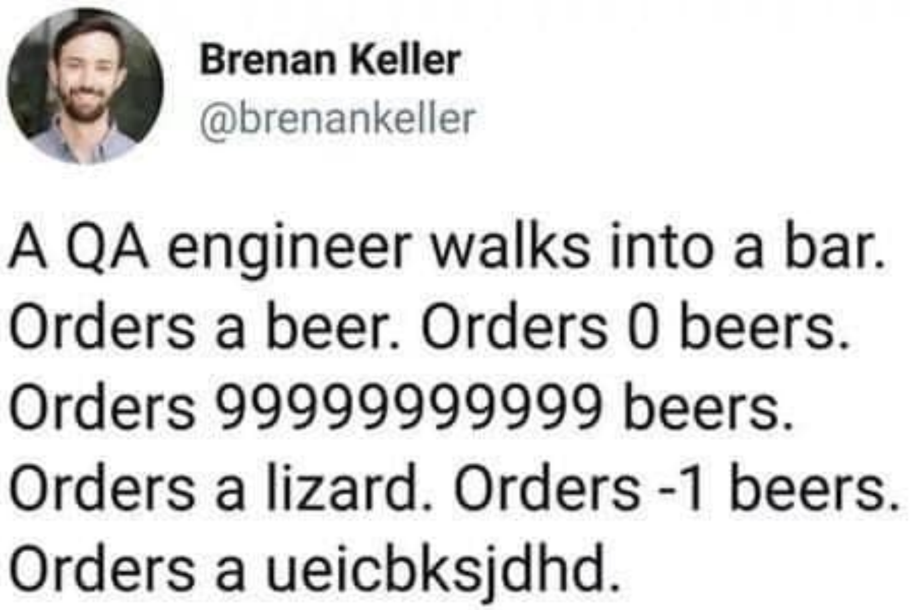
Software Testing, Verification and Validation

October 20, 2023
Week #6 — Lecture #5

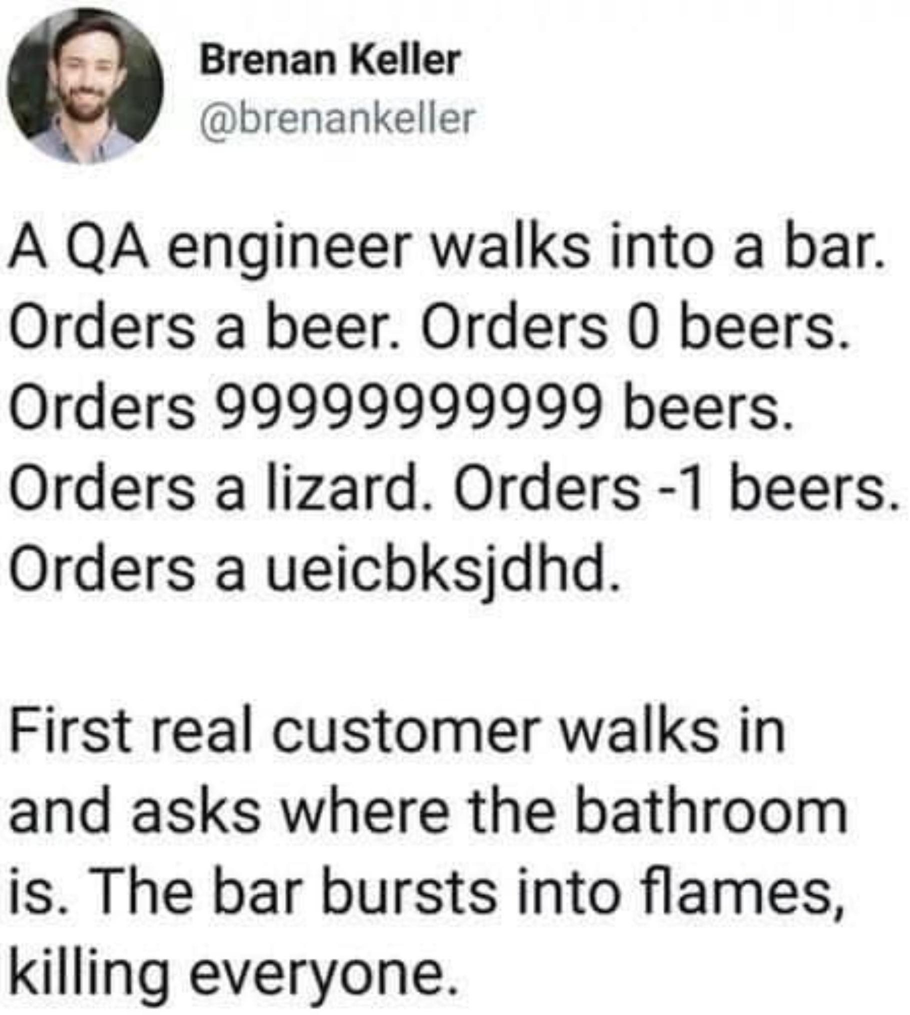
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we introduced model-based testing as part of our set of black-box techniques. This week we are moving to white-box techniques and therefore we will use the source code itself as a source of information to create tests.



Brenan Keller
@brenankeller

A QA engineer walks into a bar.
Orders a beer. Orders 0 beers.
Orders 99999999999 beers.
Orders a lizard. Orders -1 beers.
Orders a ueicbksjdhd.

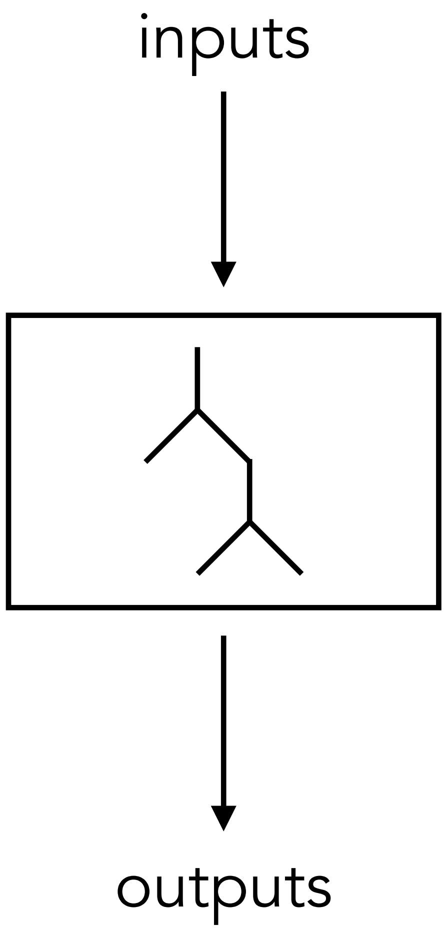


Brenan Keller
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

White-box testing



inputs

outputs

Knowledge sources

Source code

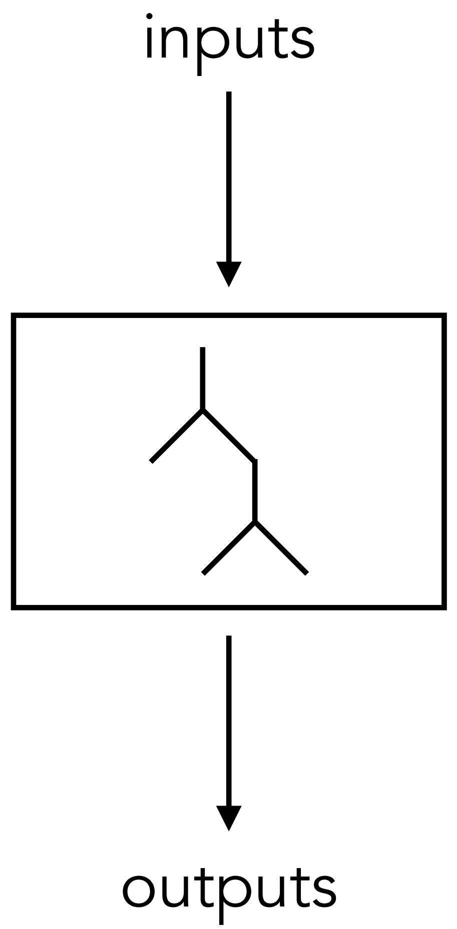
Control flow graphs

Data flow graphs

Cyclomatic complexity

…

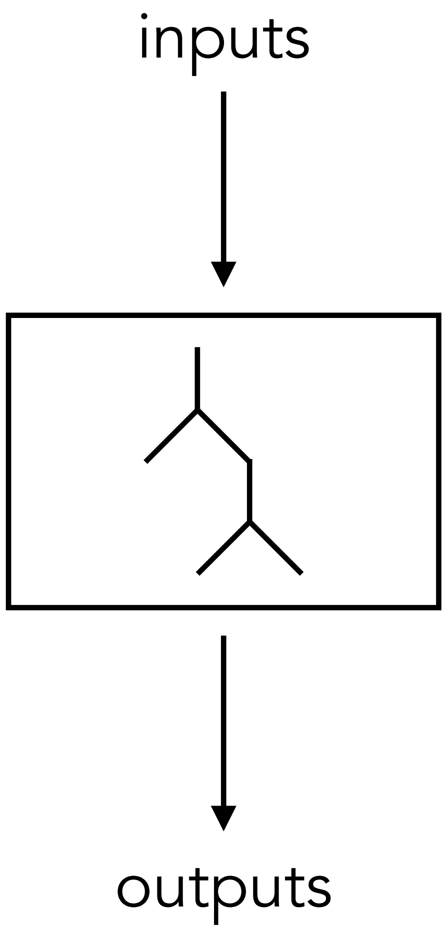
White-box testing



Techniques

- Control flow analysis
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Modified Condition/Decision coverage
  - Path coverage
- Data flow testing/coverage
  - All def
  - All p-uses
  - All c-uses
- Mutation testing
- …

White-box testing



Techniques

- 👉 Control flow analysis
  - 👉 Statement coverage
  - 👉 Branch coverage
  - Condition coverage
  - Modified Condition/Decision coverage
  - Path coverage
- Data flow testing/coverage
  - All def
  - All p-uses
  - All c-uses
- Mutation testing
- …

Adequacy

Since we cannot exhaustively test a software system, the two central questions in software testing are

(a) what does constitute an **adequate** set of test cases?

(b) how do we generate a finite set of test cases that satisfies these adequacy criteria?

Goodenough and Gerhard defined a set of tests as adequate if its correct execution implies no errors in the program, more pragmatic solutions are based on a basic insight: **If some unit of code is not executed, i.e., covered, then by definition, testing cannot reveal any faults contained in it.** The adequacy of a set of tests can therefore be measured by how much of a program is covered by the set. While ideally one would like to know how much of the possible behavior of the program is covered, there is no easy way to quantify coverage of behavior, and therefore the majority of adequacy criteria revolve around proxy measurements related to program code or specifications.

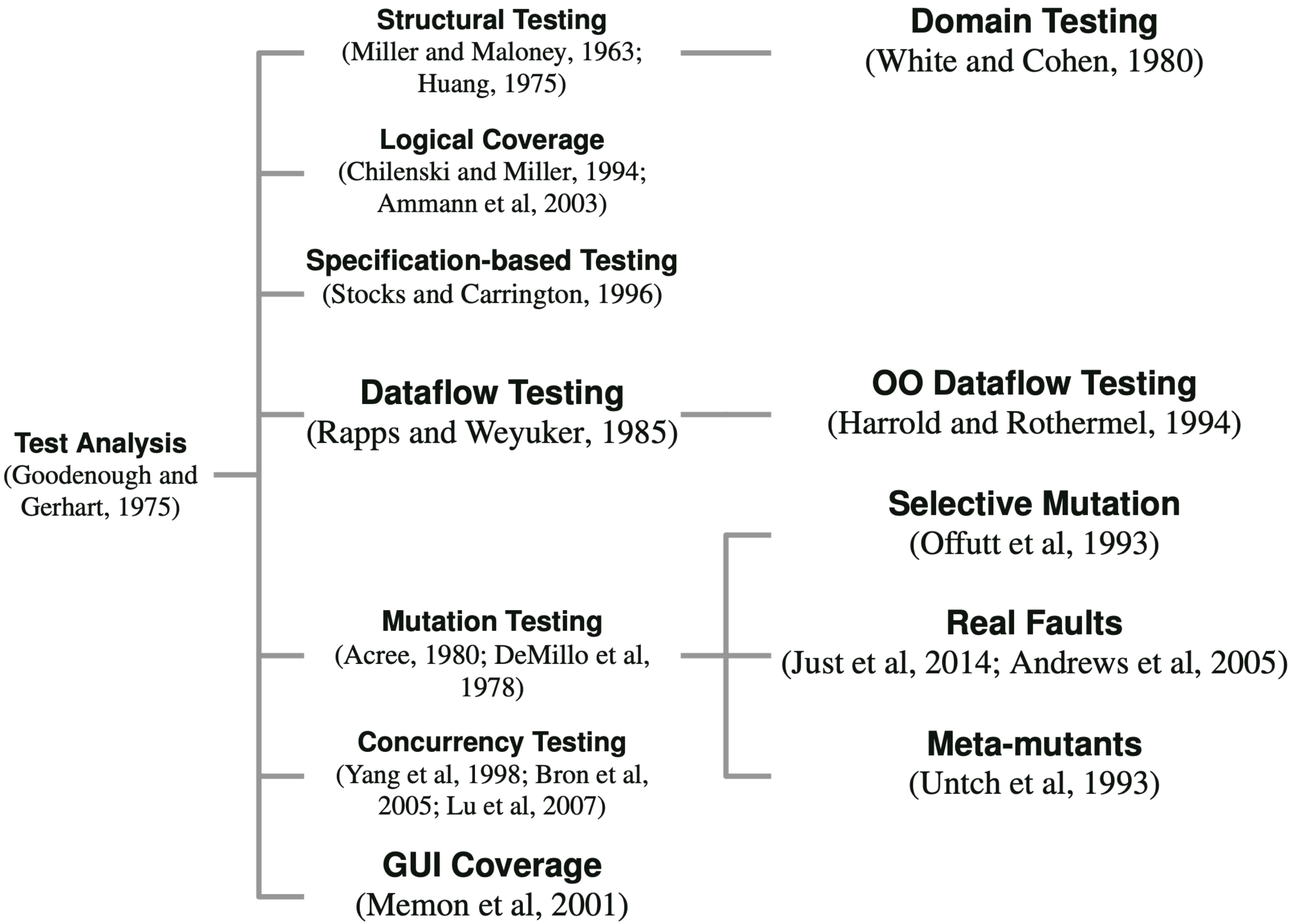
Measurements of Adequacy

Many **coverage criteria** have been proposed over time. A coverage criterion in software testing serves three main purposes.

1. It answers the question of adequacy: Have we tested enough? If so, we can *stop* adding more tests, and the coverage criterion thus serves as a stopping criterion.

2. It provides a way to quantify adequacy: We can measure not only whether we have tested enough based on our adequacy criterion, but also *how much* of the underlying proxy measurement we have covered. Even if we have not fully covered a program, does a given set of tests represent a decent effort, or has the program not been tested at all?

3. Coverage criteria can serve as generation criteria that help a tester decide what test to add next.



Test Analysis
(Goodenough and Gerhart, 1975)

Structural Testing
(Miller and Maloney, 1963; Huang, 1975)

Domain Testing
(White and Cohen, 1980)

Logical Coverage
(Chilenski and Miller, 1994; Ammann et al, 2003)

Specification-based Testing
(Stocks and Carrington, 1996)

Dataflow Testing
(Rapps and Weyuker, 1985)

OO Dataflow Testing
(Harrold and Rothermel, 1994)

Mutation Testing
(Acree, 1980; DeMillo et al, 1978)

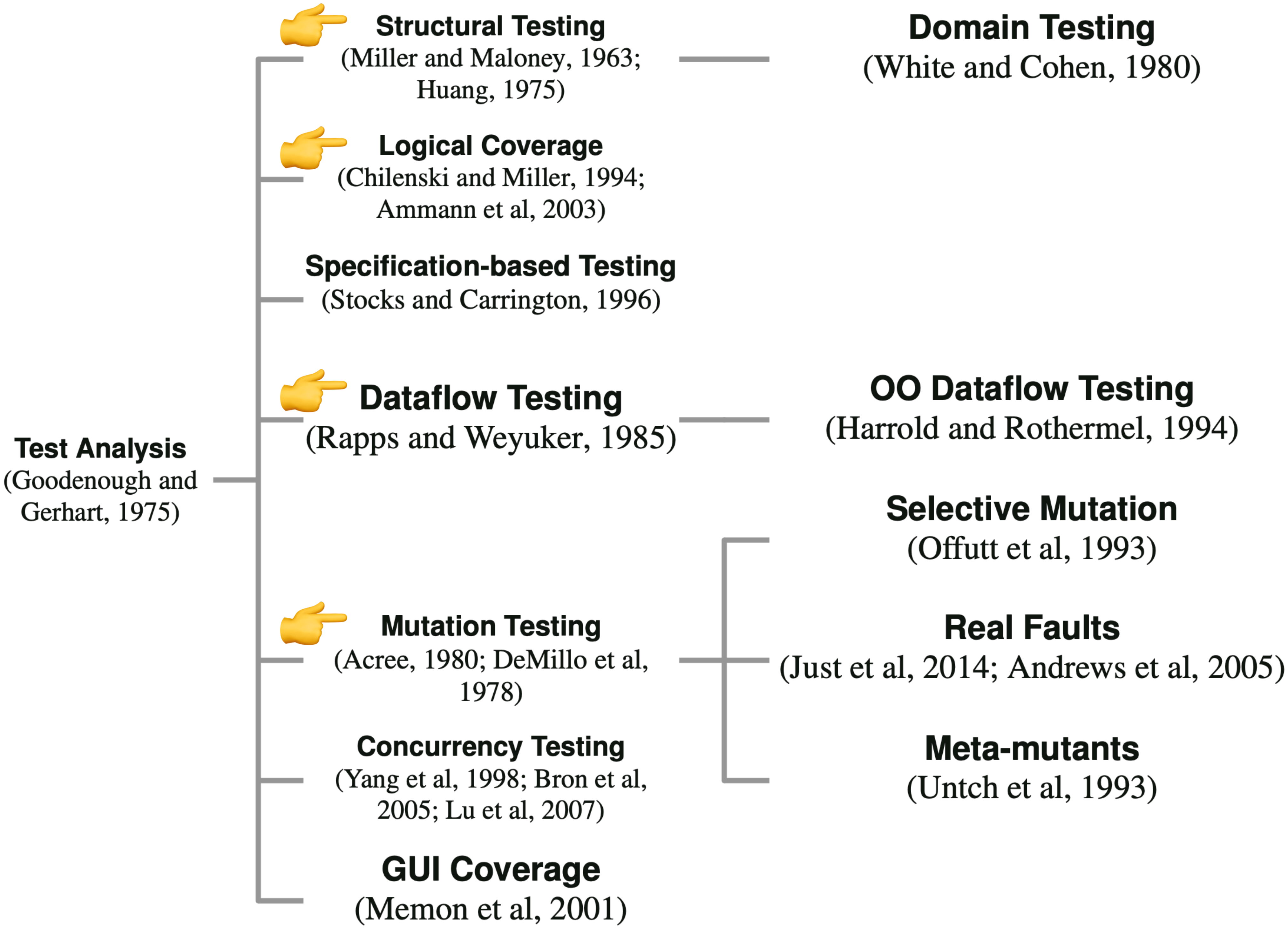
Selective Mutation
(Offutt et al, 1993)

Real Faults
(Just et al, 2014; Andrews et al, 2005)

Meta-mutants
(Untch et al, 1993)

Concurrency Testing
(Yang et al, 1998; Bron et al, 2005; Lu et al, 2007)

GUI Coverage
(Memon et al, 2001)



Test Analysis
(Goodenough and Gerhart, 1975)

👉 **Structural Testing**
(Miller and Maloney, 1963; Huang, 1975)

Domain Testing
(White and Cohen, 1980)

👉 **Logical Coverage**
(Chilenski and Miller, 1994; Ammann et al, 2003)

Specification-based Testing
(Stocks and Carrington, 1996)

👉 **Dataflow Testing**
(Rapps and Weyuker, 1985)

OO Dataflow Testing
(Harrold and Rothermel, 1994)

👉 **Mutation Testing**
(Acree, 1980; DeMillo et al, 1978)

Selective Mutation
(Offutt et al, 1993)

Real Faults
(Just et al, 2014; Andrews et al, 2005)

Meta-mutants
(Untch et al, 1993)

Concurrency Testing
(Yang et al, 1998; Bron et al, 2005; Lu et al, 2007)

GUI Coverage
(Memon et al, 2001)

Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6

Structural Testing

Structural Testing

The idea of code coverage is intuitive and simple: If no test executes a *faulty* statement, then the defect cannot be found; hence every statement should be covered by some test.

Structural Testing

1) To systematically derive tests from source code.

2) To know when to stop testing.

As a tester, when performing specification-based testing, your goal was clear: to derive classes out of the requirement specifications, and then to derive test cases for each of the classes. You were satisfied once all the classes and boundaries were systematically exercised.

The same idea applies to structural testing. First, it gives us a systematic way to devise tests. As we will see, a tester might focus on testing all the lines of a program; or focus on the branches and conditions of the program. Different criteria produce different test cases.

Second, to know when to stop. It is easy to imagine that the number of possible paths in a mildly complex piece of code is just too large, and exhaustive testing is impossible. Therefore, having clear criteria on when to stop helps testers in understanding the costs of their testing.

Line Coverage

Line Coverage

Line coverage is the most basic criterion; a set of test cases is considered to be adequate according to line coverage if all lines of code have been executed.

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

```
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

What would you test?
(now, only looking to the source code)

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

First idea: “going through all the lines”.

If our test suite exercises all the lines, we should be happy, right? 🤔

Let's assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

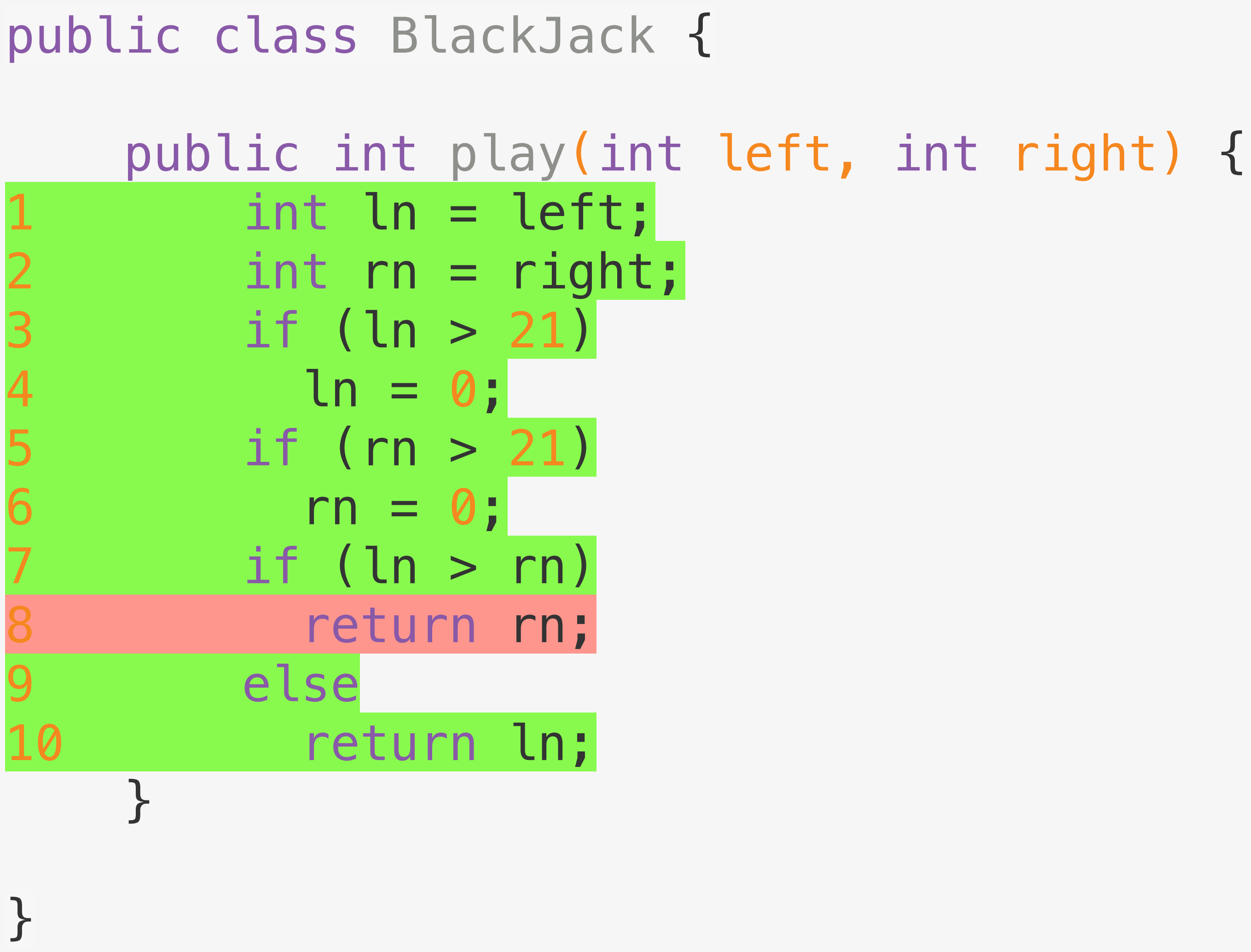
```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

First idea: "going through all the lines".

If our test suite exercises all the lines, we should be happy, right? 🤔

t1 = (30, 30)

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.



```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

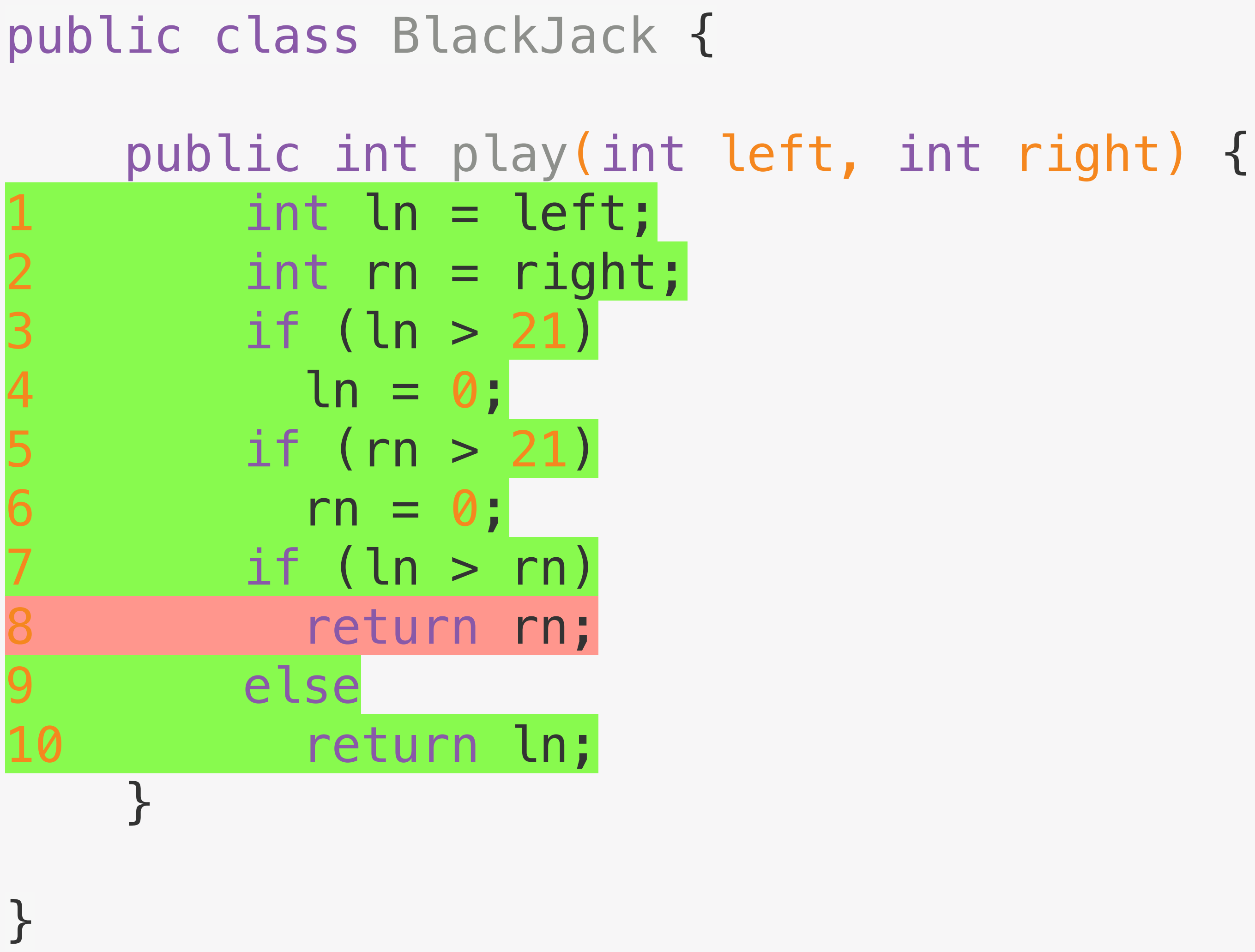
First idea: “going through all the lines”.

If our test suite exercises all the lines, we should be happy, right? 🤔

t1 = (30, 30)
9/10 = 90% line coverage

$$\text{line coverage} = \frac{\text{\# lines covered}}{\text{\# lines}} \times 100\%$$

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.



```java
public class BlackJack {

  public int play(int left, int right) {
    int ln = left;
    int rn = right;
    if (ln > 21)
     ln = 0;
    if (rn > 21)
     rn = 0;
    if (ln > rn)
     return rn;
    else
     return ln;
  }

}
```

First criteria: “going through all the lines”.

If our test suite exercises all the lines, we are happy.

t1 = (30, 30)
t2 = (?, ?)

Let's assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

First criteria: "going through all the lines".

If our test suite exercises all the lines, we are happy.

t1 = (30, 30)
t2 = (10, 9)

10/10 = 100%

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

First criteria: “going through all the lines”.

If our test suite exercises all the lines, we are happy.

t1 = (30, 30)
t2 = (10, 9)

10/10 = 100%

An interesting aspect of line coverage is that it is quite easy to visualize the achieved coverage, in order to help developers improve the code coverage of their tests.

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

Is this useful?

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

Is this useful?

t1 = (30, 30) ✅
t2 = (10, 9) ❌

Let’s assume a program that receives the number of points of two blackjack players. The program must return the number of points of the winner. In blackjack, whoever gets closer to 21 points wins. If a player goes over 21 points, the player loses. If both players lose, the program must return 0.

```
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;  ←  ln
        else
          return ln;  ←  rn
    }

}
```

Is this useful?

Yes. By exercising all lines of code, we managed to detect two faults!

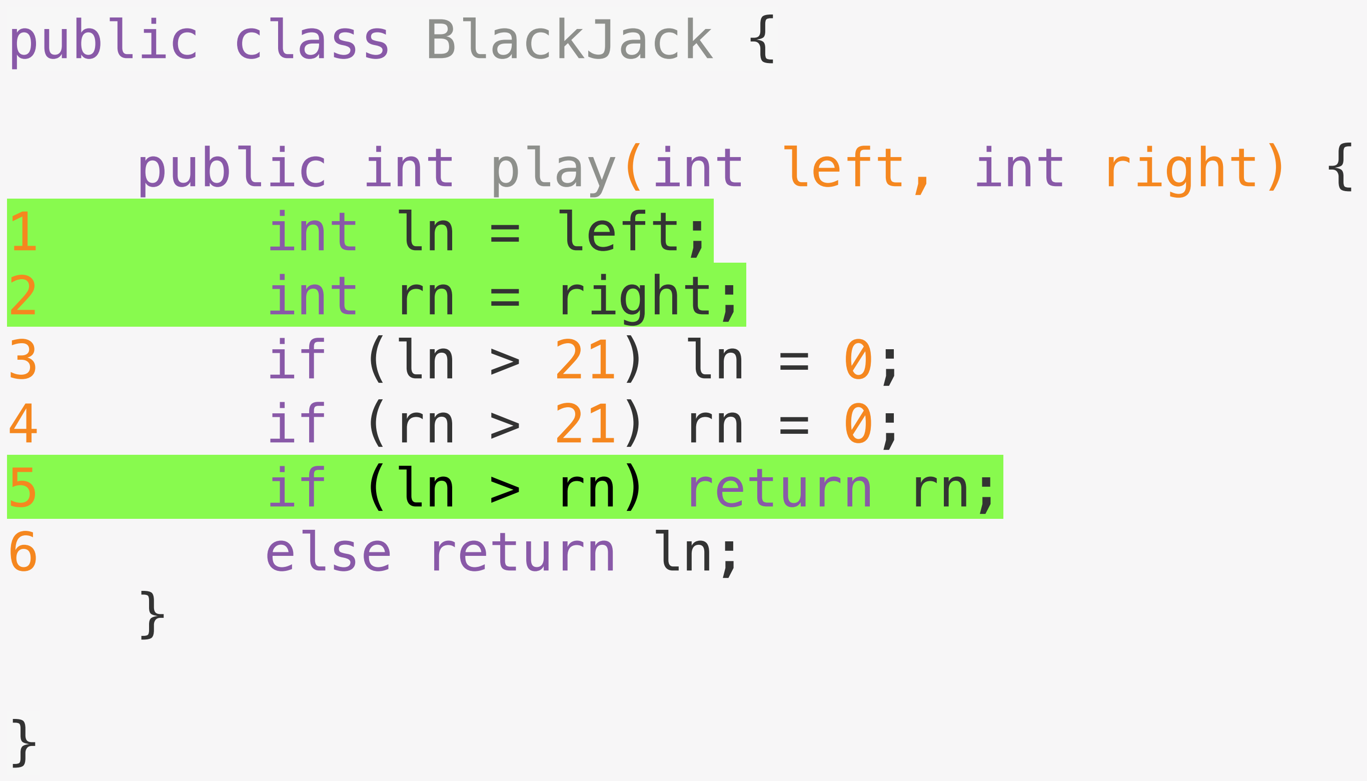
t1 = (30, 30) ✅

t2 = (10, 9) ❌

Using lines of code as a way to determine line coverage is a simple and straightforward idea. However, counting the covered lines is not always a good way of calculating the coverage. The number of lines in a piece of code depends on the decisions taken by the programmer who writes the code.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21) ln = 0;
        if (rn > 21) rn = 0;
        if (ln > rn) return rn;
        else return ln;
    }

}
```

Using lines of code as a way to determine line coverage is a simple and straightforward idea. However, counting the covered lines is not always a good way of calculating the coverage. The number of lines in a piece of code depends on the decisions taken by the programmer who writes the code.



```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21) ln = 0;
4       if (rn > 21) rn = 0;
5       if (ln > rn) return rn;
6       else return ln;
    }

}
```

t2 = (10, 9)
4/10 = 40% line coverage, previously

3/6 = 50% line coverage, now

Using lines of code as a way to determine line coverage is a simple and straightforward idea. However, counting the covered lines is not always a good way of calculating the coverage. The number of lines in a piece of code depends on the decisions taken by the programmer who writes the code.

```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21) ln = 0;
4       if (rn > 21) rn = 0;
5       if (ln > rn) return rn;
6       else return ln;
    }

}
```

t2 = (10, 9)
4/10 = 40% line coverage, previously

3/6 = 50% line coverage, now

Some coverage tools measure coverage at statement level. Statements are the unique instructions that your JVM, for example, executes. This is a bit better, as splitting one line of code in two would not make a difference.

Is there anything else I could do with code coverage?

SPOILER ALERT

Localize the fault! Automatically!

```java
public Complex reciprocal() {
    if (isNaN)
        return NaN;
    if (real == 0.0 && imaginary == 0.0)
        return NaN;
    if (isInfinite)
        return ZERO;
    if (FastMath.abs(real) < FastMath.abs(imaginary)) {
        double q = real / imaginary;
        double scale = 1.0 / (real * q + imaginary);
        return createComplex(scale * q, -scale);
    } else {
        double q = imaginary / real;
        double scale = 1.0 / (imaginary * q + real);
        return createComplex(scale, -scale * q);
    }
}
```

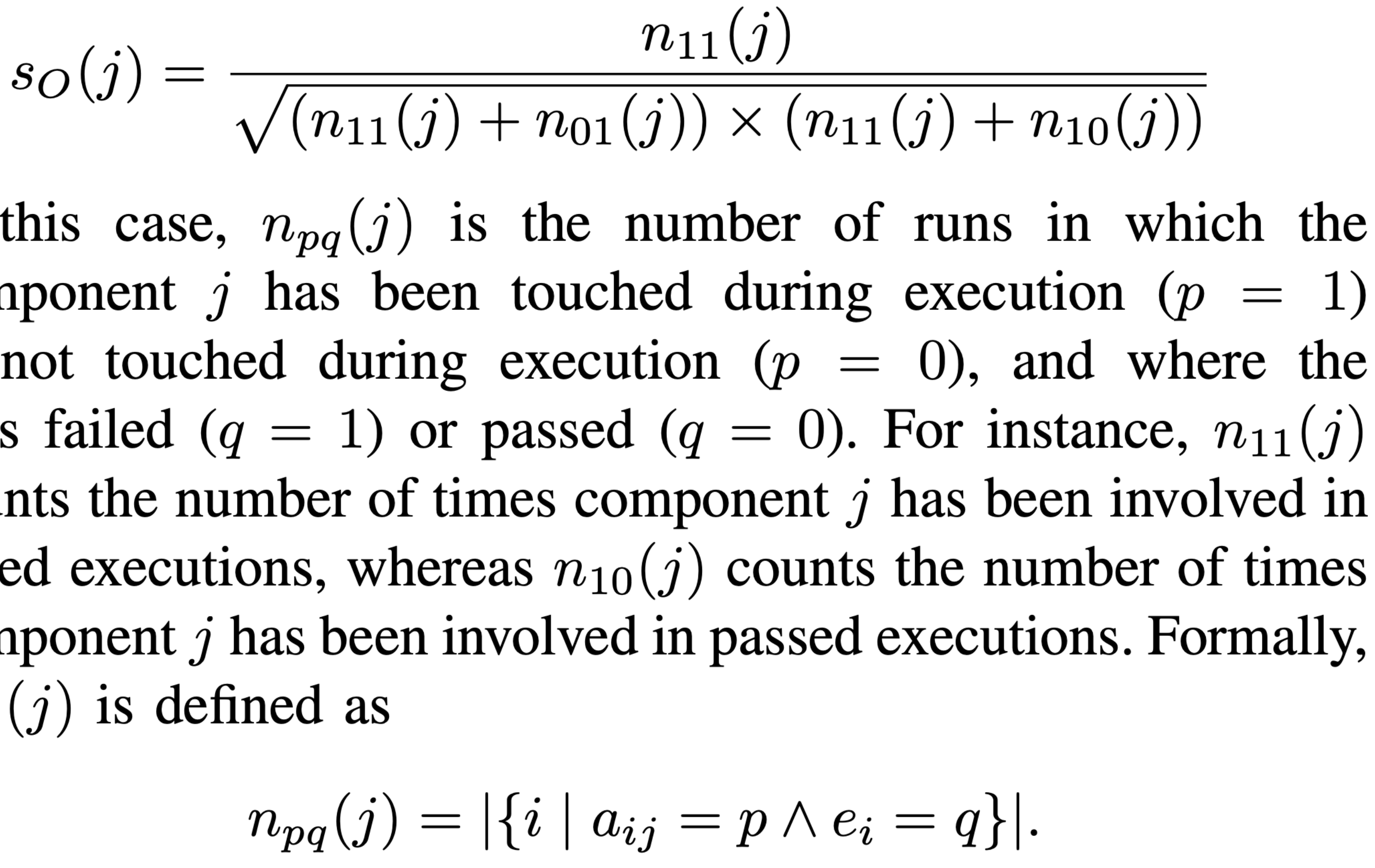
`Reciprocal` function from the Apache Commons Math project. This function returns the multiplicative inverse of `this` element.

t1	t2	t3	t4	t5	t6
✅	✅	✅	✅	✅	❌

```java
public Complex reciprocal() {
    if (isNaN)
        return NaN;
    if (real == 0.0 && imaginary == 0.0)
        return NaN;
    if (isInfinite)
        return ZERO;
    if (FastMath.abs(real) < FastMath.abs(imaginary)) {
        double q = real / imaginary;
        double scale = 1.0 / (real * q + imaginary);
        return createComplex(scale * q, -scale);
    } else {
        double q = imaginary / real;
        double scale = 1.0 / (imaginary * q + real);
        return createComplex(scale, -scale * q);
    }
}
```

		t1	t2	t3	t4	t5	t6
		✅	✅	✅	✅	✅	❌
	`public Complex reciprocal() {`						
1	`    if (isNaN)`	●	●	●	●	●	🐞
2	`        return NaN;`					●	
3	`    if (real == 0.0 && imaginary == 0.0)`	●	●	●	●		🐞
4	`        return NaN;`						🐞
5	`    if (isInfinite)`	●	●	●	●		
6	`        return ZERO;`				●		
7	`    if (FastMath.abs(real) < FastMath.abs(imaginary)) {`	●	●	●			
8	`        double q = real / imaginary;`	●		●			
9	`        double scale = 1.0 / (real * q + imaginary);`	●		●			
10	`        return createComplex(scale * q, -scale);`	●		●			
11	`    } else {`		●				
12	`        double q = imaginary / real;`		●				
13	`        double scale = 1.0 / (imaginary * q + real);`		●				
14	`        return createComplex(scale, -scale * q);`		●				
	`    }`						
	`}`						

Ochiai coefficient



$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}}$$

in this case, $n_{pq}(j)$ is the number of runs in which the component j has been touched during execution ($p = 1$) or not touched during execution ($p = 0$), and where the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(j)$ counts the number of times component j has been involved in failed executions, whereas $n_{10}(j)$ counts the number of times component j has been involved in passed executions. Formally, $n_{pq}(j)$ is defined as

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}|.$$

		t1	t2	t3	t4	t5	t6	
		✅	✅	✅	✅	✅	❌	
	`public Complex reciprocal() {`							
1	`    if (isNaN)`	●	●	●	●	●	🐞	
2	`        return NaN;`					●		0.000
3	`    if (real == 0.0 && imaginary == 0.0)`	●	●	●	●		🐞	
4	`        return NaN;`						🐞	
5	`    if (isInfinite)`	●	●	●	●			0.000
6	`        return ZERO;`				●			0.000
7	`    if (FastMath.abs(real) < FastMath.abs(imaginary)) {`	●	●	●				0.000
8	`        double q = real / imaginary;`	●		●				0.000
9	`        double scale = 1.0 / (real * q + imaginary);`	●		●				0.000
10	`        return createComplex(scale * q, -scale);`	●		●				0.000
11	`    } else {`		●					0.000
12	`        double q = imaginary / real;`		●					0.000
13	`        double scale = 1.0 / (imaginary * q + real);`		●					0.000
14	`        return createComplex(scale, -scale * q);`		●					0.000
	`    }`							
	`}`							

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}}$$

		t1	t2	t3	t4	t5	t6	
		✓	✓	✓	✓	✓	✗	
	`public Complex reciprocal() {`							
1	`    if (isNaN)`	●	●	●	●	●	●	0.408
2	`        return NaN;`					●		0.000
3	`    if (real == 0.0 && imaginary == 0.0)`	●	●	●	●		●	0.447
4	`        return NaN; /* FAULT */`						🐞	1.000
5	`    if (isInfinite)`	●	●	●	●			0.000
6	`        return ZERO;`				●			0.000
7	`    if (FastMath.abs(real) < FastMath.abs(imaginary)) {`	●	●	●				0.000
8	`        double q = real / imaginary;`	●		●				0.000
9	`        double scale = 1.0 / (real * q + imaginary);`	●		●				0.000
10	`        return createComplex(scale * q, -scale);`	●		●				0.000
11	`    } else {`		●					0.000
12	`        double q = imaginary / real;`		●					0.000
13	`        double scale = 1.0 / (imaginary * q + real);`		●					0.000
14	`        return createComplex(scale, -scale * q);`		●					0.000
	`    }`							
	`}`							

The program contains a fault in line 4. It should be `return INF;` instead of `return NaN;`

SPOILER ALERT

Automatic Program Repair!

Decision (or branch) Coverage

Statement/Line coverage is generally seen as a weak criterion, although in practice it is one of the most common criteria used. One reason for this is that it is very intuitive and easy to understand. Stronger criteria are often based on the control flow graph of the program under test. For example, consider the following snippet:

```
if (e < 0)
    e = 0;
if (b == 0)
    b = 1;
```

Statement/Line coverage is generally seen as a weak criterion, although in practice it is one of the most common criteria used. One reason for this is that it is very intuitive and easy to understand. Stronger criteria are often based on the control flow graph of the program under test. For example, consider the following snippet:

```
if (e < 0)
    e = 0;
if (b == 0)
    b = 1;
```

It is possible to achieve 100% statement coverage of this snippet with a single test where `e` is less than 0 and `b` equals 0. This test case would make both if conditions evaluate to `true`, but there would be no test case where either of the conditions evaluates to `false`. **Branch coverage captures the notion of coverage of all edges in the control flow graph, which means that each if condition requires at least one test where it evaluates to `true`, and at least one test where it evaluates to `false`.** In the case of above snippet, we would need at least two test cases to achieve 100% branch coverage, i.e., t1: `e<0` and `b==0`; t2: `e>=0` and `b!=0`.

Decision (or Branch) Coverage

Complex programs often rely on lots of complex conditions (e.g., `if` statements composed of many conditions). When testing these programs, aiming at 100% line coverage might not be enough to cover all the cases we want. We need a stronger criterion.

Decision coverage (or branch coverage) works similar to line and statement coverage, except with branch coverage we count (or aim at covering) all the possible decision outcomes.

A set of test cases will achieve 100% decision (or branch) coverage when tests exercise all the possible outcomes of all decision blocks.

Decision (or Branch) Coverage

Decisions (or branches) are easy to identify in a **Control-Flow Graph** (CFG). A control-flow graph is a representation of all paths that might be traversed during the execution of a piece of code. It consists of **basic blocks**, **decision blocks**, and **arrows/edges** that connect these blocks.

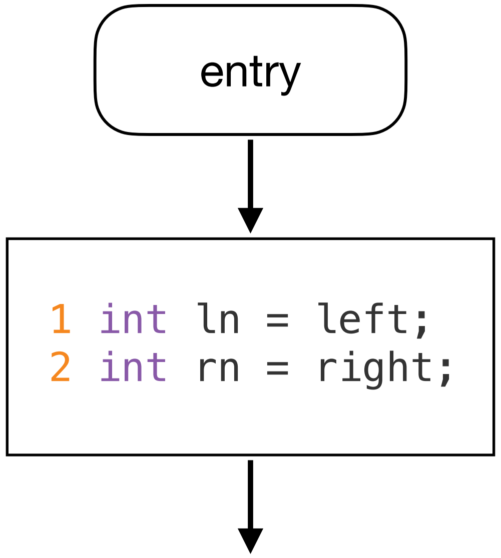
```java
public class BlackJack {

  public int play(int left, int right) {
    int ln = left;
    int rn = right;
    if (ln > 21)
      ln = 0;
    if (rn > 21)
      rn = 0;
    if (ln > rn)
      return rn;
    else
      return ln;
  }

}
```

A basic block is composed of **the maximum number of statements that are executed together no matter what happens**. In the code below, lines 1-2 are always executed together. Basic blocks are often represented by a square:

```
public class BlackJack {

  public int play(int left, int right) {
1   int ln = left;
2   int rn = right;
3   if (ln > 21)
4     ln = 0;
5   if (rn > 21)
6     rn = 0;
7   if (ln > rn)
8     return rn;
9   else
10    return ln;
  }

}
```

A basic block is composed of **the maximum number of statements that are executed together no matter what happens**. In the code below, lines 1-2 are always executed together. Basic blocks are often represented by a square:

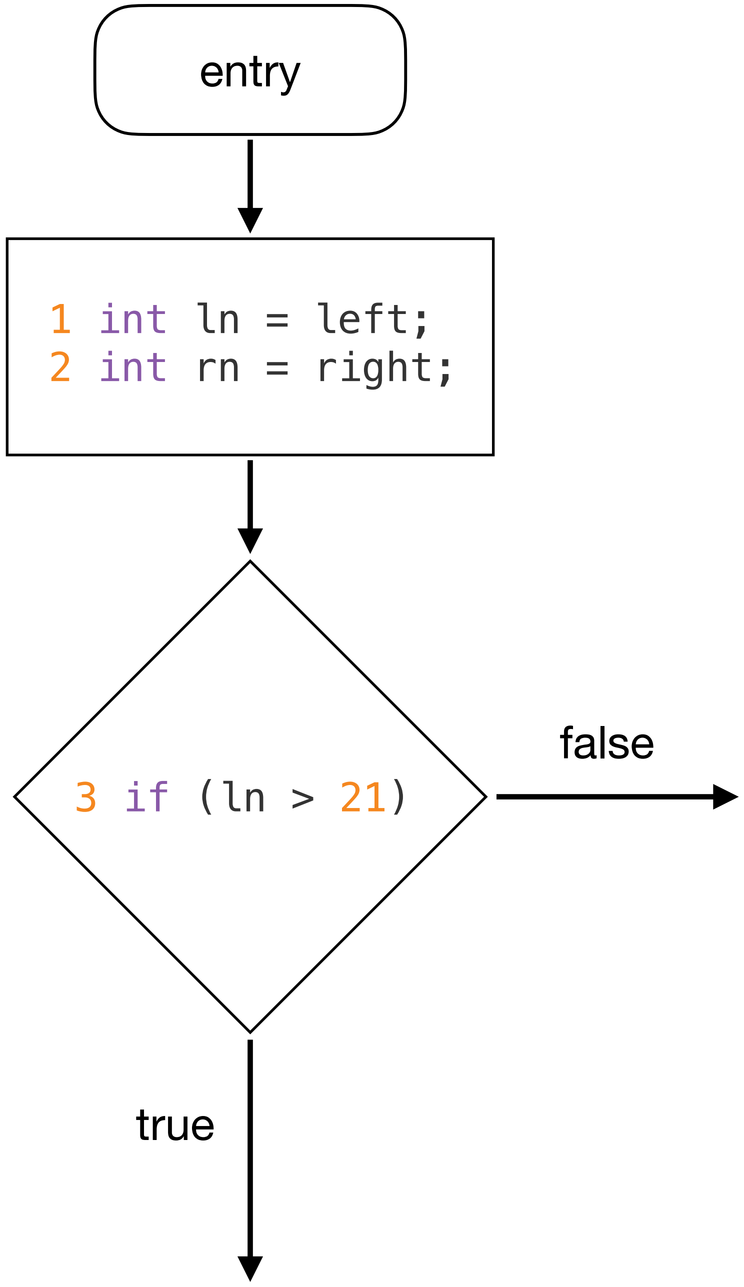
```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21)
4         ln = 0;
5       if (rn > 21)
6         rn = 0;
7       if (ln > rn)
8         return rn;
9       else
10        return ln;
    }

}
```



```
1 int ln = left;
2 int rn = right;
```

A **decision block**, on the other hand, **represents all the statements in the source code that can create different branches**, e.g., line 3. This if statement creates a decision moment in the application: based on the condition, it is decided which code block will be executed next. Decision blocks are often represented by diamonds. This decision block happens right after the basic block we created above, and thus, they are connected by means of an edge.

```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21)
4         ln = 0;
5       if (rn > 21)
6         rn = 0;
7       if (ln > rn)
8         return rn;
9       else
10        return ln;
    }

}
```

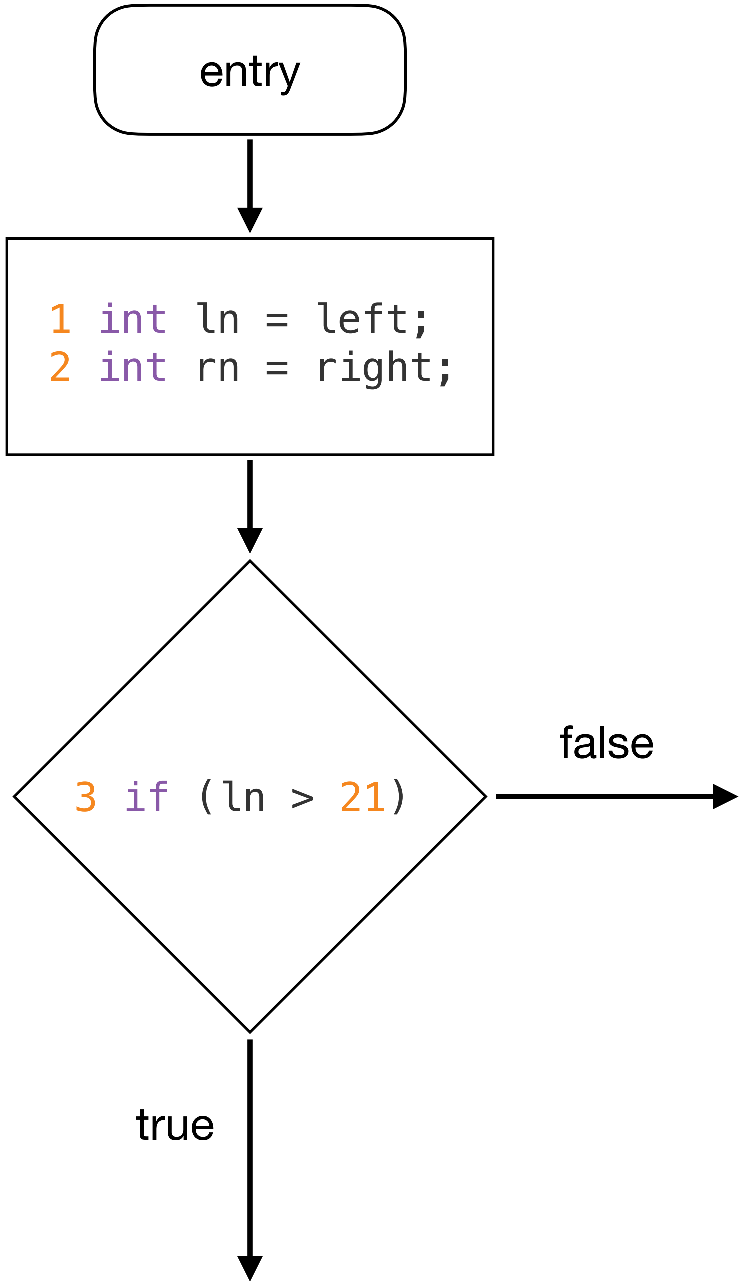


```
1 int ln = left;
2 int rn = right;
```

```
3 if (ln > 21)
```

A **basic block has always a single outgoing edge**. A **decision block**, on the other hand, always **has two outgoing edges (indicating where you go in case of the decision being evaluated to `true`, and where you go in case the decision is evaluated to `false`)**. In case of the decision block being evaluated to `true`, line 4 is executed, and the program continues to line 5. Otherwise, it proceeds straight to line 5, which is another decision block.

```
public class BlackJack {

    public int play(int left, int right) {
1         int ln = left;
2         int rn = right;
3         if (ln > 21)
4           ln = 0;
5         if (rn > 21)
6           rn = 0;
7         if (ln > rn)
8           return rn;
9         else
10          return ln;
    }

}
```



$$\text{branch coverage} = \frac{\text{\# decision outcomes covered}}{\text{\# decision outcomes}} \times 100\%$$

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)  // True ? False ?
          ln = 0;
        if (rn > 21)  // True ?  False ?
          rn = 0;
        if (ln > rn)  // True ?  False ?
          return rn;
        else
          return ln;
    }

}
```

```
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)  // True ✅  False ❌
          ln = 0;
        if (rn > 21)  // True ?  False ?
          rn = 0;
        if (ln > rn)  // True ?  False ?
          return rn;
        else
          return ln;
    }

}
```

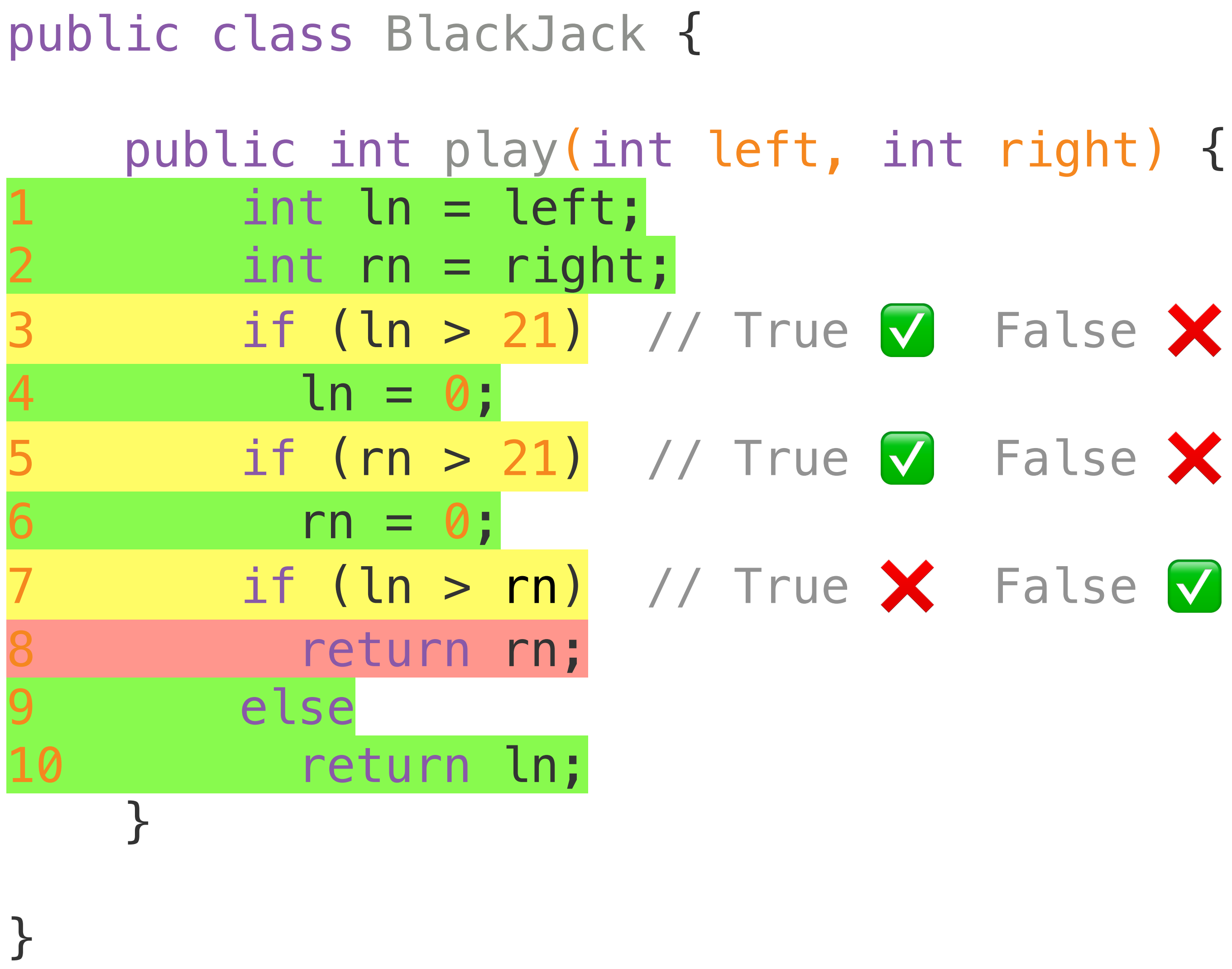
t1 = (30, 30)

```
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)  // True ✅   False ❌
          ln = 0;
        if (rn > 21)  // True ✅   False ❌
          rn = 0;
        if (ln > rn)  // True ?  False ?
          return rn;
        else
          return ln;
    }

}
```

t1 = (30, 30)

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)   // True ✅   False ❌
          ln = 0;
        if (rn > 21)   // True ✅   False ❌
          rn = 0;
        if (ln > rn)   // True ❌   False ✅
          return rn;
        else
          return ln;
    }

}
```

t1 = (30, 30)



```
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)   // True ✓   False ✗
          ln = 0;
        if (rn > 21)   // True ✓   False ✗
          rn = 0;
        if (ln > rn)   // True ✗   False ✓
          return rn;
        else
          return ln;
    }

}
```

t1 = (30, 30)

3/6 = 50% branch coverage

McCabe’s Cyclomatic Complexity

McCabe’s Cyclomatic Complexity

Cyclomatic Complexity (CC) is a metric used for measuring the complexity of a software program. This metric was developed by Thomas J. McCabe in 1976 and it is based on a control-flow representation of the program. In other words, it is a quantitative measure of linearly independent paths in the source code of a software program. A **linearly independent path is defined as a path that has at least one edge which has not been traversed by any other path.**

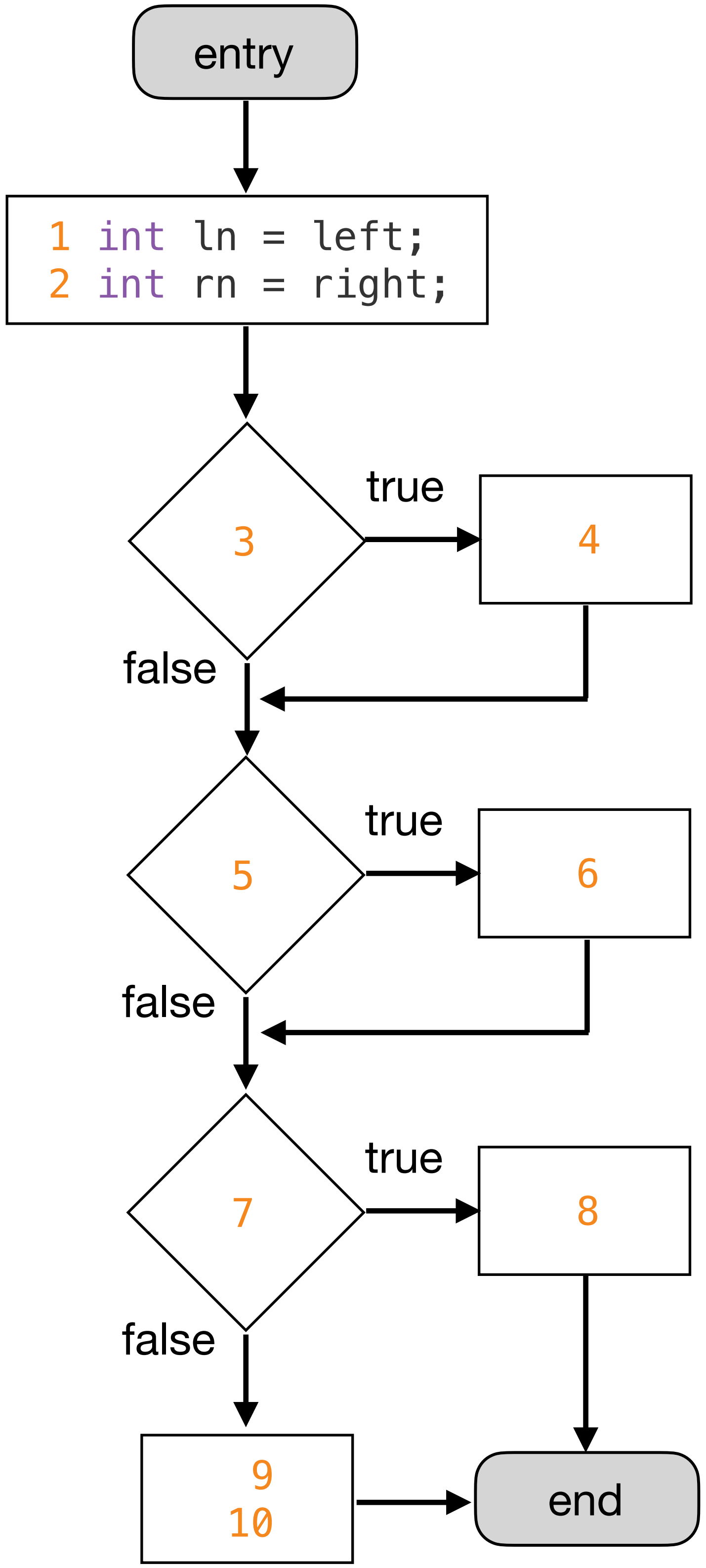
How to compute CC?

1. Construct the control-flow graph with nodes and edges from the source code.

CC, example

1. Construct the control-flow graph with nodes and edges from the source code.

```
public class BlackJack {

  public int play(int left, int right) {
1   int ln = left;
2   int rn = right;
3   if (ln > 21)
4     ln = 0;
5   if (rn > 21)
6     rn = 0;
7   if (ln > rn)
8     return rn;
9   else
10    return ln;
  }

}
```



How to compute CC?

1. Construct the control-flow graph with nodes and edges from the source code.

2. Compute CC (lower CC == better code)

   - of a structured function, e.g., a function with a single exit point, aka return statement, as

$$E - N + 2P$$

   - of a non-structured function, e.g., a function with more than on exit point, aka more than one return statement, as

$$E - N + P$$

E = number of edges

N = number of nodes

P = number of connected components, aka graphs, P is always 1 for single functions

CC, example

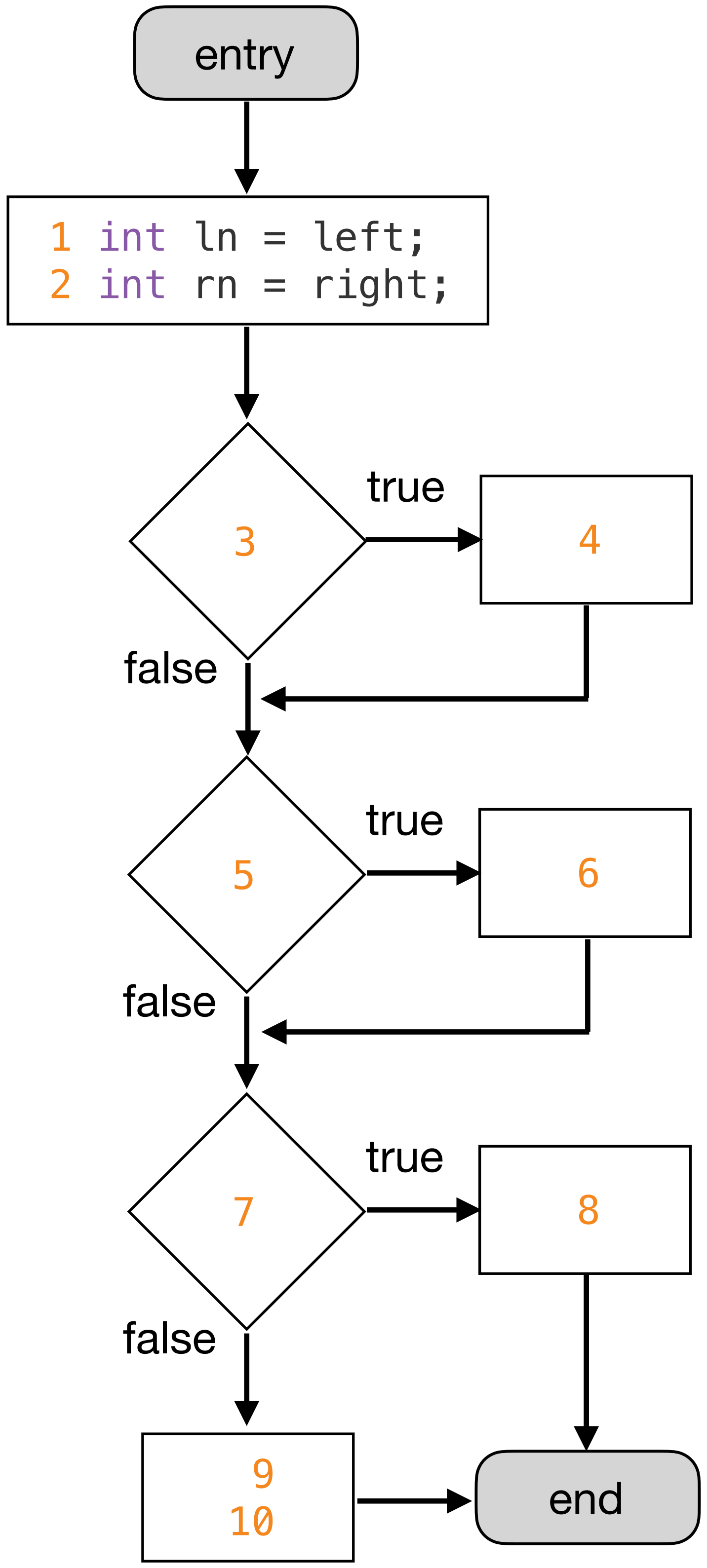
2. Compute CC as, E - N + P

```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21)
4         ln = 0;
5       if (rn > 21)
6         rn = 0;
7       if (ln > rn)
8         return rn;
9       else
10        return ln;
    }
```

# Edges = 12

# Nodes = 8 “normal” nodes + 1 entry node + 1 exit node

CC = 12 - 10 + 1 = 3 independent paths, but which ones?



CC, example

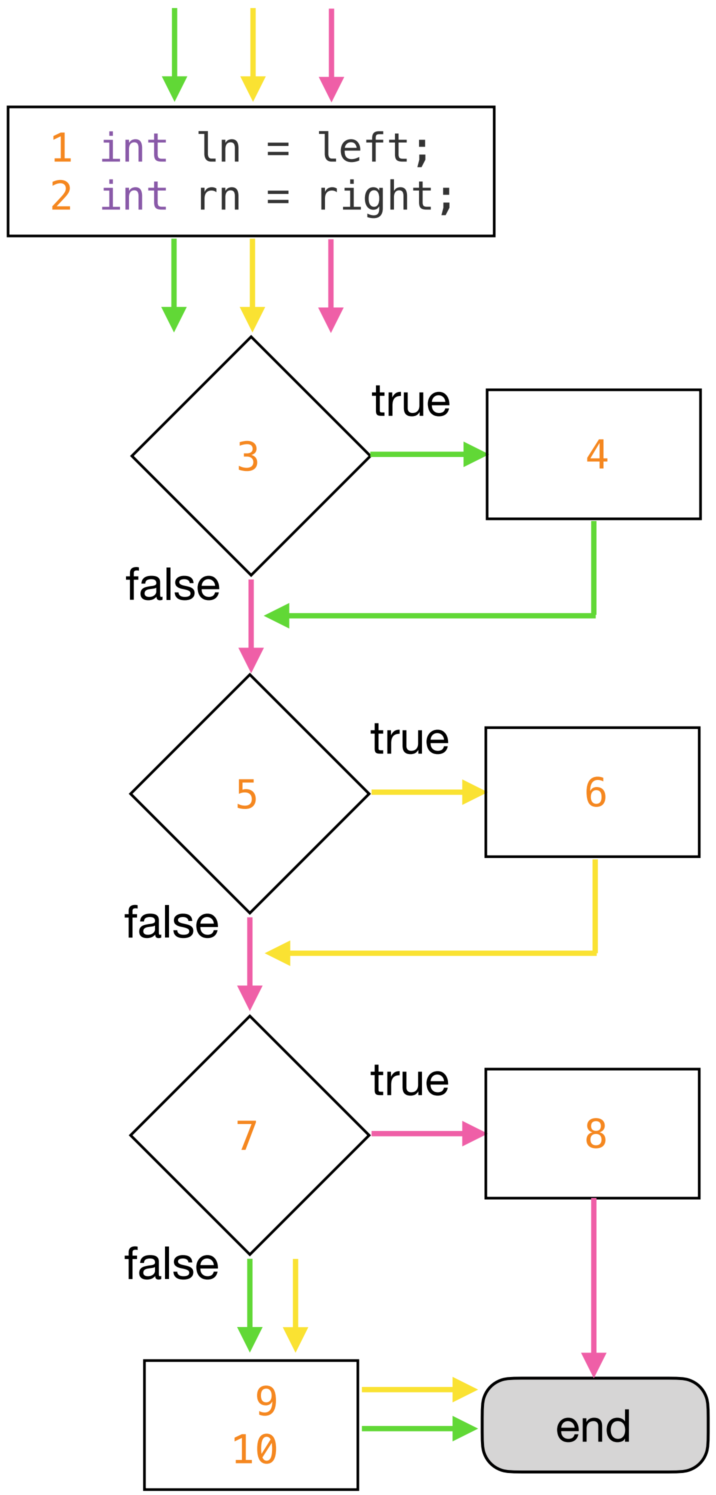
3. Identify the independent paths. Recall that a linearly independent path is defined as a path that has at least one edge which has not been traversed by any other path.

```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21)
4         ln = 0;
5       if (rn > 21)
6         rn = 0;
7       if (ln > rn)
8         return rn;
9       else
10        return ln;
    }
```

path 1: 1, 2, 3, **4**, 5, 7, 9, 10

path 2: 1, 2, 3, 5, **6**, 7, 9, 10

path 3: 1, 2, 3, 5, 7, **8**



How to compute CC?

1. Construct the control-flow graph with nodes and edges from the source code.

2. Compute CC (lower CC == better code)

   - of a structured function, e.g., a function with a single exit point, aka return statement, as

$$E - N + 2P$$

   - of a non-structured function, e.g., a function with more than on exit point, in each exit point is connected back to the entry point, as

$$E - N + P$$

3. Identify the independent paths.

4. Derive the tests.

CC, example

4. Derive the tests. Tip: the number of tests is basically the cyclomatic complexity value of the program.

```java
public class BlackJack {

  public int play(int left, int right) {
    int ln = left;
    int rn = right;
    if (ln > 21)
     ln = 0;
    if (rn > 21)
     rn = 0;
    if (ln > rn)
     return rn;
    else
     return ln;
  }

}
```

Cyclomatic Complexity, 👍 and 👎

Advantages 👍

- It is easy to apply.

- It is able to guide the testing process, i.e., it helps developers and testers to determine independent path executions.

- Developers can assure that all the paths have been tested at least once.

- Helps developers and testers to focus more on the uncovered paths.

Disadvantages 👎

- It is the measure of the programs' control complexity and not the data complexity.

Tools

- EclEmma: Java Code Coverage for Eclipse
  https://www.eclemma.org
- JaCoCo
  https://www.jacoco.org/jacoco
- Cobertura: A code coverage utility for Java
  http://cobertura.github.io/cobertura/
- Code2flow: automatic generation of diagrams from source code
  https://app.code2flow.com
- MetricsTree: an IntelliJ IDEA plugin to compute source code metrics as cyclomatic complexity, for example
  https://github.com/b333vv/metricstree
- GZoltar - Java Library for Automatic Debugging
  http://www.gzoltar.com
- Program repair
  https://program-repair.org

References

- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6.
- Chapter 5 of the Practical software testing a process-oriented approach. Ilene Burnstein, 2002.
- Chapter 12 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys.
- W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, Franz Wotawa; A survey on software fault localization, 2016.
- Thomas Durieux, Fernanda Madeiral, Matias Martinez, Rui Abreu; Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts, 2019.