

# Software Engineering Project

Morgan Ericsson



[morgan@cse.gu.se](mailto:morgan@cse.gu.se)



[@morganericsson](https://twitter.com/morganericsson)



[morganericsson](https://github.com/morganericsson)



UNIVERSITY OF  
GOTHENBURG

---

**CHALMERS**

# My Weekend (AKA Research Time!)

- Current project requires a Java parser (i.e., a tool that can interpret Java source code)
- Simple enough, there are plenty of existing parsers (for example, in Eclipse), so adapt
  - set of rules that describe the language (250ish)

[illegible]

# Does it Work?

- All the rules are written (based on the spec), so how do we know if it works?
- Test it on a simple program!

```
public class Hello {  
    // method main(): ALWAYS the APPLICATION entry point  
    public static void main (String[] args) {  
        System.out.println ("Hello World!");  
    }  
}
```

# Does it Work?

- All we know is whether it works with the given input or not?
  - no means more work
  - what about yes?
- Evaluate what yes means...

# Does it Work?

- Further testing
  - tested on 60 selected Java programs
  - 4 worked
- Iterative fixes finally resulted in all 60 working
  - more testing?

# Yes!

- Tried to parse 20,000 Java programs
  - 27 failed to parse
- A problem?
  - depends, all 27 relied on deprecated features in Java
- Does it work?
  - depends
  - 5 = a parses

# Some Truths About Testing

- It is impossible to completely test a program
- Testing cannot show that a program does not have any errors
- All errors will not be fixed
- It can be difficult to determine if something is an error or not
- Requirements and specifications will change

# Not New

*“an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders”,*  
**Ada Lovelace** (1815-1852)

*“It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that 'Bugs' — as such little faults and difficulties are called — show themselves and months of intense watching, study and labor are requisite...”,*  
**Thomas Edison** (1847-1931)



# Why Should We Care?

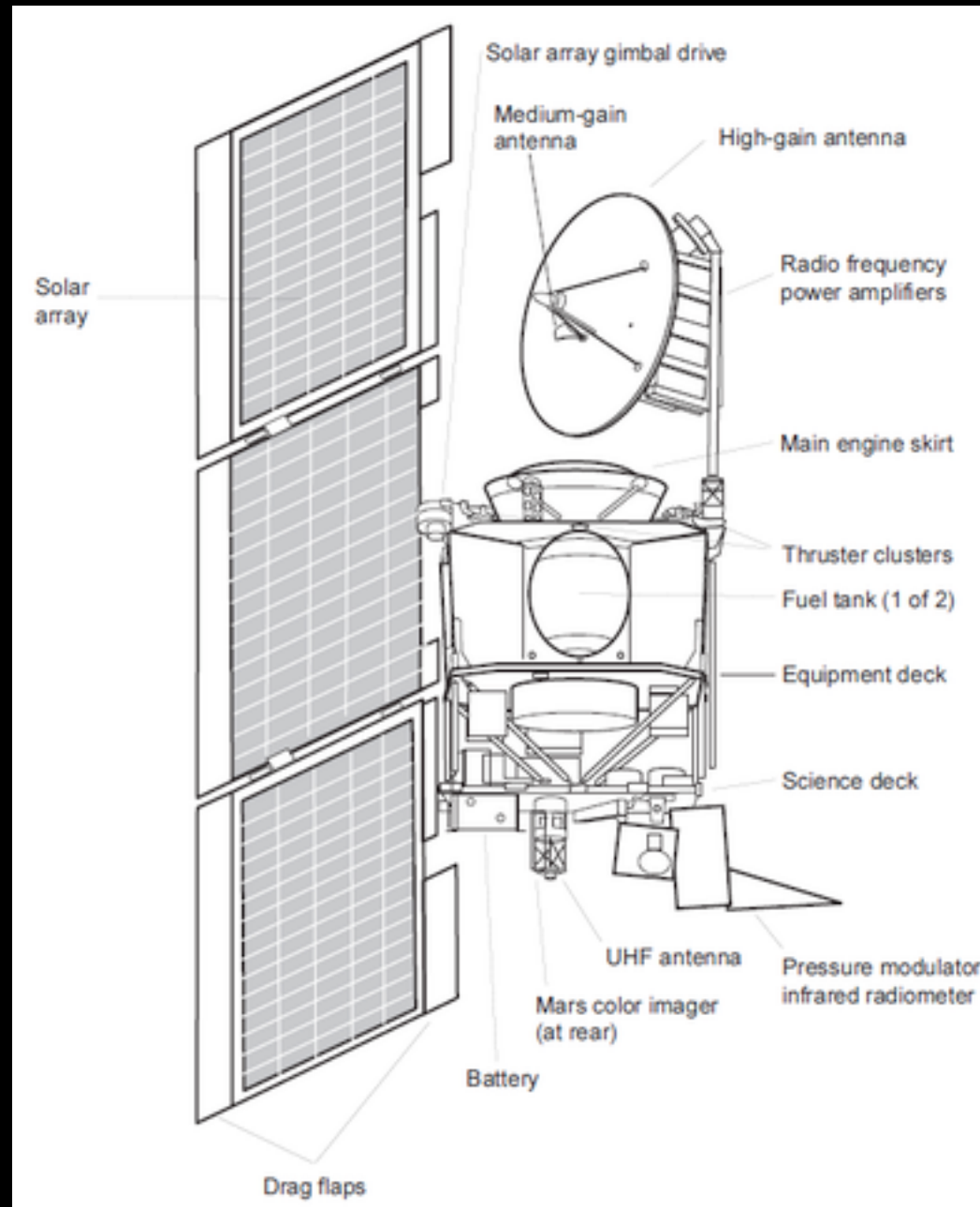
- NIST (National Institute for Standards and Technology) determined that software errors cost **\$60B / year** (2002)
- Industry average, **15-50 errors / KLOC** (release)
- Microsoft, 0.5 errors / KLOC (release)
  - Windows Vista, **50,000 KLOC**

# Pentium FDIV



- Certain inputs result in the **wrong result / precision**
- On average **1 error per 9 billion operations** (random data)
- **5** out of **1066** rows in a table where **wrong** due to incorrect loop conditions
- intel spent about **\$500M** (estimate) to replace CPUs

# Mars Climate Orbiter



- Navigation errors resulted in an **approach** at **too low orbit**, and was **destroyed** on entry
- Two systems used **different units**
  - Pounds-seconds
  - Newton-seconds
- Approx. \$330M

# Ariane 5 (flight 501)



- Exploded 37s after launch
- Software from Ariane 4 was **reused**
  - **Changed requirements**
- Different trajectory
- 64bit numbers converted to 16bit (**overflow exception**)
- But, **exceptions** were **not handled** for **performance** reasons...

# Is a Program Correct?

- **Prove** it
- **Show** that a **model** (that is presumably implemented by the program) has certain properties
- **Test** if the program has certain **properties**

# Prove Correctness

- Use **formal methods** (maths) to **prove** that a program is correct
- Quite difficult and involved, even for small programs
- Generally **not practical**, even for safety-critical programs

# Test a Model

- Extract an **abstract model** from a program and **show** that the model is **correct**
- Shows **properties** for **all executions**
- In practice,
  - difficult to show **non-trivial properties**
  - for programs with **non-trivial structure**

# Example

- seL4, an operating system kernel
  - 7,500 lines of C code
  - 200,000 lines of proof
  - 25-30 man years (research project)
- Due to project contributions, they claim a similar proof can be done in about 10 man years



# So, Testing...

- “**Execute**” the program and see if it behaves correctly
  - **static**: read, inspect, analyse
  - **dynamic**: run
- Realistic and **always necessary**
- *“Beware of bugs in the above code; I have only proved it correct, not tried it”, **Donald Knuth***

# Static Testing

- Read the code and look for (common) errors
  - are all variables initialized?
  - are loop conditions correct?
  - are array indices within bounds?
  - ...
- Tools can help

# Example

```
public int countPositive(int[] x) {  
    int count;  
    for (int i=1; i<=x.length; i++) {  
        if (x[i]>=0) count++;  
    }  
    return count;  
}
```

# Dynamic Testing

- Run the program (incl. compilations) and see what happens
  - Test different inputs, check outputs
  - One run tests one set of inputs!

# Example

```
public int countPositive(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]>=0) count++;  
    }  
    return count;  
}
```

```
a=countPositive([1,2,3,4]);      //a==3  
b=countPositive([-1,2,-3,4]);    //b==2  
c=countPositive([-1,-2,-3,-4]);  //c==0
```

# Verification/Validation

- Often used incorrectly
  - **Verification**, check according to **specification**
  - **Validation**, check according to initial **requirements**
- Verification: You implemented it **correctly**
- Validation: You implemented the **correct thing**

# Test Cases and Suites

- A test case represents an **execution** of a program
  - input and expected output
  - e.g.,  $([-1, 2, -3, 4], 2)$
- A **test suite** is a set of test **cases**

# Black Box Testing

- Consider the program that is tested as a black box (**impenetrable**)
- Testing does not consider source code or internal properties
- Provides **input**, checks **output**
- Useful to test **systems** or when there is **no available source code**



# White Box Testing

- Look inside the box
- Use source code to create test cases

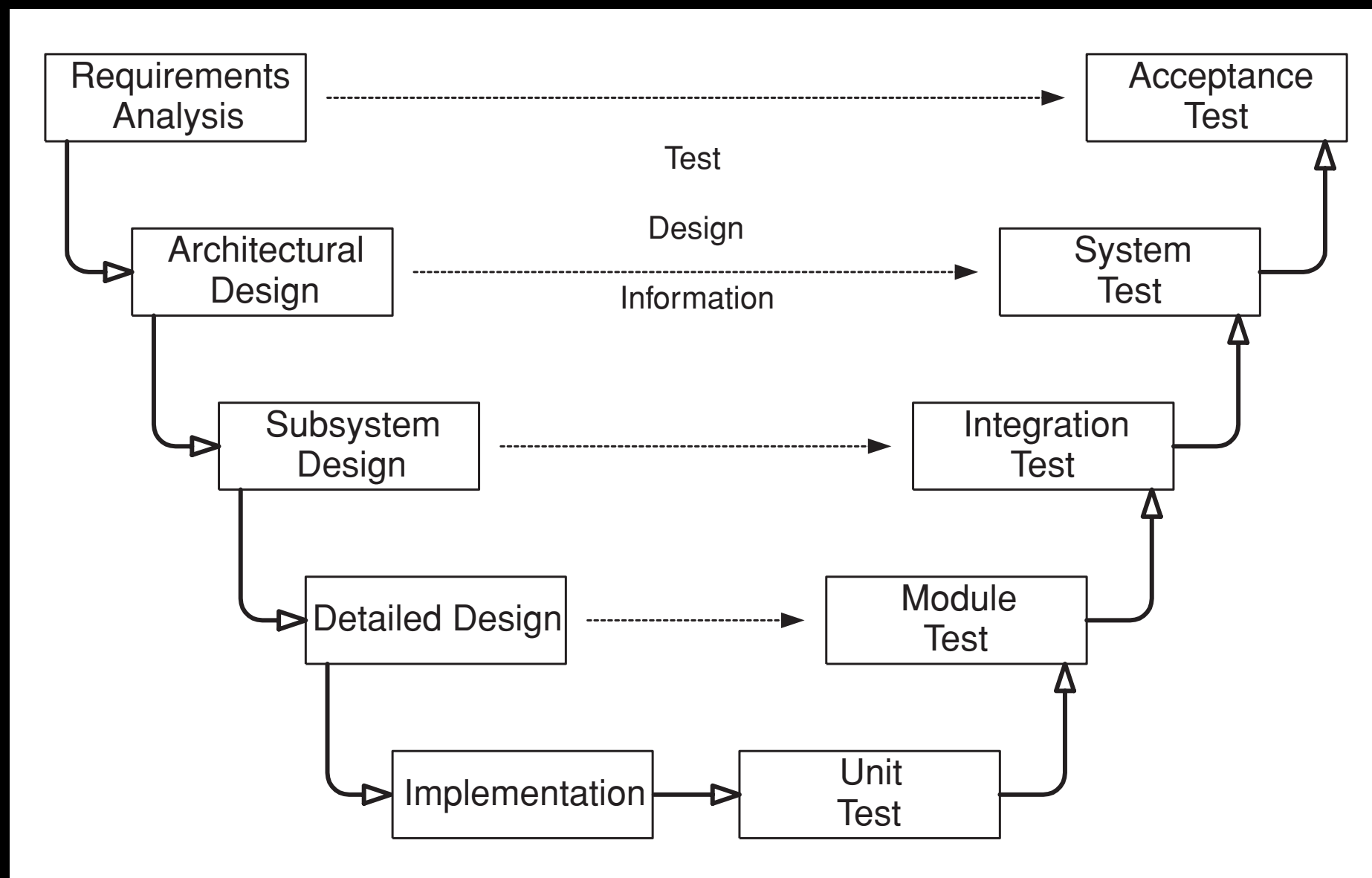
# Coverage

- **Measures** what a test suites tests, based on structural properties, e.g.
  - which **lines** of code are executed
  - which boolean **expressions** are evaluated to both **true** and **false** during a test
  - which **control flows** are tested?

# Coverage

- Often used as a **quality measure** of the testing / test suite
- **higher** coverage is better
- **not** a very **good** measure, but...
- easy to understand and reason about (when white box testing)

# When / What Do We Test?



# Pass or Fail?

- **Test-to-Pass**, tests if the program **works**
  - simple test cases
  - that should work
- **Test-to-Fail**, tries to **break** the program
  - evil test cases
  - that might not work

# Initial Example

- Black box testing
  - I had the source, but did not consider it
- **First** 60 cases, test to **pass**
- **Next** 20,000 cases, test to **fail** (approx.)
- Why start with test to pass?

# A Good Approach?

- 60 + 20,000 randomly selected Java programs
  - so, in a sense, double black box
- Reasonable?
  - quite slow (1 test / second = 5+ hours)
  - all 27 problems were “identical” (“so 1 told me something, 26 did not”)

# A Better Approach?

- Consider the program a **function**
  - with a **domain** and a **range**
- **Partition** the domain into blocks with similar properties
  - test a block
  - test between blocks (edges)

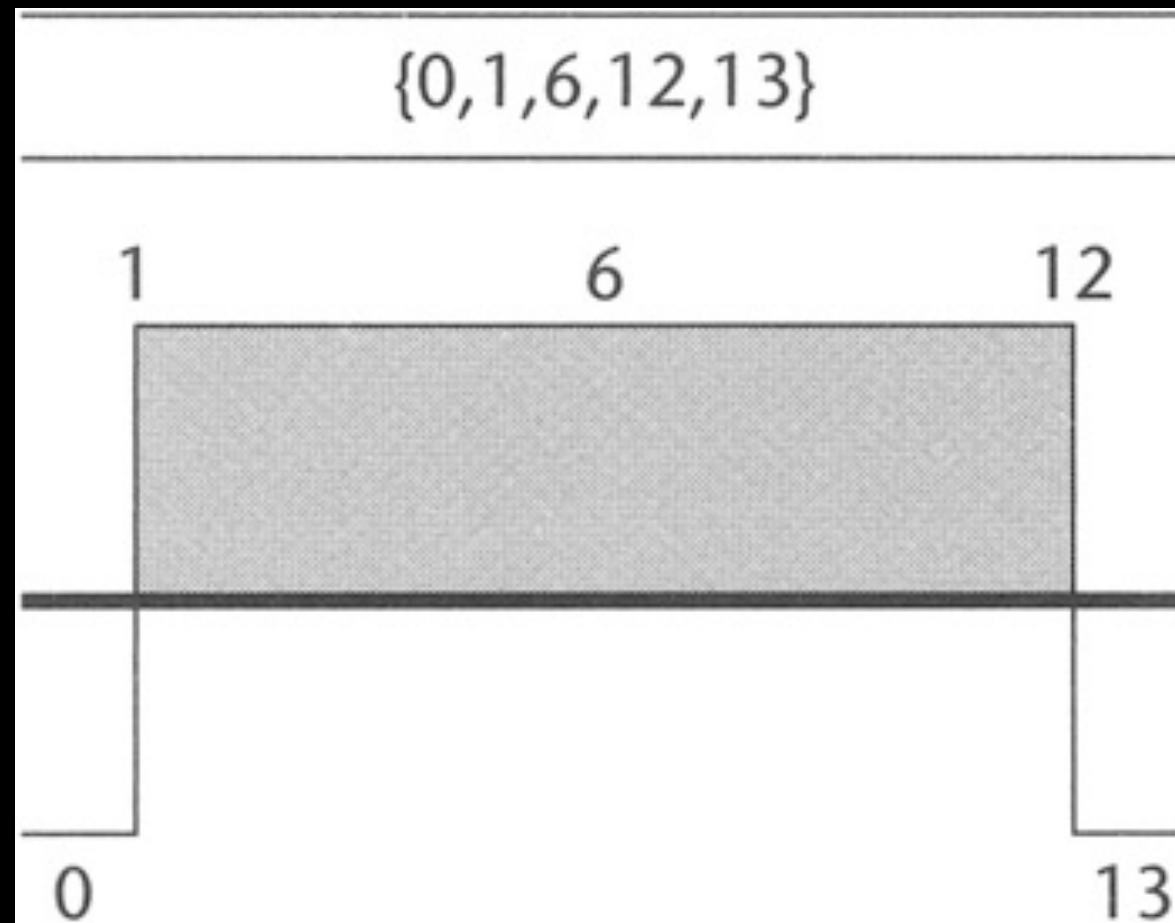


# Example

$P_1: x < y$	$P_2: x = y$	$P_3: x > y$
$\langle (1, 2), 2 \rangle$	$\langle (1, 1), 1 \rangle$	$\langle (3, 2), 3 \rangle$
$\langle (3, 9), 9 \rangle$	$\langle (3, 3), 3 \rangle$	$\langle (3, 2), 3 \rangle$
...	...	...

Testing a function that determines  
the larger of two integers

# Example

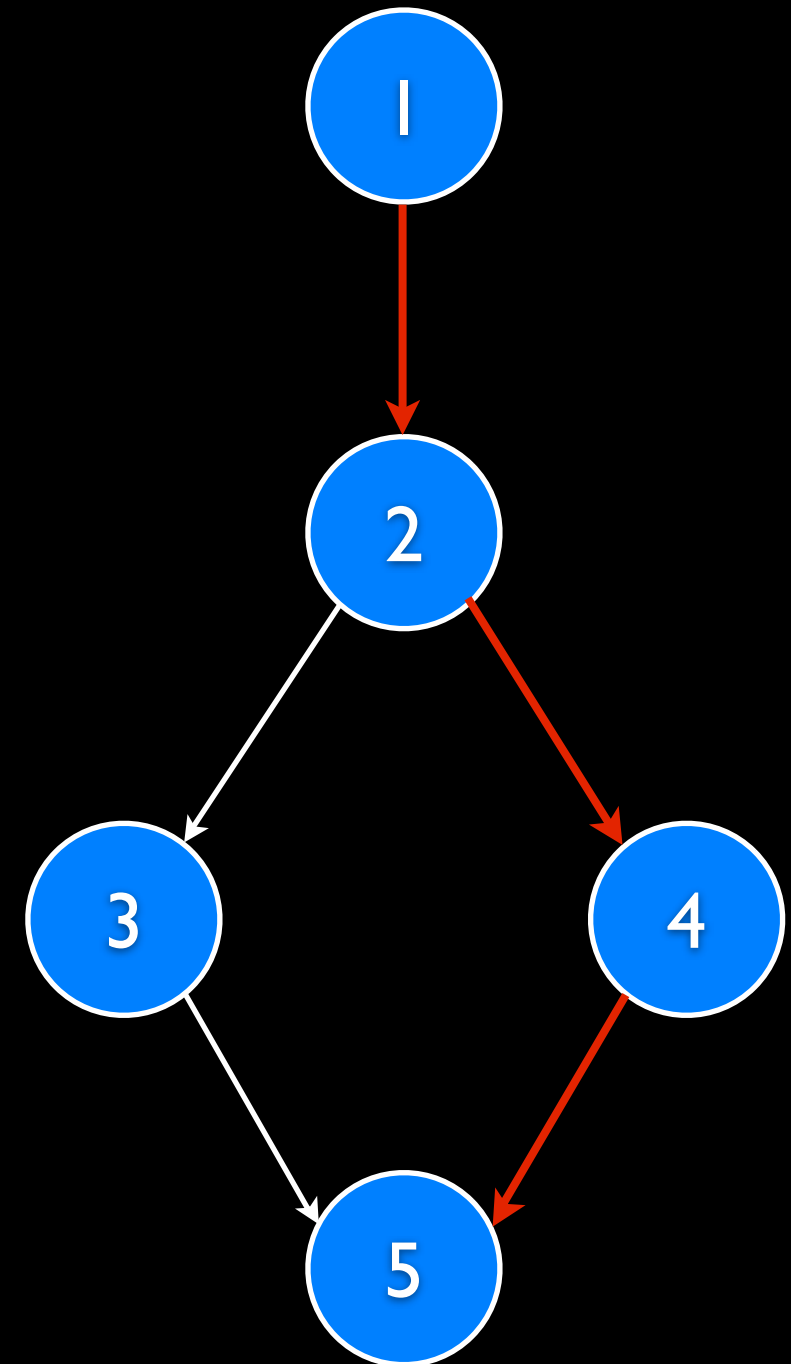


# White Box

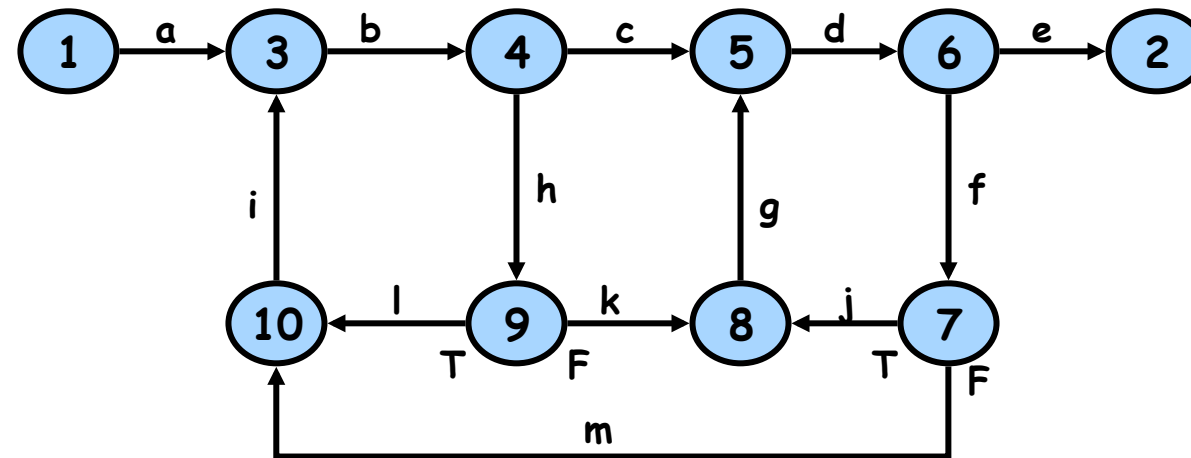
- Find **paths** through the program ...
- ... and make sure you **cover** these
- **Statement** coverage
- **Branch** coverage

# Example

```
int max(int x, int y) {  
1:  int max;  
2:  if(x>y)  
3:    max=x;  
   else  
4:    max=y  
5:  return max;  
}
```



# Example



PATHS	DECISIONS				PROCESS LINKS												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
<i>abcde</i>	<i>T</i>	<i>T</i>			*	*	*	*	*								
<i>abhkgde</i>	<i>F</i>	<i>T</i>		<i>F</i>	*	*		*	*		*	*			*		
<i>abhlibcde</i>	<i>TF</i>	<i>T</i>		<i>T</i>	*	*	*	*	*			*	*			*	
<i>abcdfjgde</i>	<i>T</i>	<i>TF</i>	<i>T</i>		*	*	*	*	*	*	*			*			
<i>abcdfmibcde</i>	<i>T</i>	<i>TF</i>	<i>F</i>		*	*	*	*	*	*		*					*

# Common sense

*“The more we learn about testing, the more we realize that statement and branch coverage are minimum floors below which we dare not fall, rather than ceilings to which we should aspire.”* B. Beizer

# Example: ABS

```
/* ABS
```

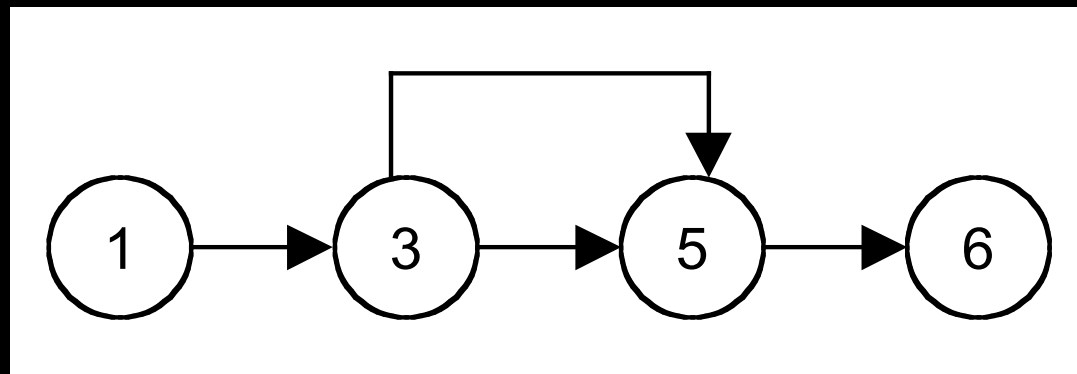
This program function returns the absolute value of the integer passed to the function as a parameter.

INPUT: An integer.

OUTPUT: The absolute value if the input integer.

```
*/
```

```
1  int ABS(int x)
2  {
3      if(x < 0)
4          x = -x;
5      return x;
6  }
```



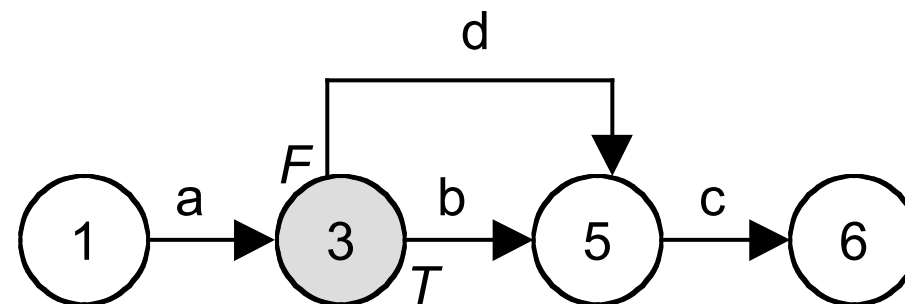
# Example: ABS

- Possible to **completely** test ABS
- Since there is a **maximum** integer
- However, **not practical** since since we often use either 32-bit or 64-bit integers
  - so, best case,  $2^{32}$  test cases



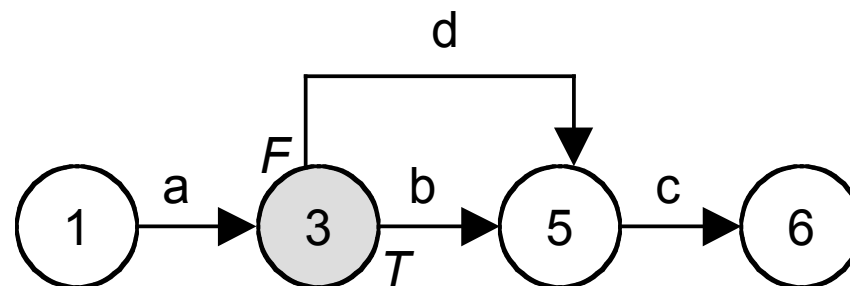
# Statement Coverage

PATHS	PROCESS LINKS				TEST CASES	
	a	b	c	d	INPUT	OUTPUT
abc	✓	✓	✓		A Negative Integer, x	-x
adc	✓		✓	✓	A Positive Integer, x	x



# Branch Coverage

PATHS	DECISIONS	TEST CASES	
		INPUT	OUTPUT
abc	T	A Negative Integer, x	-x
adc	F	A Positive Integer, x	x



# Unit Testing

*“If code has no **automated test cases** to show it works, then the **assumption** must be that it **doesn't**”*

# Unit Testing

- Parts of the source code (s.k. **units**) are **tested**
- A unit is the **smallest testable entity**
  - often **procedure / method**
- White box
- Often written by the **developer** (TDD)
- Often **automated**

# Benefits

- Makes it easier to **change** the code
  - test checks so everything still works
- Supports **integration** testing
  - shows that each unit works
  - (but not the same thing)

# Benefits (more later)

- Living documentation
  - **shows** how the **API/methods** should be used
  - illustrates cases that are **important**
- Design
  - if (and they should) the test cases are written **before implementation**, they can **guide** the implementation

# Form

- Unit tests call or executes part of the source code
- The result is validated by asserting facts
  - e.g. `assertNotEqual("apple", "orange")`

# Example

```
public void testAbs10() {  
    assertEquals(10.0, Operations.myAbs(10.0));  
}
```

```
public void testAbsn10() {  
    assertEquals(10.0, Operations.myAbs(-10.0));  
}
```

```
public void testSqrt25() {  
    assertEquals(Math.round(5.0),  
        Math.round(Operations.mySqrt(25.0)));  
}
```



# Acceptance Testing (XP)

- Tests a **user story**
  - from a “**customer**” perspective
- The customer can **accept** or **reject**
- Testing should be at least modeled by the customer
- AKA **functional** testing or **customer** testing