programming 2

4. Testing and Debugging

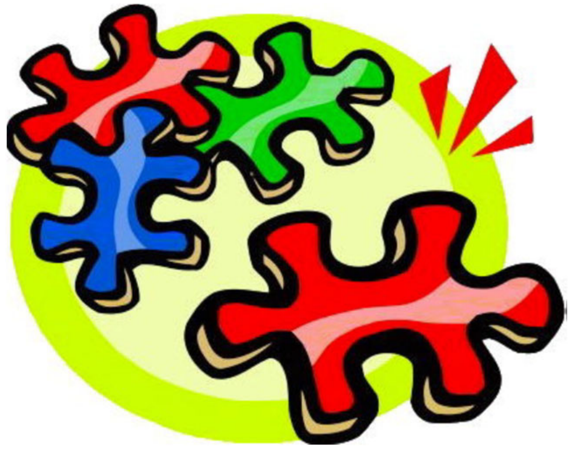
Timo Kehrer

https://seg.inf.unibe.ch/teaching/current/p2/

Recap: Who's to blame?

If preconditions are violated, the client is to blame.

If invariants or postconditions are violated, the provider is to blame!



Recap: Checking Class Invariants

```
// Constructor
public LinkStack() {
    ...
    assert invariant();
}
```

Every class has its own invariant:

```
protected boolean invariant() {
    return (size >= 0) &&
        ((size == 0 && this.top == null) || (size > 0 && this.top != null));
}
```

✎ When should you check invariants?

✓ At the end of each constructor (when non-trivial) and as part of post-conditions (we will see this in a minute)

Recap: Checking Pre-conditions

Assert pre-conditions to inform clients when *they* violate the contract.

This is all you have to do!

```
public E top() {
    assert !this.isEmpty(); // pre-condition
    return top.item;
}
```

✎ When should you check pre-conditions to methods?

✓ *Always check pre-conditions, raising exceptions if they fail.*

Recap: Checking Post-conditions

Assert post-conditions and invariants to inform yourself when *you* violate the contract.

```
public void push(E item) {
        top = new Cell(item, top);
        size++;
        assert !this.isEmpty(); // post-condition
        assert this.top() == item; // post-condition
        assert invariant();
}
```

This is all you have to do!

✎ When should you check post-conditions?

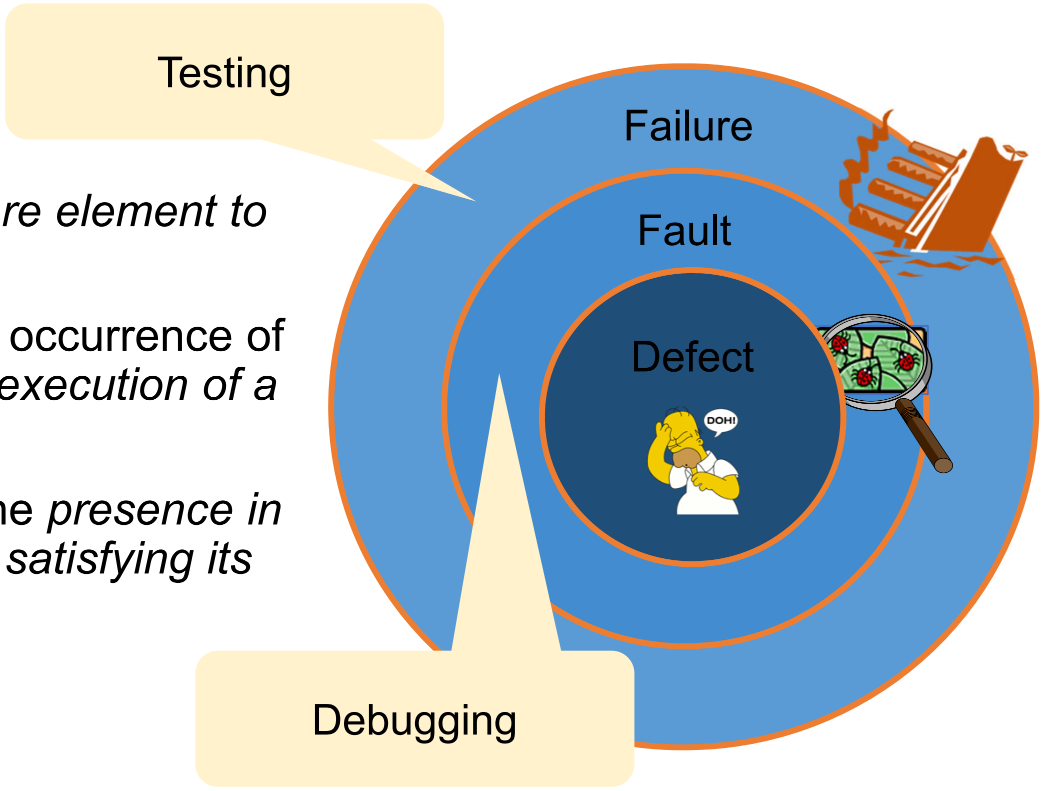
✓ *Check them whenever the implementation is non-trivial.*

Recap: What can go wrong?

A failure is the *inability of a software element to satisfy its purpose*.

A fault (aka. error, infection) is the occurrence of an *abnormal condition during the execution of a software element.*

A defect (aka. “bug”, mistake) is the *presence in the software of some element not satisfying its specification.*



Outline

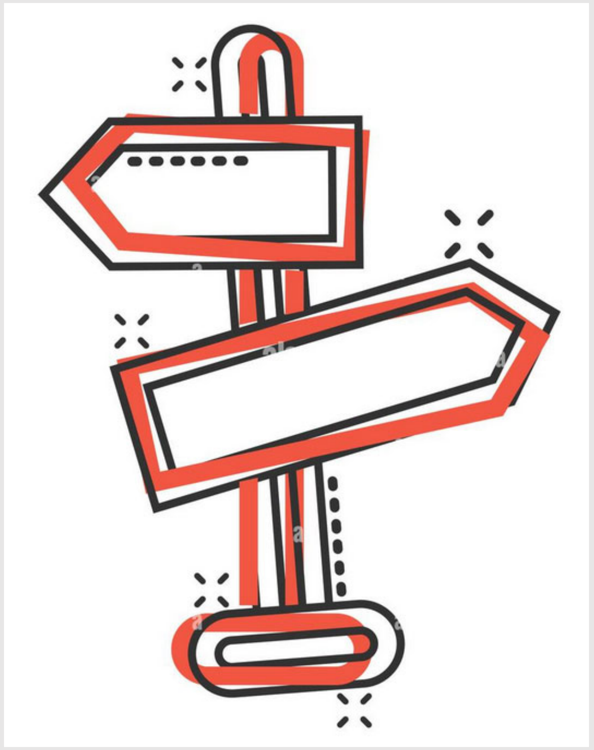
UNIVERSITÄT BERN

Motivation and Background

A Testing Framework

Testing Strategies
- Black-box Testing
- White-box Testing

Debugging



Outline

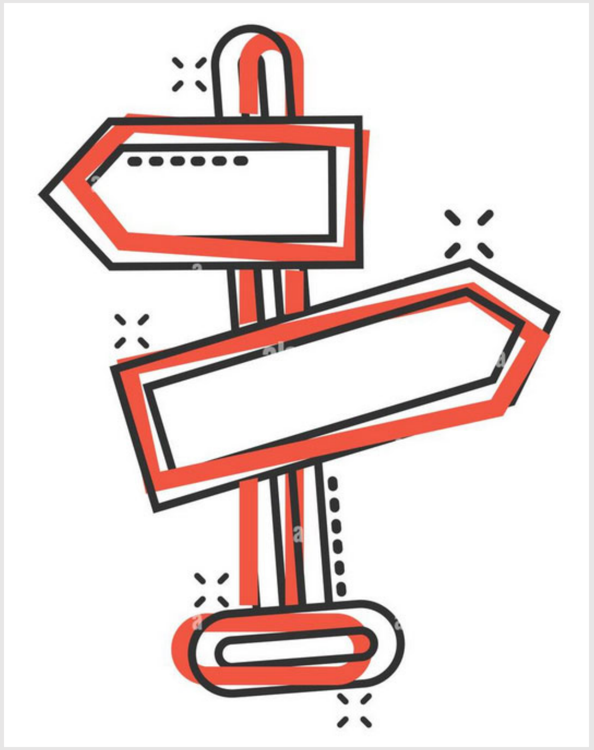
UNIVERSITÄT BERN

Motivation and Background

A Testing Framework

Testing Strategies
- Black-box Testing
- White-box Testing

Debugging



The Problem

"Testing is not closely integrated with development. This prevents you from measuring the progress of development — you can't tell when something starts working or when something stops working."

— "Test Infected", Beck & Gamma, 1998

Interactive testing is tedious and seldom exhaustive.

Automated tests are better, but,

- how to integrate testing into the development process?
- how to organize suites of tests?

"Developers should spend 25-50% of their time developing tests."

Test Levels

Unit testing:	test *individual (stand-alone) components*
Integration testing:	test *interactions between components*
System testing:	test that the complete system fulfils *functional and non-functional requirements*
Acceptance testing (alpha/beta testing):	test system with *real rather than simulated data*

Testing is always iterative!

Testing Practices

During Development

- When you need to add new functionality, *write the tests first.*
  - You will be done when the test runs.
- When you need to redesign your software, refactor in small steps, and *run the (regression) tests after each step*.
  - Fix what's broken before proceeding.

During Debugging

- When someone discovers a failure in your program, *first write a test* that demonstrates the failure.
  - Then debug until the test succeeds.

Test-Driven Development

Regression Testing

TRAFFIC Principle

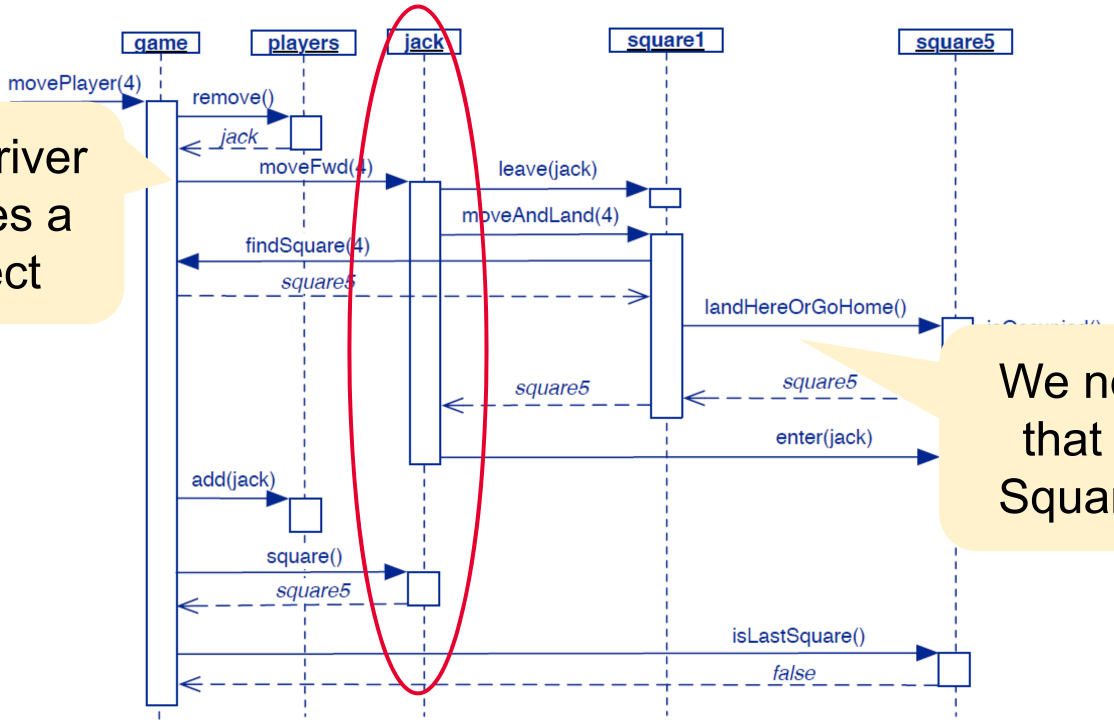
Testing Style

- Write unit tests that *thoroughly test a single class*
- Write tests *as you develop* (even before you implement)
- Write tests for *every new piece of functionality*
- Write automated tests that you can (re-)run any time, preferably using a testing framework.

"The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run."

Test Drivers and Stubs

What if we want to test a single Person object that interacts with other objects?

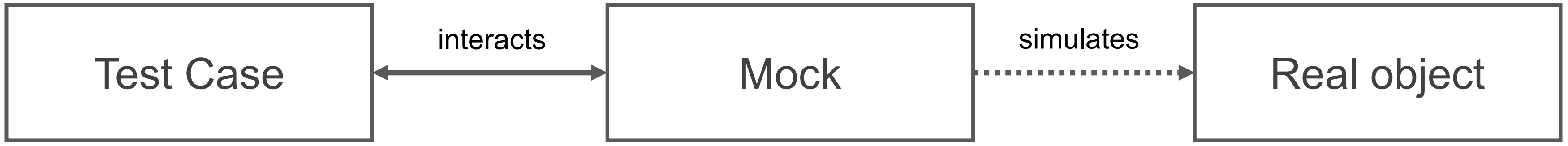


We need a driver that simulates a Game object

We need stubs that simulate Square objects

Outlook: Mock Objects

A *mock object* simulates the behaviour of a real object in a controlled way to support automated testing.



With mock objects, we are able to implement scripted scenarios in which single objects are tested in isolation.

Outline

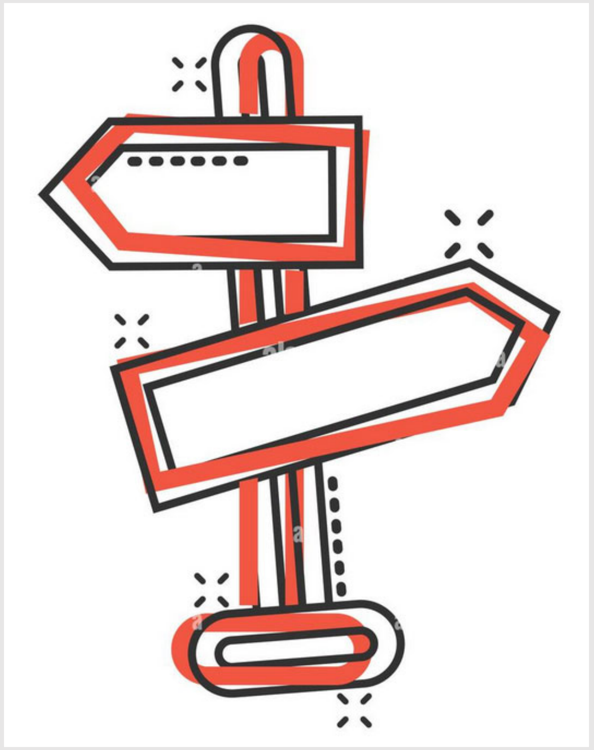
UNIVERSITÄT BERN

Motivation and Background

A Testing Framework

Testing Strategies
- Black-box Testing
- White-box Testing

Debugging



JUnit – A Testing Framework

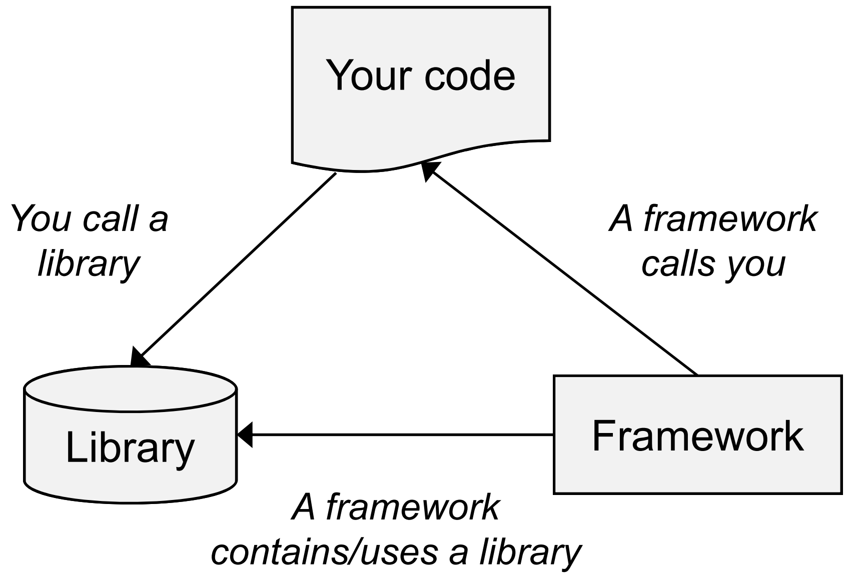
- JUnit is a simple framework to write repeatable tests.
- It is an instance of the xUnit architecture for unit testing frameworks written by Kent Beck and Erich Gamma.

JUnit supports/provides:

- writing *Test Cases and Test Suites*
- methods for *setting up and cleaning up test data* (“fixtures”)
- methods for *making assertions*
- textual and graphical tools for *running tests*

Excursus: Frameworks vs. Libraries

In traditional application architectures, user code makes use of library functionality in the form of procedures or classes.



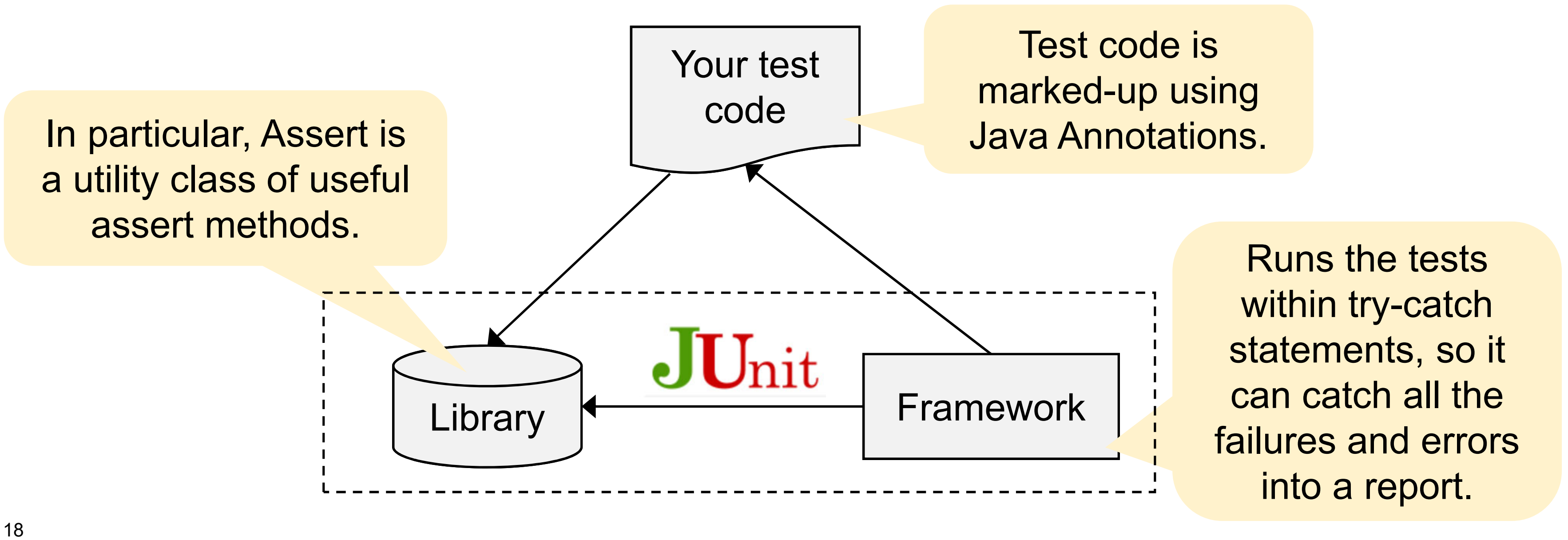
- A framework *reverses* the usual relationship between generic and application code.
- Frameworks provide both generic functionality and application architecture.

Essentially, a framework says: "Don't call me — I'll call you."

Junit 5

Provides all the infrastructure needed to organize and run tests.

The “missing details” to be filled in consist of the concrete tests that you must provide.

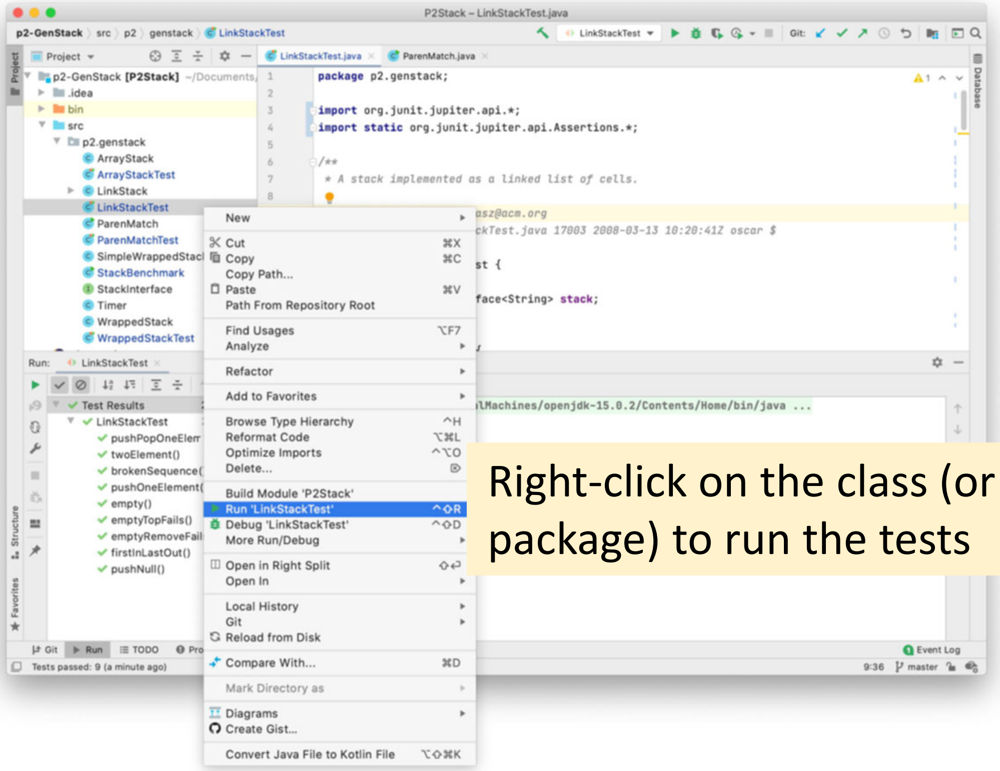

Your test code
Test code is marked-up using Java Annotations.
In particular, Assert is a utility class of useful assert methods.
JUnit
Library
Framework
Runs the tests within try-catch statements, so it can catch all the failures and errors into a report.

Failures and Errors

JUnit distinguishes between *failures and errors:*

- A failure is *a failed assertion*, i.e., an anticipated problem that you test.
- An error is *a condition you didn't check for*, i.e., a runtime error.

Running Tests from IntelliJ



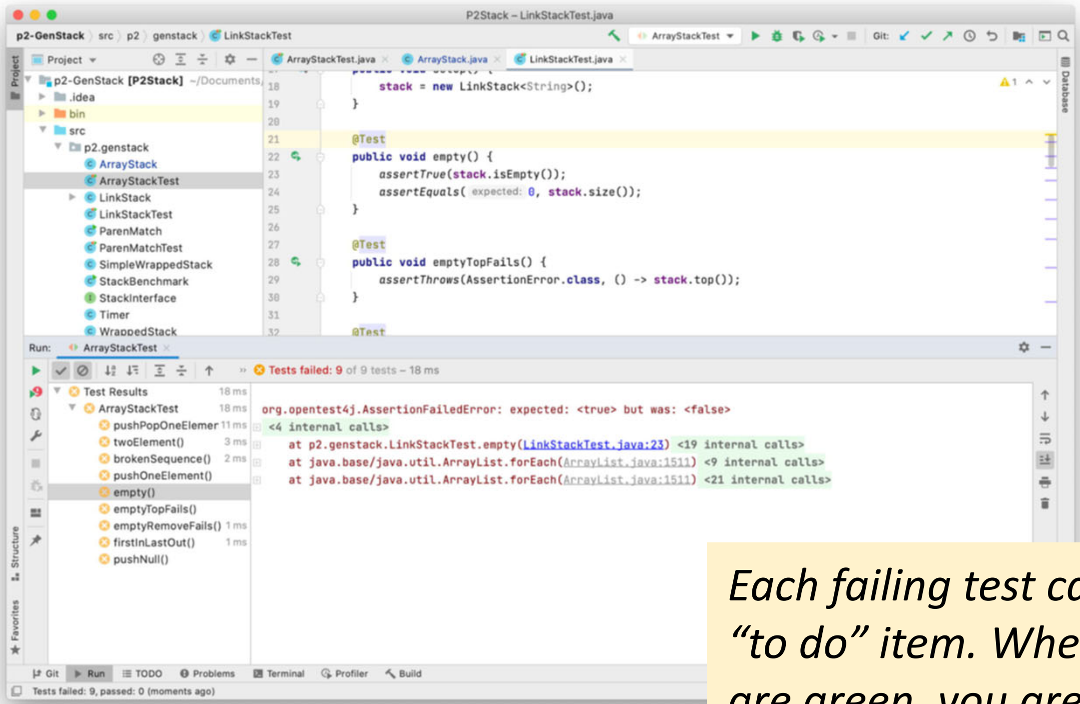
```
package p2.genstack;

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

/**
 * A stack implemented as a linked list of cells.

asz@acm.org
ckTest.java 17003 2008-03-13 10:20:41Z oscar $

st {

face<String> stack;
```

```
lMachines/openjdk-15.0.2/Contents/Home/bin/java ...
```

Tests passed: 9 (a minute ago)

Right-click on the class (or package) to run the tests

Failing Tests as „To Do“ Items



```
    stack = new LinkStack<String>();
}

@Test
public void empty() {
    assertTrue(stack.isEmpty());
    assertEquals( expected: 0, stack.size());
}

@Test
public void emptyTopFails() {
    assertThrows(AssertionError.class, () -> stack.top());
}

@Test
```

```
Tests failed: 9 of 9 tests – 18 ms

Test Results                18 ms
  ArrayStackTest            18 ms
    pushPopOneElemer        11 ms
    twoElement()             3 ms
    brokenSequence()         2 ms
    pushOneElement()
    empty()
    emptyTopFails()
    emptyRemoveFails()       1 ms
    firstInLastOut()         1 ms
    pushNull()
```

```
org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
<4 internal calls>
    at p2.genstack.LinkStackTest.empty(LinkStackTest.java:23) <19 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <21 internal calls>
```

```
Tests failed: 9, passed: 0 (moments ago)
```

Each failing test can be seen as a “to do” item. When all the tests are green, you are done.

Outline

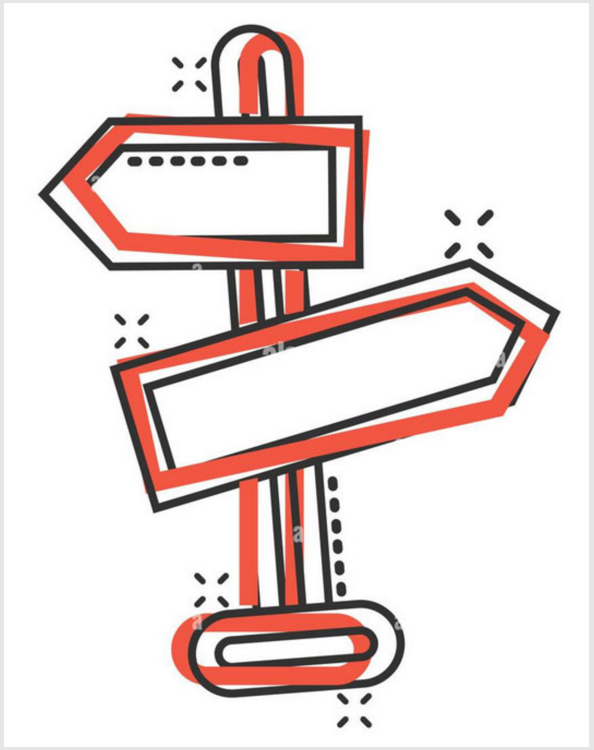
Motivation and Background

A Testing Framework

Testing Strategies

– Black-box Testing
– White-box Testing

Debugging



Testing and Correctness

“Program testing can be used to show the presence of bugs, but never to show their absence!”

—Edsger Dijkstra, 1970

Testing Strategies

Tests should cover "all" functionality

Testing an interface

Black-box Testing
- ❑ Test every public method
- ❑ Test boundary conditions
- ❑ Test key scenarios
- ❑ Test exceptional scenarios

Testing an algorithm

White-box Testing
- ❑ Every line of code (i.e., statements)
- ❑ Every branching behavior induced by conditional statements

Outlook:
- ❑ Every path through the code
- ❑ Every complex condition
- ❑ …

Outline

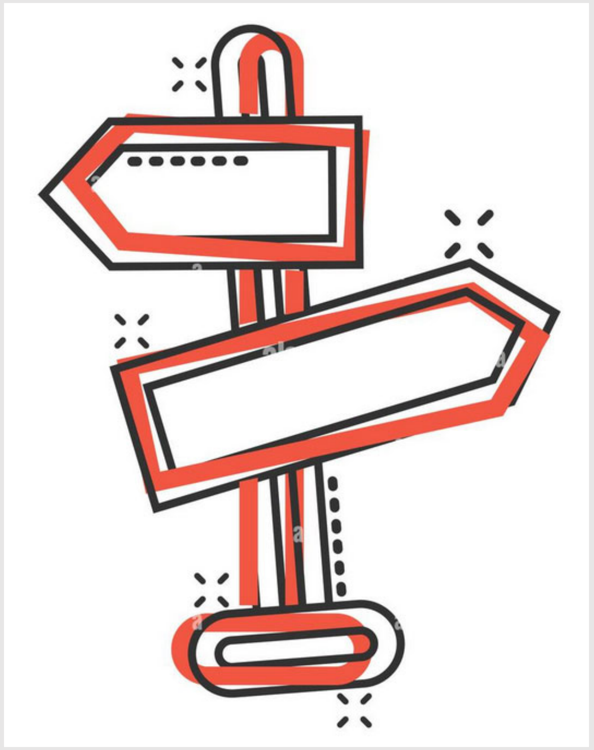
UNIVERSITÄT BERN

Motivation and Background

A Testing Framework

Testing Strategies
- **Black-box Testing**
- White-box Testing

Debugging



Testing our Stack Interface

Recall our stack interface from the last lecture

```
public interface IStack<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```

We will develop some tests to exercise all the public methods.

Testing Public Methods

`isEmpty()`	True when it’s empty; false otherwise (needs a push)
`size()`	Zero when empty; non-zero otherwise (needs a push)
`push()`	Possible any time; affects size and top
`top()`	Only valid if not empty; needs a push; returns the last element pushed
`pop()`	Only valid if not empty; needs a push first; affects size and top

A LinkStackTest Class

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class LinkStackTest {
    protected IStack<String> stack;

    @BeforeEach
    public void setUp() {
        stack = new LinkStack<String>();
    }
…
}
```

Start by setting up the test fixture

Testing the Empty Stack

We can test both `isEmpty()`and `size()`with an initial stack

```
@Test
public void empty() {
    assertTrue(stack.isEmpty());
    assertEquals(0, stack.size());
}
```

Alternatively, we could write two separate tests, one for each condition

Testing a non-empty Stack: Push

We modify the stack and test the new state:

```
@Test
public void pushOneElement() {
    stack.push("a");
    assertFalse(stack.isEmpty());
    assertEquals(1, stack.size());
    assertEquals("a", stack.top());
}
```

Testing a non-empty Stack: Push and Pop

We push and pop and test if the stack is empty.

```
@Test
public void pushPopOneElement() {
    stack.push("a");
    stack.pop();
    assertTrue(stack.isEmpty());
    assertEquals(0, stack.size());
}
```

At this point we have minimally tested the entire stack interface

Testing Boundary Conditions

The only boundary value in the stack interface could be if `null` is pushed:

```
@Test
public void pushNull() {
    stack.push(null);
    assertFalse(stack.isEmpty());
    assertEquals(1, stack.size());
    assertEquals(null, stack.top());
}
```

Testing for Failure

A special kind of boundary condition is checking whether the class behaves as expected when the preconditions for a method do not hold.

```
@Test
public void emptyTopFails() {
    assertThrows(AssertionError.class, () -> stack.top());
}

@Test
public void emptyRemoveFails() {
    assertThrows(AssertionError.class, () -> stack.pop());
}
```

Testing a Key Scenario

We should also test a more complex scenario that exercises the interaction between methods:

```
@Test
public void firstInLastOut() {
    stack.push("a");
    stack.push("b");
    stack.push("c");
    assertEquals("c", stack.top());
    stack.pop();
    assertEquals("b", stack.top());
    stack.pop();
    assertEquals("a", stack.top());
    stack.pop();
    assertTrue(stack.isEmpty());
}
```

Testing an Exceptional Scenario

```
@Test
public void brokenSequence() {
     stack.push("a");
     stack.pop();
     assertThrows(AssertionError.class, () -> stack.pop());
}
```

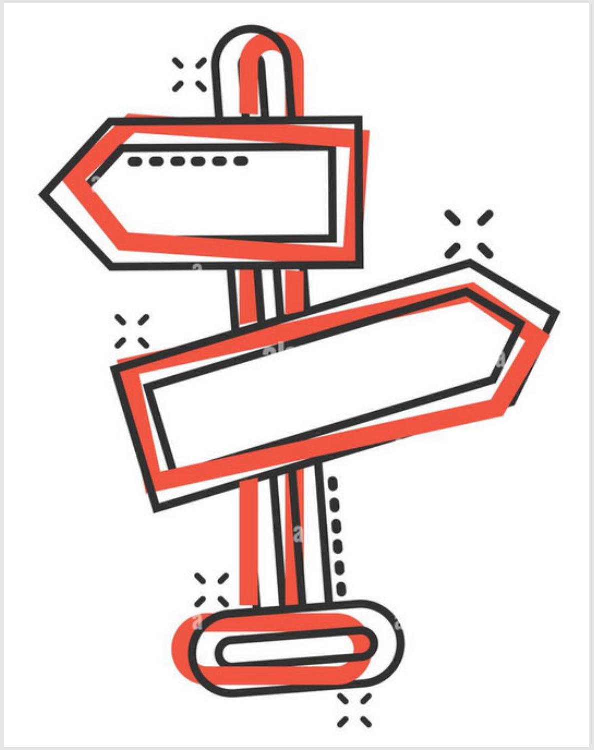
Outline

Motivation and Background

A Testing Framework

Testing Strategies
- Black-box Testing
- **White-box Testing**

Debugging

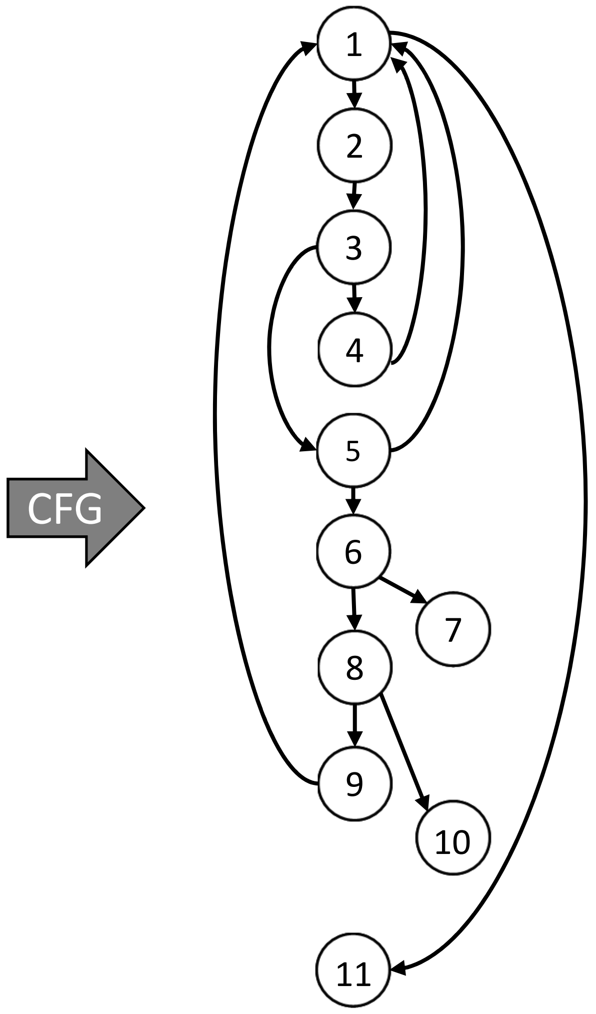


Testing the parenMatch Algorithm

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

White-box Testing: Idea

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                return false;  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

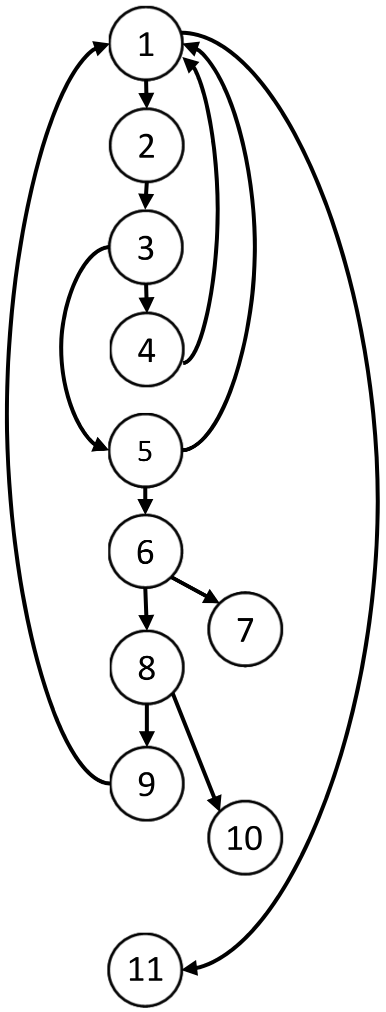


Line/Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

Requires every statement (node in the CFG) to be executed by at least one test

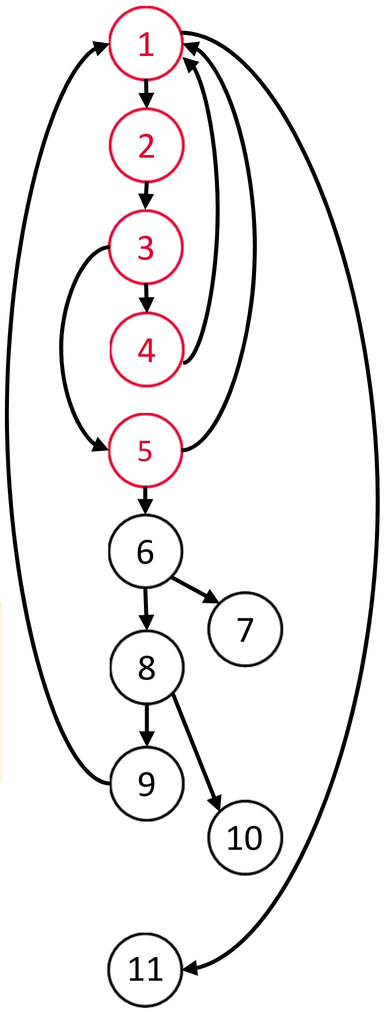
CFG


$$\begin{tikzcd}
1 \arrow[d] \arrow[ddddddd, bend left=60] & \\
2 \arrow[d] & \\
3 \arrow[d] \arrow[dd, bend right] & \\
4 \arrow[uuu, bend right] & \\
5 \arrow[d] \arrow[uuuu, bend right] & \\
6 \arrow[d] \arrow[r] & 7 \\
8 \arrow[d] \arrow[r] & 10 \\
9 \arrow[uuuuuuu, bend left=60] & \\
11 &
\end{tikzcd}$$

Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

Test 1: "()"

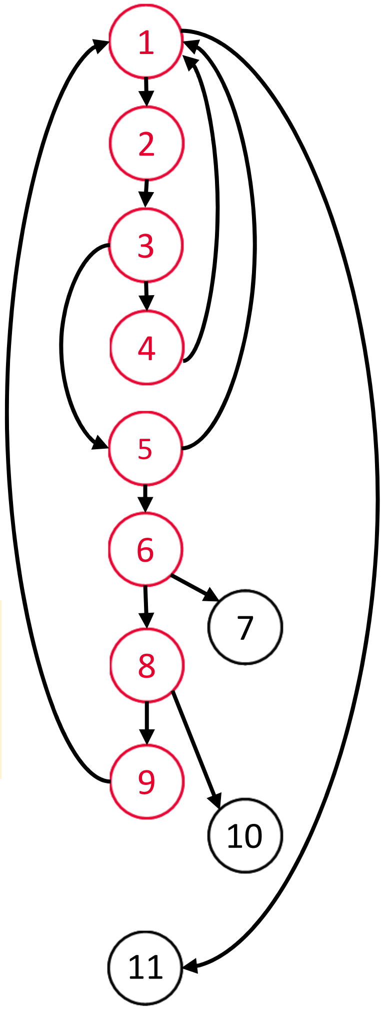


Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"

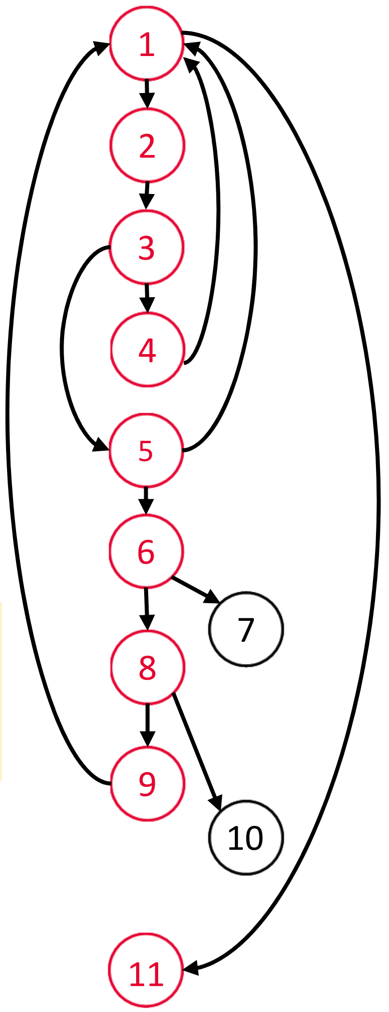


Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"

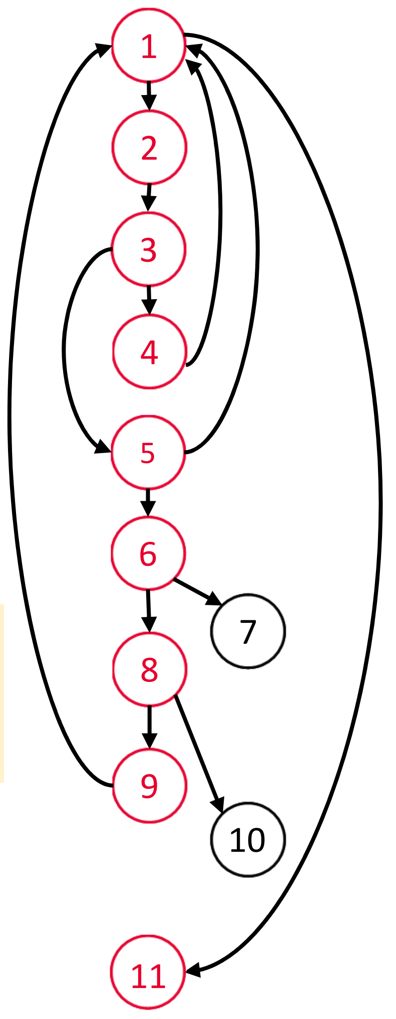


Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"
Test 2: ")"

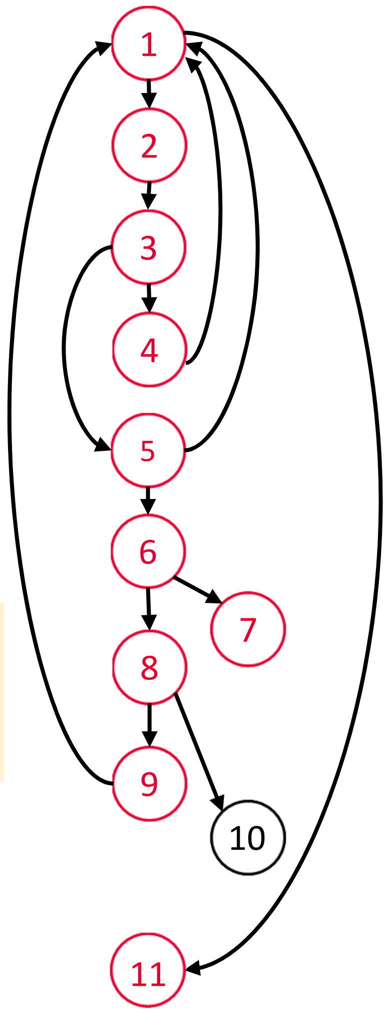


Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"
Test 2: ")"

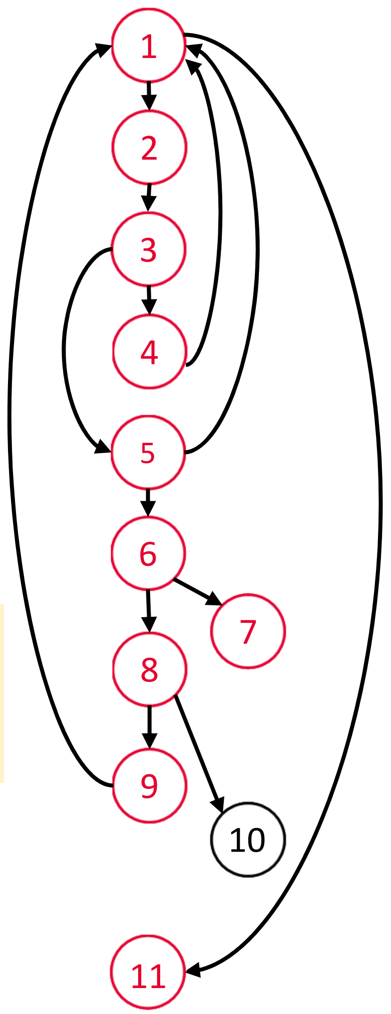


Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"
Test 2: ")"
Test 3: "(]"

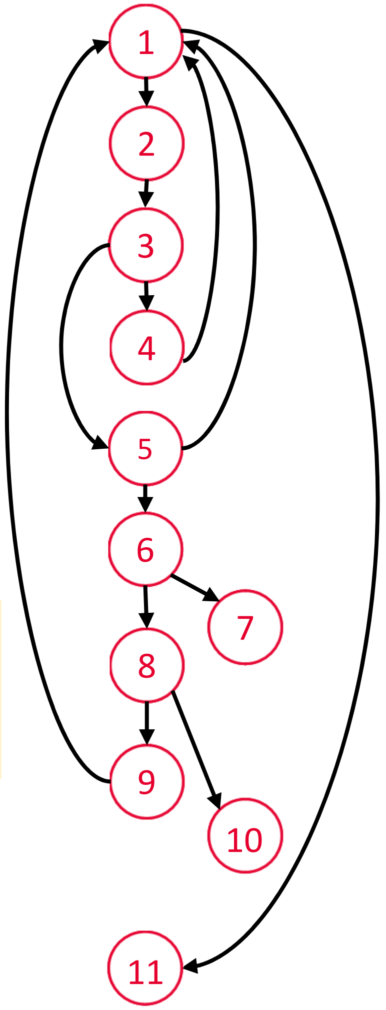


Example: Statement Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"
Test 2: ")"
Test 3: "(]"

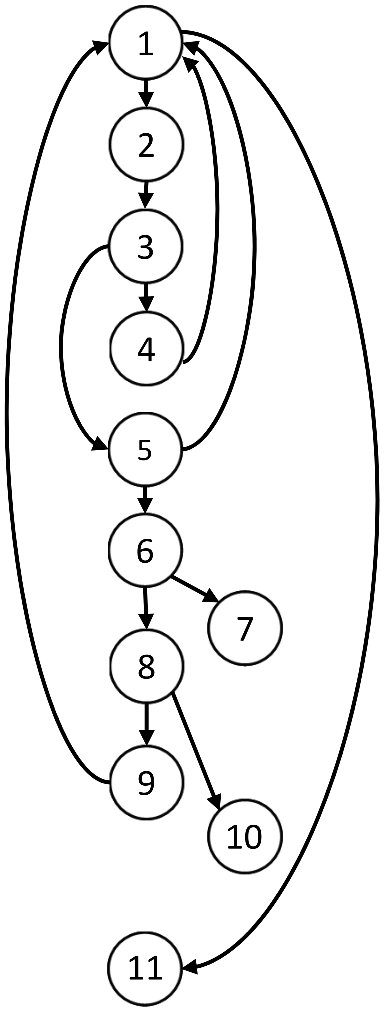


Branch Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

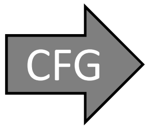
Requires every branch (edge in the CFG) to be executed by at least one test

CFG

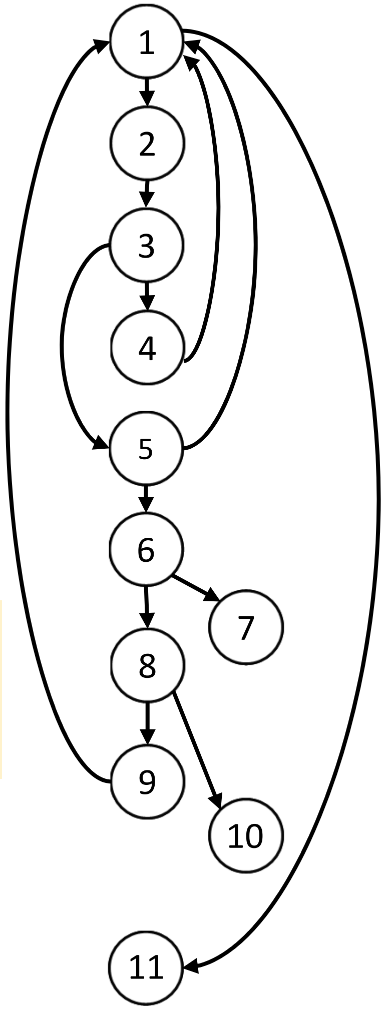


Example: Branch Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
             stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```



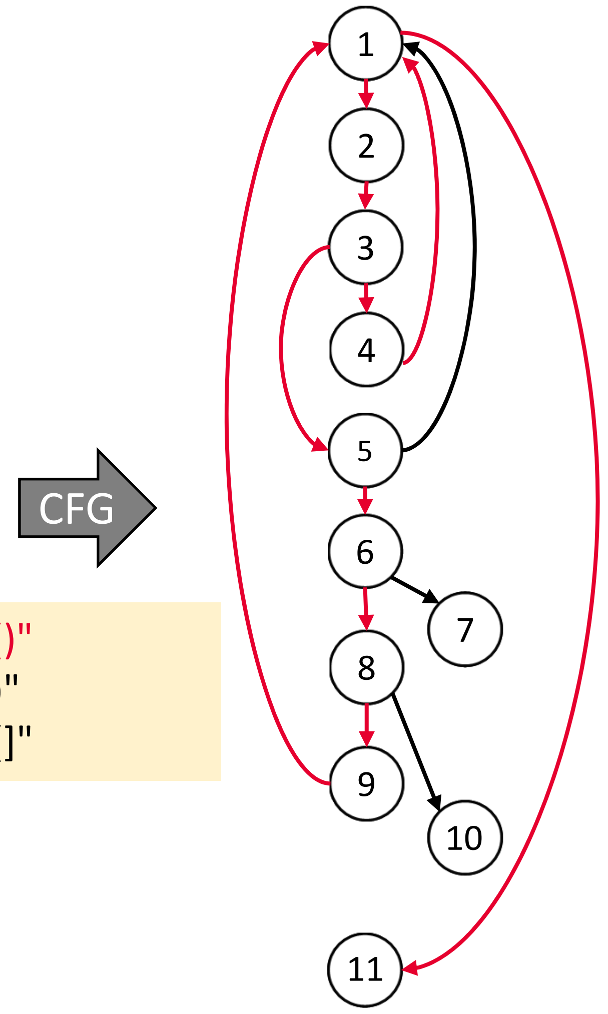
Test 1: "()"
Test 2: ")"
Test 3: "(]"



Example: Branch Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                return false;  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

Test 1: "()"
Test 2: ")"
Test 3: "(]"

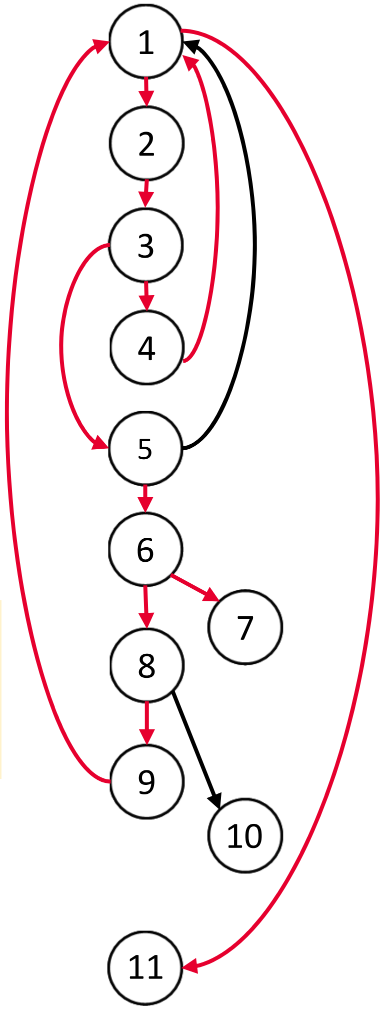


Example: Branch Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

CFG

Test 1: "()"
Test 2: ")"
Test 3: "(]"



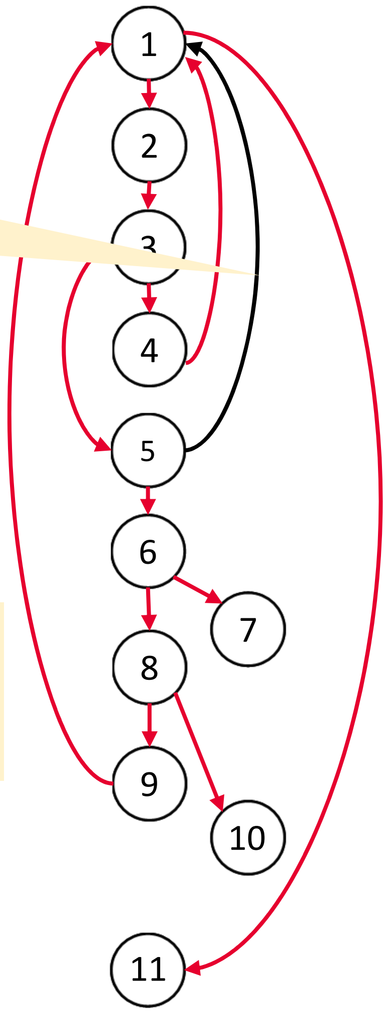
Example: Branch Coverage

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        else if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c)) {  (8)
                stack.pop();  (9)
            } else { return false; }  (10)
        }
    } return stack.isEmpty();  (11)
}
```

(1) -> (5) not yet covered by any test

CFG

Test 1: "()"
Test 2: ")"
Test 3: "(]"

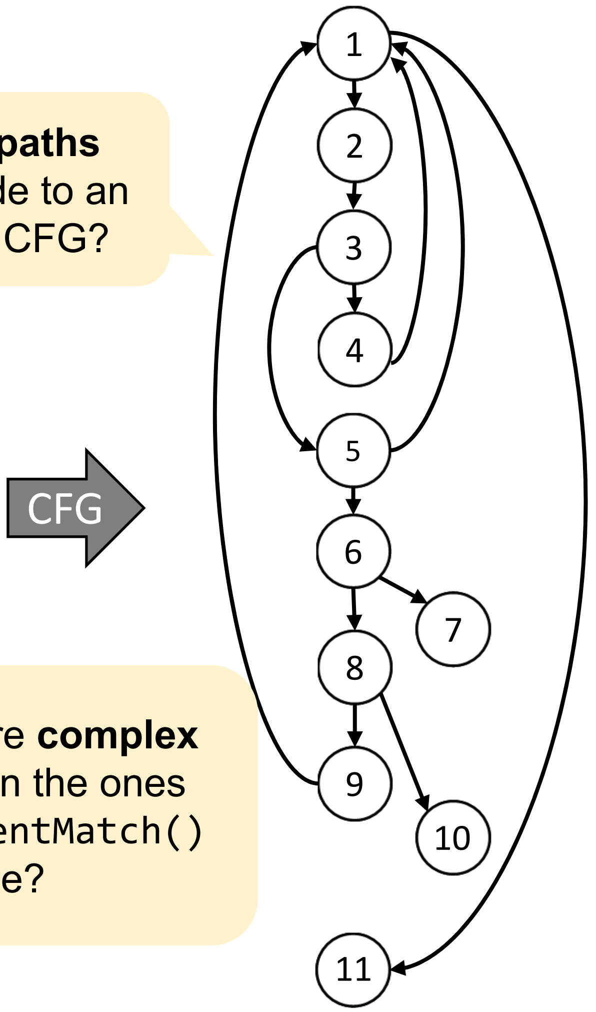


Outlook

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {  (1)
        char c = line.charAt(i);  (2)
        if (isLeftParen(c)) {  (3)
            stack.push(matchingRightParen(c));  (4)
        }
        if (isRightParen(c)) {  (5)
            if (stack.isEmpty()) {  (6)
                return false;  (7)
            }
            if (stack.top().equals(c))
                stack.pop();  (9)
            } else { return false; }  (
        }
    } return stack.isEmpty();  (11)
}
```

What about **all paths** from the start node to an end node of the CFG?

What about more **complex conditions** than the ones used in our `parentMatch()` example?

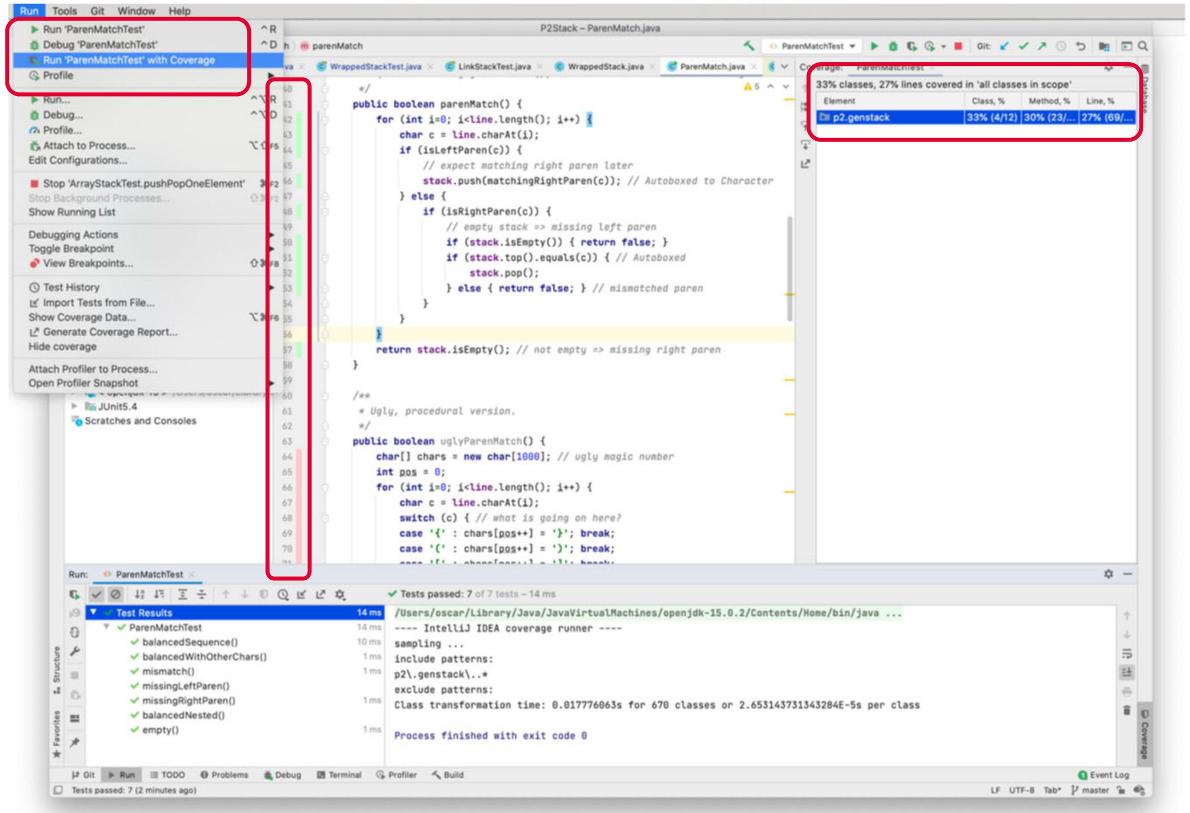


Coverage Tools

A *coverage tool* can tell you what part of your code has been exercised by a test run or an interactive session. This helps you to:

- identify dead code
- Identify missing tests

Running Tests with Coverage



```
     */
    public boolean parenMatch() {
        for (int i=0; i<line.length(); i++) {
            char c = line.charAt(i);
            if (isLeftParen(c)) {
                // expect matching right paren later
                stack.push(matchingRightParen(c)); // Autoboxed to Character
            } else {
                if (isRightParen(c)) {
                    // empty stack => missing left paren
                    if (stack.isEmpty()) { return false; }
                    if (stack.top().equals(c)) { // Autoboxed
                        stack.pop();
                    } else { return false; } // mismatched paren
                }
            }
        }
        return stack.isEmpty(); // not empty => missing right paren
    }

    /**
     * Ugly, procedural version.
     */
    public boolean uglyParenMatch() {
        char[] chars = new char[1000]; // ugly magic number
        int pos = 0;
        for (int i=0; i<line.length(); i++) {
            char c = line.charAt(i);
            switch (c) { // what is going on here?
            case '{' : chars[pos++] = '}'; break;
            case '(' : chars[pos++] = ')'; break;
```

33% classes, 27% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
p2.genstack	33% (4/12)	30% (23/...	27% (69/...

✓ Tests passed: 7 of 7 tests – 14 ms

```
✓ Test Results                    14 ms
  ✓ ParenMatchTest                14 ms
    ✓ balancedSequence()          10 ms
    ✓ balancedWithOtherChars()     1 ms
    ✓ mismatch()                   1 ms
    ✓ missingLeftParen()
    ✓ missingRightParen()          1 ms
    ✓ balancedNested()
    ✓ empty()                      1 ms
```

```
/Users/oscar/Library/Java/JavaVirtualMachines/openjdk-15.0.2/Contents/Home/bin/java ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
p2\.genstack\..*
exclude patterns:
Class transformation time: 0.017776063s for 670 classes or 2.653143731343284E-5s per class

Process finished with exit code 0
```

Tests passed: 7 (2 minutes ago)

Outline

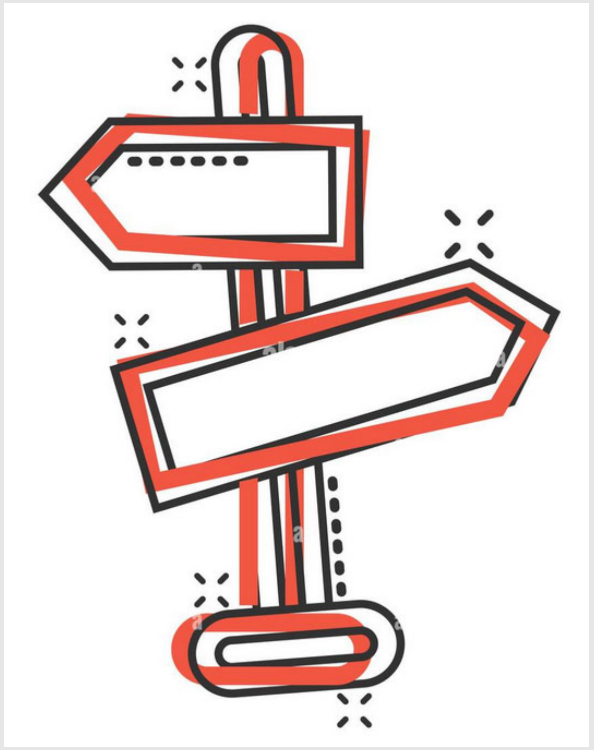
u^b UNIVERSITÄT BERN

Motivation and Background

A Testing Framework

Testing Strategies
- Black-box Testing
- White-box Testing

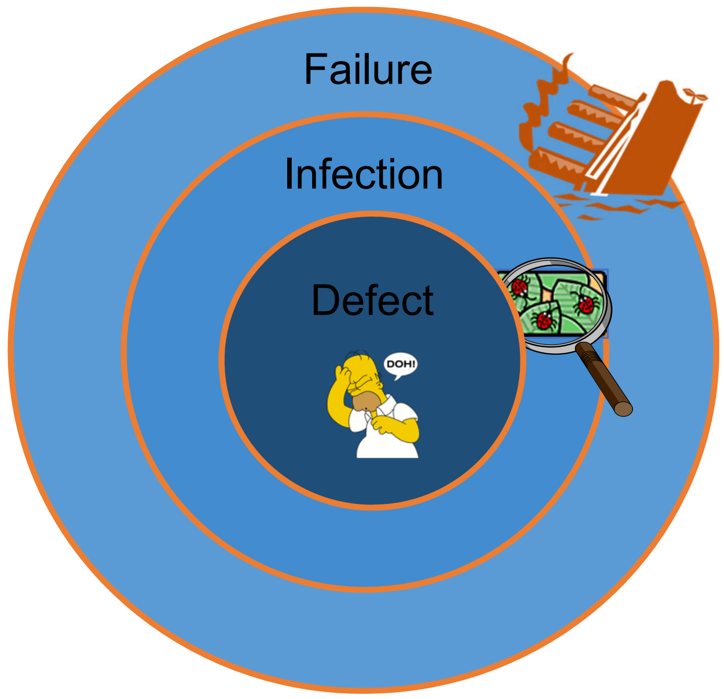
Debugging



UNIVERSITÄT BERN

Debugging

- Every failure can be traced back to some infection, and every infection is caused by some defect.
- Debugging means to relate a given failure to the defect – and to remove the defect.



The Devil's Guide to Debugging

Find the error by guessing

- Scatter print statements randomly throughout the code.
- If the print statements do not reveal the error, start making changes until something appears to work.
- Do not save the original version of the code and do not keep a record of the changes that have been made.

Debugging by superstition

- Why blame yourself when you can blame the computer, the operating system, the compiler, the data, other programmers (especially those ones who write library routines!), and best of all, the stupid users!

Don't waste time trying to understand the problem

- For example, why try to understand why a particular case is not handled by a supposedly general subroutine when you can make a quick fix?

The TRAFFIC Principle

- ❑ **T**rack the problem
- ❑ **R**eproduce the failure
- ❑ **A**utomate and simplify
- ❑ **F**ind possible infection origins
- ❑ **F**ocus on likely origins
- ❑ **I**solate the infection chain
- ❑ **C**orrect the defect

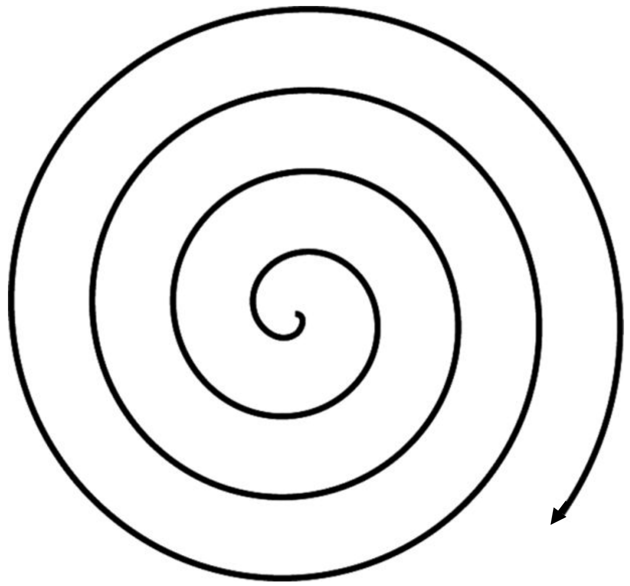
Make use of automated tests

Make use of an interactive debugger

Make use of regression testing to check for potential side effects

The Scientific Debugging Method

- Just like testing and development, debugging is an iterative process.
- Essentially, we formulate, refine, test and validate a debugging hypothesis (i.e., an attempt to understand and explain what is happening)



Example: Running parenMatch.main

```
Please enter parenthesized expressions to test
(empty line to stop)
(hello) (world)
"(hello) (world)" is balanced
static public void main(String args[]) {
"static public void main(String args[]) {" is not balanced
()
"()" is not balanced
}
"}" is balanced
"" is balanced
bye!
```

Observation of a failure

Write a test that demonstrates the failure

```
@Test
public void multipleLines() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("(hello) (world)", stack);
    assertTrue(pm.parenMatch());
    pm = new ParenMatch("static public void main(String args[]) {", stack);
    assertFalse(pm.parenMatch());
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
    pm = new ParenMatch("}", stack);
    assertFalse(pm.parenMatch());
}
```

"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."

-- Martin Fowler

A first Hypothesis

```
@Test
public void unbalancedBalanced() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("(", stack);
    assertFalse(pm.parenMatch());
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
}
```

Refining our Hypothesis

```
@Test
public void unbalancedBalanced() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("(", stack);
    assertFalse(pm.parenMatch());
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
}
```

✗

```
@Test
public void balancedUnbalanced() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
    pm = new ParenMatch(")", stack);
    assertFalse(pm.parenMatch());
}
```

✓

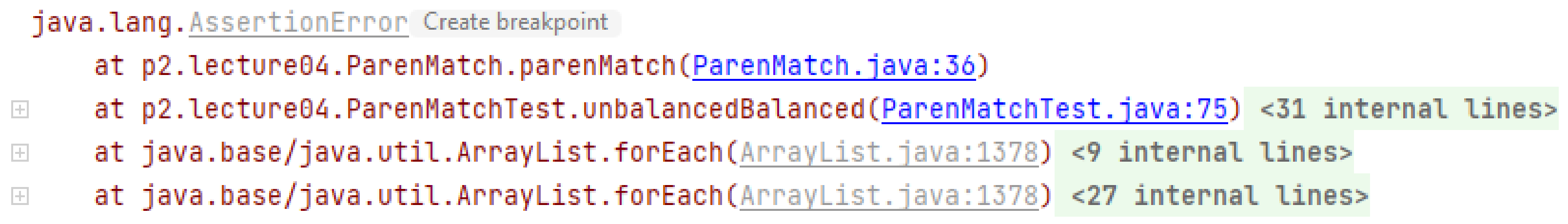
Add the missing Contract ...

```
@Test
public void unbalancedBalanced() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("(", stack);
    assertFalse(pm.parenMatch());
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
}
```

```
public class ParenMatch {
    …
    public boolean parenMatch() {
        assert stack.isEmpty();
        for (int i=0; i<line.length(); i++) {
        …
    }
}
```

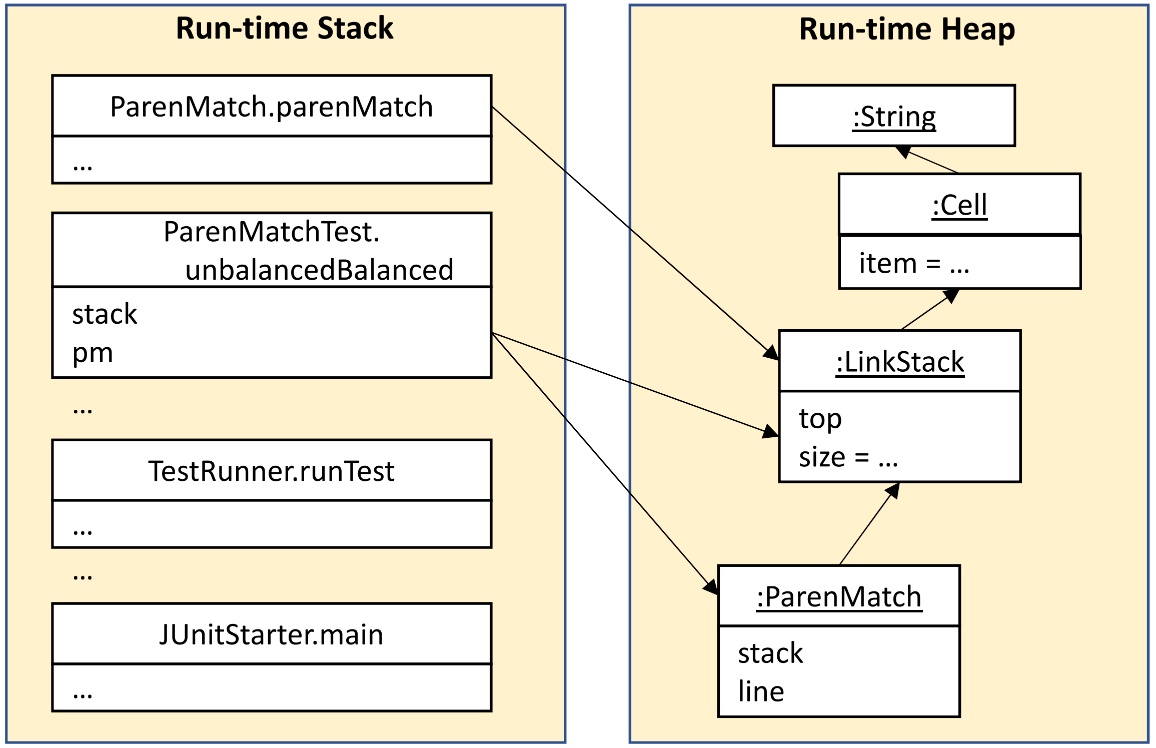
… and run the Test again.

```
@Test
public void unbalancedBalanced() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("(", stack);
    assertFalse(pm.parenMatch());
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
}
```



```
java.lang.AssertionError Create breakpoint
    at p2.lecture04.ParenMatch.parenMatch(ParenMatch.java:36)
    at p2.lecture04.ParenMatchTest.unbalancedBalanced(ParenMatchTest.java:75) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1378) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1378) <27 internal lines>
```

Excursus: Run-time Stack and Heap



The **Heap** grows with each new Object created,

and shrinks when Objects are **garbage-collected**.

Debuggers

A **debugger** is a tool that allows you to examine the state of a running program:

- *step* through the program instruction by instruction
- *view the source code* of the executing program
- *inspect* (and modify) values of variables in various formats
- *set* and unset *breakpoints* anywhere in your program
- *execute* up to a specified breakpoint

Using Debuggers

Interactive debuggers are available for most mature programming languages and integrated in IDEs.

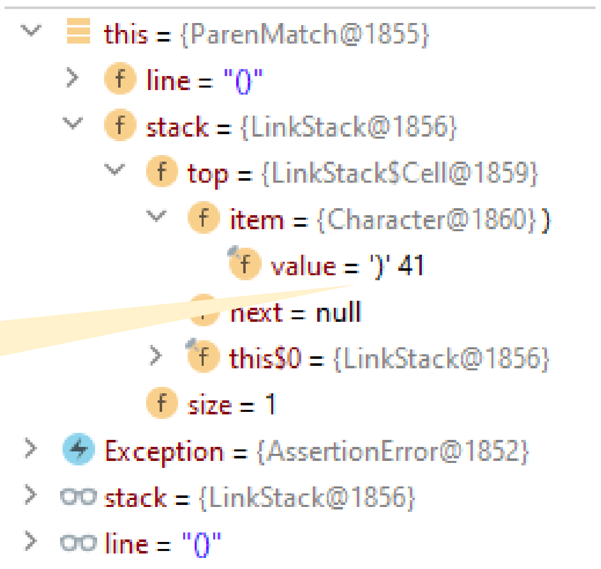
Classical debuggers are *line-oriented* (e.g., jdb); most modern ones are *graphical*.

✎ When should you use a debugger?

✓ *When you are unsure why (or where) your program is not working.*

Verifying our Hypothesis

```
@Test
public void unbalancedBalanced() {
    IStack stack = new LinkStack<Character>();
    pm = new ParenMatch("(", stack);
    assertFalse(pm.parenMatch());
    pm = new ParenMatch("()", stack);
    assertTrue(pm.parenMatch());
}
```



```
v this = {ParenMatch@1855}
  > line = "()"
  v stack = {LinkStack@1856}
    v top = {LinkStack$Cell@1859}
      v item = {Character@1860} )
          value = ')' 41
        next = null
      > this$0 = {LinkStack@1856}
    size = 1
> Exception = {AssertionError@1852}
> stack = {LinkStack@1856}
> line = "()"
```

Indeed, the closing bracket ")" to be expected by the previous call to `parenMatch()` is still on our stack.

With TDD, we may have avoided the bug…

```
public class ParenMatchTest {
    protected ParenMatch pm;

    @BeforeEach
    public void setUp() {
        IStack stack = new LinkStack<Character>();
        pm = new ParenMatch(stack);
    }

    @Test
    public void unbalancedBalanced() {
        assertFalse(pm.parenMatch("()"));
        assertTrue(pm.parenMatch("()"));
    }
    ...
}
```

Testing early can have a positive influence on the design process.

Pass the string to check directly as an argument to the parenMatch method, which should also reset the internal stack to be empty

What you should know!

- How does a framework differ from a library?
- What is a unit test?
- What is an annotation?
- What is a test "fixture"?
- What is a regression test? Why is it important?
- What strategies should you apply to design a test?
- How does test-driven development work?
- How can testing drive design?
- What are the run-time stack and heap?

Can you answer these questions?

– Why can't you use tests to demonstrate absence of defects?
– How do you know when you have written enough tests?
– How many assertions should a test contain?
– Is it better to write long test scenarios or short, independent tests?
– Why doesn't Java allocate objects on the run-time stack?