System Specification and Refinement (EECS 3342)

In EECS 3342, you

- construct high level, abstract mathematical models of a system (consisting of both the system and its environment) amenable to formal reasoning.

In this course, you

- work with models that are automatically generated from Java bytecode.

System Specification and Refinement (EECS 3342)

In EECS 3342, you

- apply set theory and predicate logic to express properties.

In this course, you

- implement listeners in Java to check properties.

System Specification and Refinement (EECS 3342)

In EECS 3342, you

- use practical tools for constructing and reasoning about the models.

In this course, you

- use, modify and extend practical tools for checking properties of the models.

In EECS 3342, you

- use a theorem prover (which often needs input from you).

In this course, you

- use a model checker (which needs no input from you).

System Specification and Refinement (EECS 3342)

In EECS 3342, you

- focus on designing code that is correct by construction.

In this course, you

- focus on finding bugs in code.

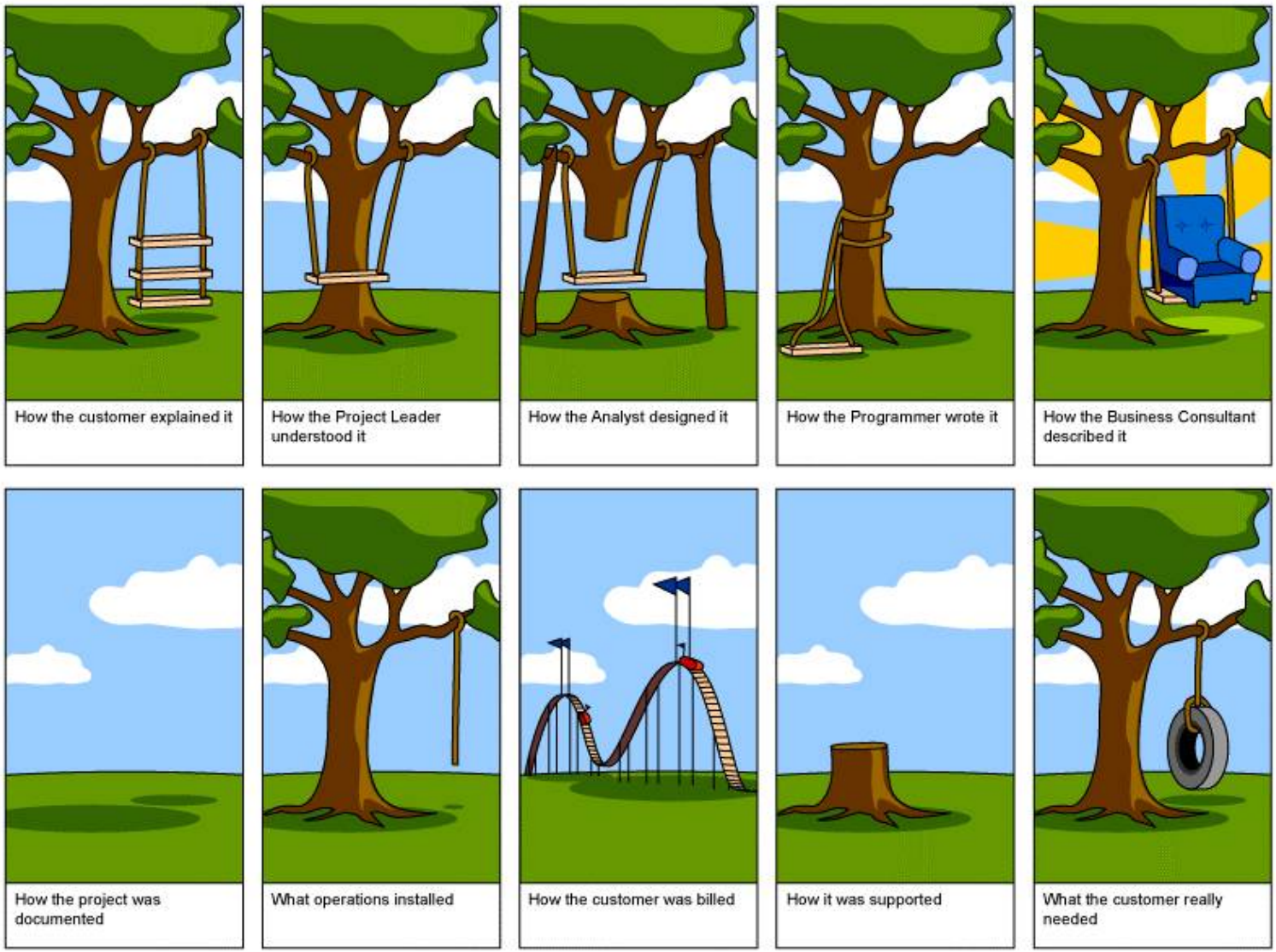
Bugs are everywhere

EECS 4315

www.eecs.yorku.ca/course/4315/

What is verification?

"Have you made what you were trying to make?"

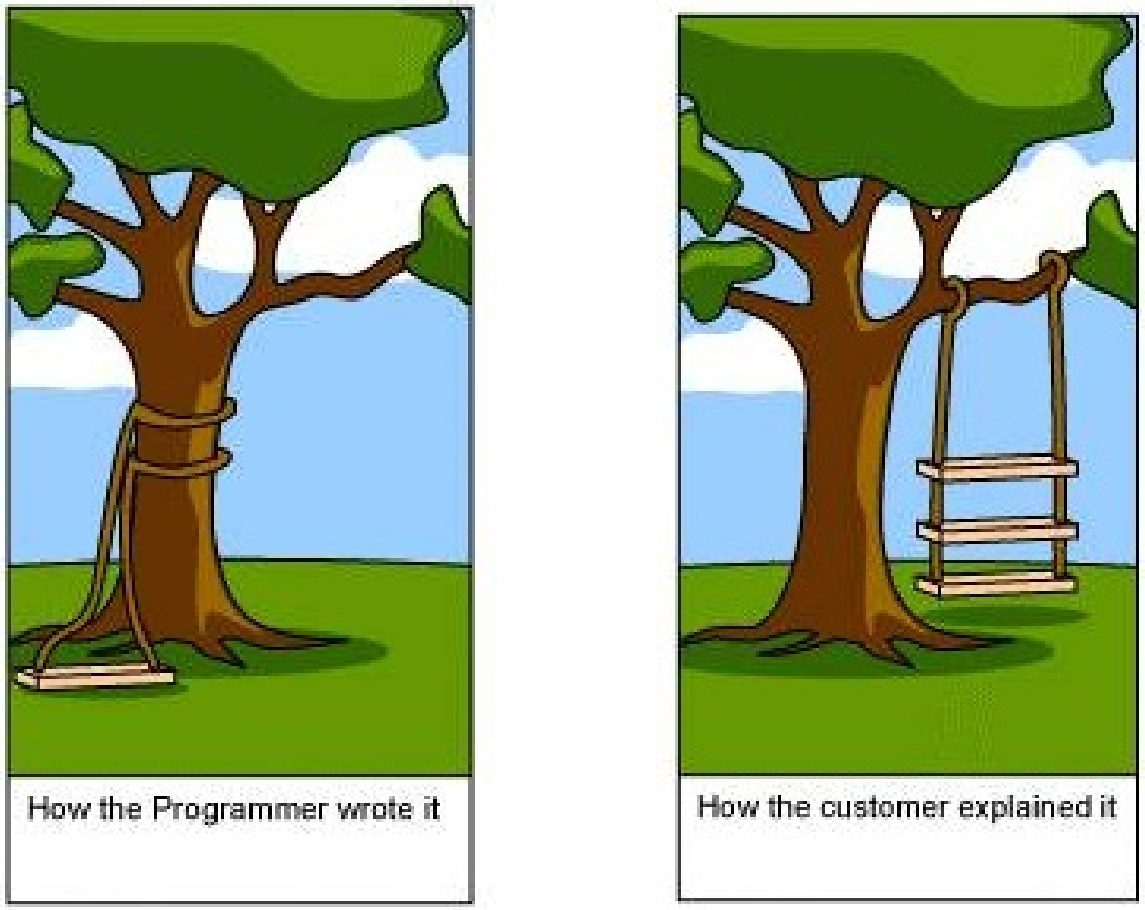


Source: Paragon Innovations

What is Verification?

"Have you made what you were trying to make?"

"Does the code satisfy (all the properties of) its specification?"



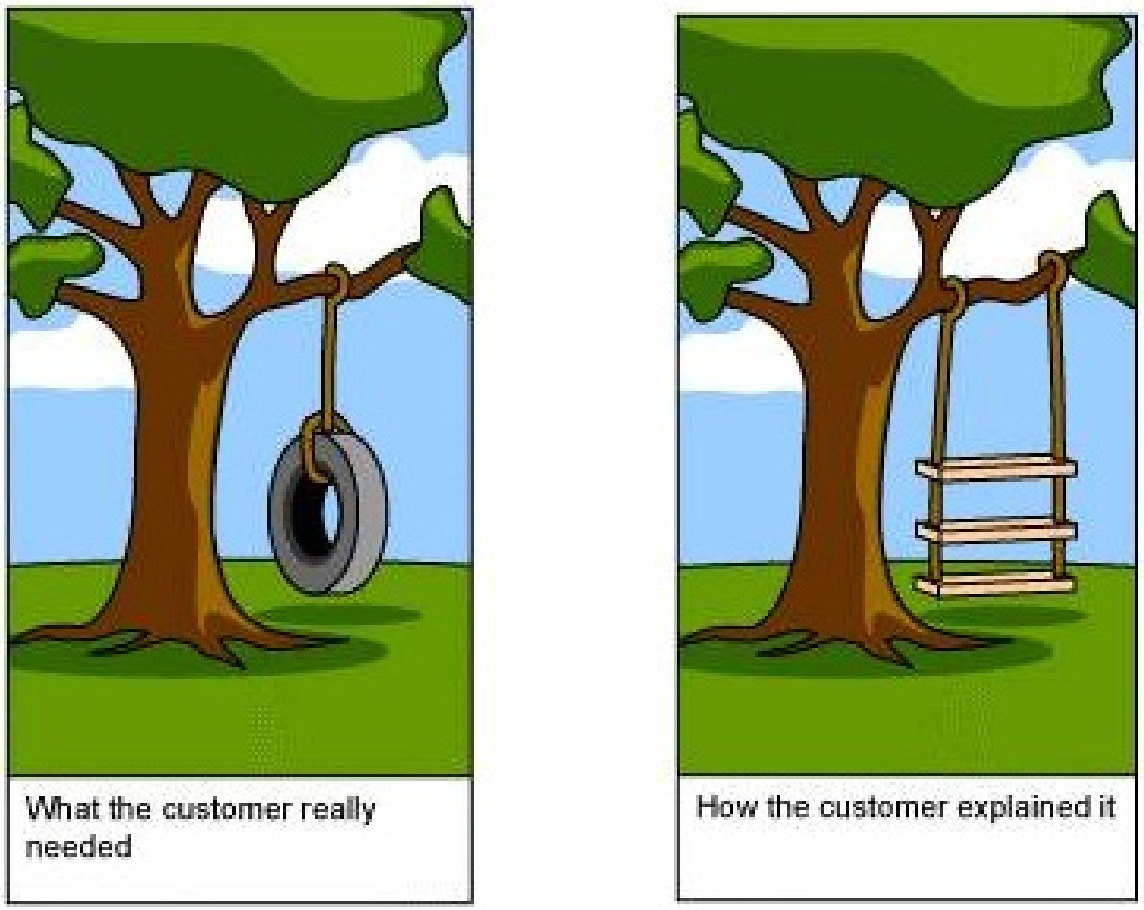
Source: Paragon Innovations

In contrast to . . .

"Have you made the right thing?"

Is the specification of the system correct?

which is also known as validation.



Source: Paragon Innovations

Why do we verify?

Bugs are everywhere.

Source: Bruce Campbell

Classic bug

1968 Brazilian Beetle

Source: Dan Palatnik

Classic bug

"A clear example of the risks of poor programming and verification techniques is the tragic story of the Therac-25 — one in a series of radiation therapy machines developed and sold over a number of years by Atomic Energy Canada Limited (AECL). As a direct result of inadequate programming techniques and verification techniques, at least six patients received massive radiation overdoses which caused great pain and suffering and from which three died."

Peter Roosen-Runge. Software Verification Tools.

Source: unknown

Classic bug

A computer malfunction at Bank of New York brought the Treasury bond market's deliveries and payments systems to a near standstill for almost 28 hours . . . it seems that the primary error occurred in a messaging system which buffered messages going in and out of the bank. The actual error was an overflow in a counter which was only 16 bits wide, instead of the usual 32. This caused a message database to become corrupted. The programmers and operators, working under tremendous pressure to solve the problem quickly, accidentally copied the corrupt copy of the database over the backup, instead of the other way around."

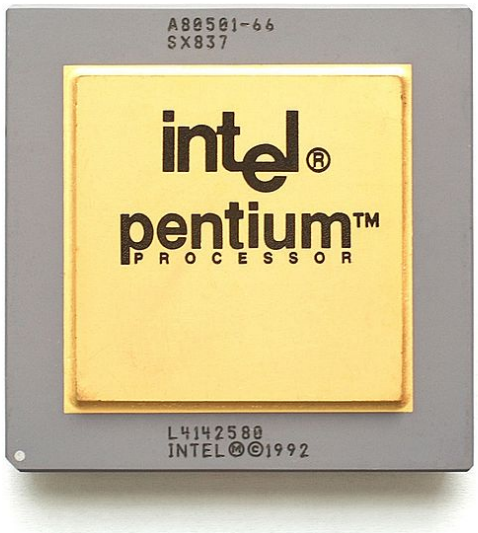
Wall Street Journal, November 25, 1985.

Source: unknown

Classic bug

"To correct an anomaly that caused inaccurate results on some high-precision calculations, Intel Corp. last week confirmed that it had updated the floating-point unit (FPU) in the Pentium microprocessor. The company said that the glitch was discovered midyear and was fixed with a mask change in recent silicon. "This was a very rare condition that happened once every 9 to 10 billion operand pairs" said Steve Smith, a Pentium engineering manager at Intel."

EE Times, November 7, 1994.

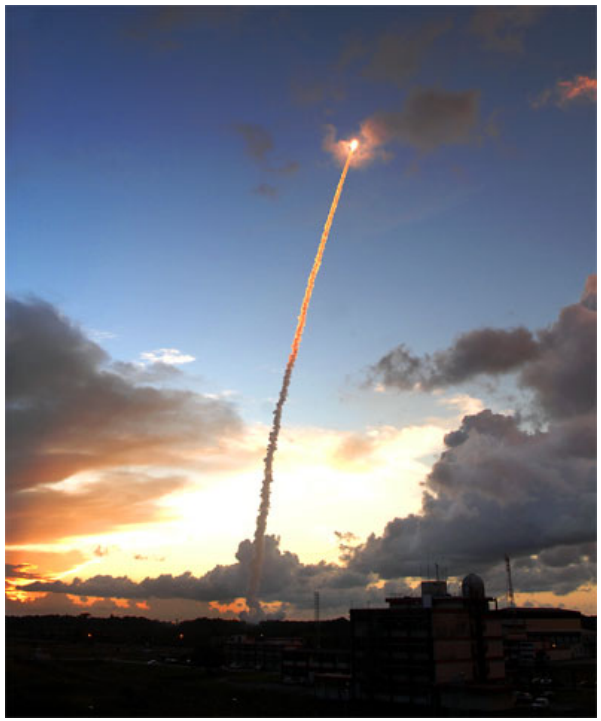


Source: Konstantin Lanzet

Classic bug

"On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 meters, the launcher veered off its flight path, broke up and exploded. . . . The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception. . . . . The data conversion instructions (in Ada code) were not protected from causing an operand error, although other conversions of comparable variables in the same place in the code were protected."

Report of the Ariane Inquiry Board



Source: unknown

Bug of the 21st century

2012 Beetle

Source: unknown

Bug of the 21st century

The Toronto skyline

Source: unknown

Bug of the 21st century

The Toronto skyline on August 14, 2003

Source: unknown

Bug of the 21st century

The first known death caused by a self-driving car was disclosed by Tesla Motors on Thursday, a development that is sure to cause consumers to second-guess the trust they put in the booming autonomous vehicle industry. . . . Against a bright spring sky, the car's sensors system failed to distinguish a large white 18-wheel truck and trailer crossing the highway, Tesla said. The car attempted to drive full speed under the trailer, with the bottom of the trailer impacting the windshield of the Model S, Tesla said."

Danny Yadron and Dan Tynan, The Guardian, July 1, 2016

Source: Daily Mail

Bug of the 21st century

"The maneuvering characteristics augmentation system (MCAS) is a software system that is part of the Boeing 737 MAX flight control system. When it detects that the aircraft is operating in manual flight, with the flaps up, at a high angle of attack, it adjusts the horizontal stabilizer trim to add positive force feedback to the pilot. The activation logic of MCAS has been shown to be vulnerable to erroneous angle of attack data, as analyses have shown following the Lion Air Flight 610 and Ethiopian Airlines Flight 302 crashes. Flaws found in the MCAS implementation ..."

wikipedia.org.

Source: www.flickr.com/people/aceyyc

Bug of the 21st century

"The Knight Capital Group announced on Thursday that it lost $440 million when it sold all the stocks it accidentally bought Wednesday morning because a computer glitch. . . . The company said the problems happened because of new trading software that had been installed. The event was the latest to draw attention to the potentially destabilizing effect of the computerized trading that has increasingly dominated the nation's stock markets."

Nathaniel Popper, The New York Times, August 2, 2012.

Source: Brendan McDermid

Bug of the 21st century

https://www.youtube.com/watch?v=FZ1st1Vw2kY

Why do we verify?

Ask Jessie J!

Source: unknown

It's all about the money money money.

What's the price tag?

Bank of New York bug: $5 million
Pentium bug: $475 million
Ariane bug: $500 million
Blackout bug: $6 billion
Boeing 737 MAX bug: $10 billion

"The cost of software bugs to the U.S. economy is estimated at $60 billion per year."

National Institute of Standards and Technology, 2002

"Wages-only estimated cost of debugging: US $312 billion per year."

Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak and Tomer Katzenellenbogen, 2013

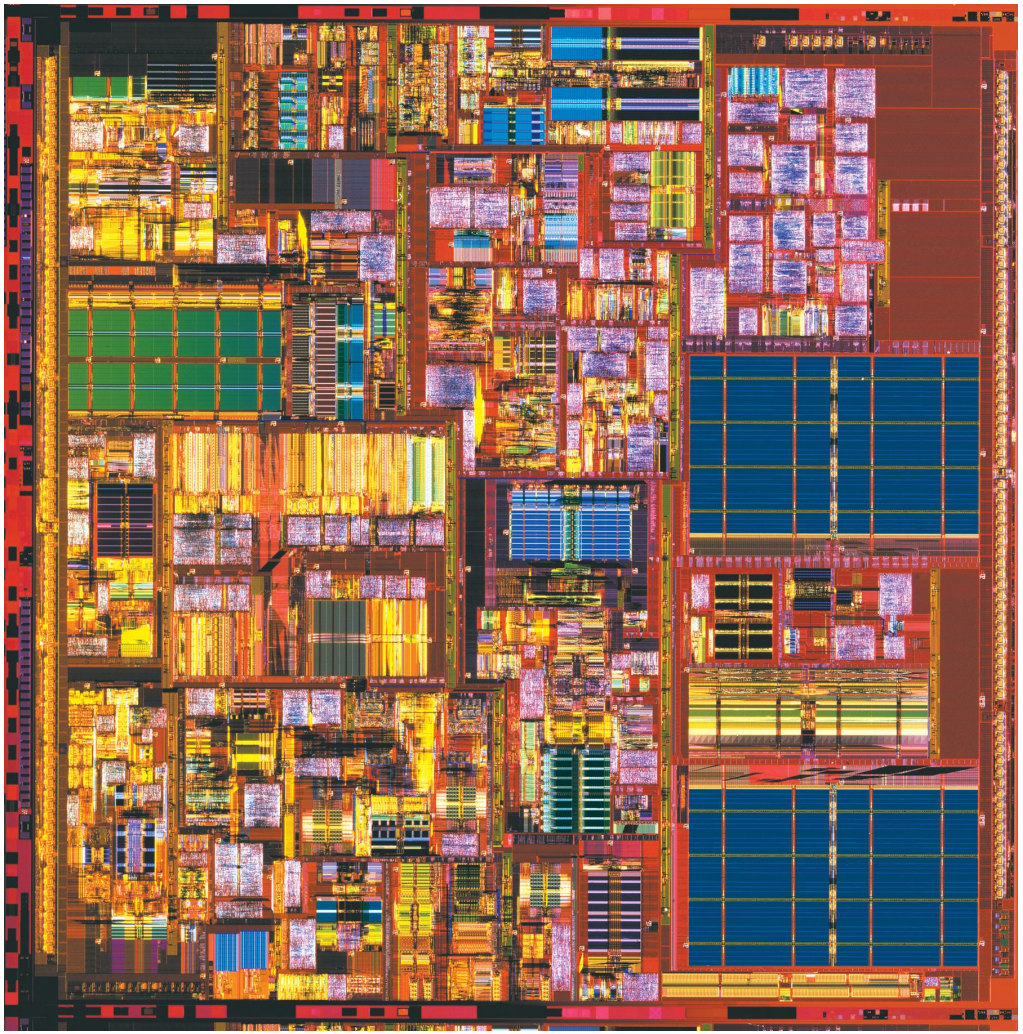
"The cost of poor quality software in the US in 2018 is approximately $2.84 trillion."

Herb Krasner, 2018

Why are bugs introduced?

Hardware and software systems are among the most complex artifacts ever produced by humans.

Pentium 4 microprocessor



Source: unknown

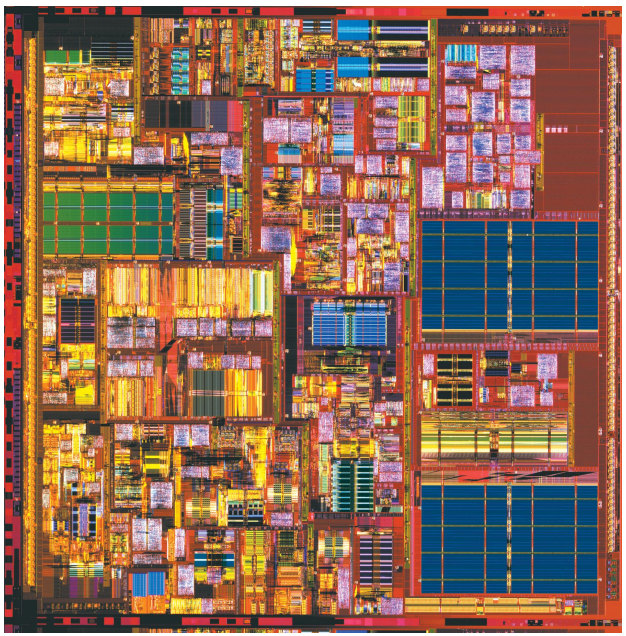
- transistors:
  55 million
- area:
  146 mm^2

If . . .

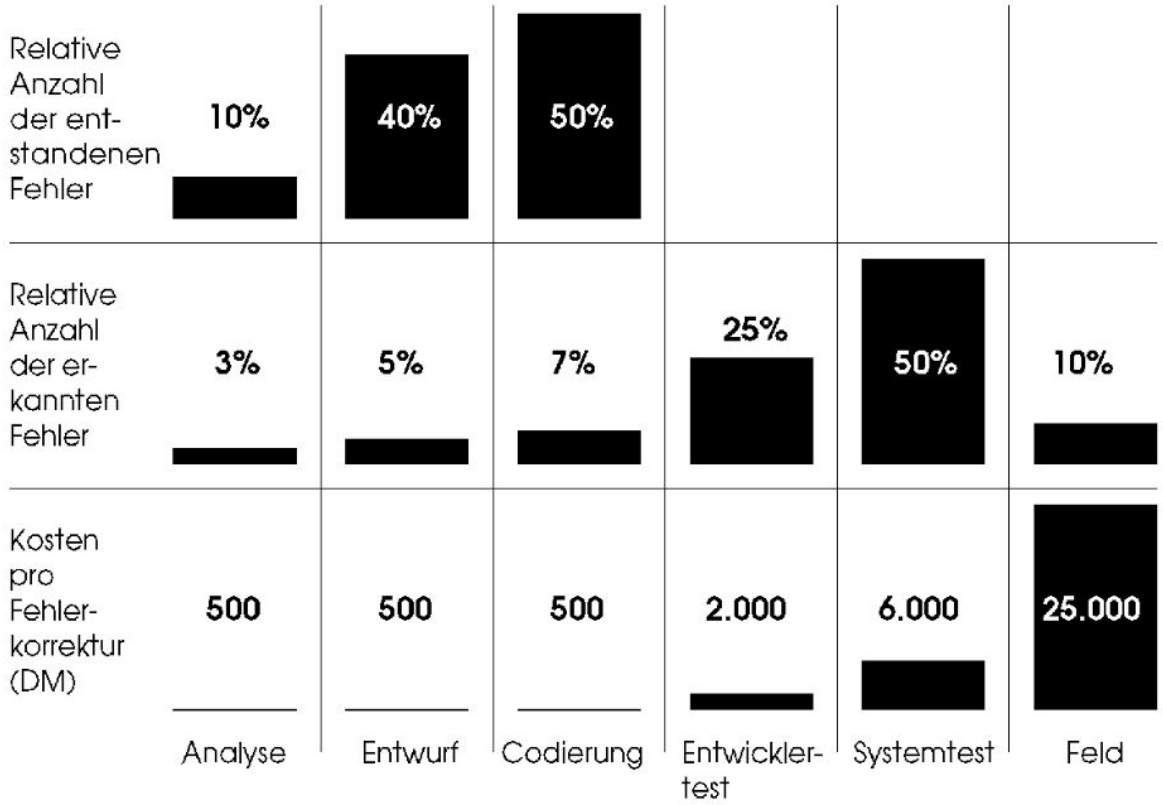
. . . the connections on a microprocessor were roads in the GTA, . . .

Area of microprocessor: 146 mm^2
Area of GTA: 7,124 km^2
Scale: 12 mm / 84 km $\approx$ 1 / 70,000,000

. . . then, since each connection is 0.13 μm wide, the roads in the GTA would be 3 feet wide, 3 feet apart and eight layers deep!

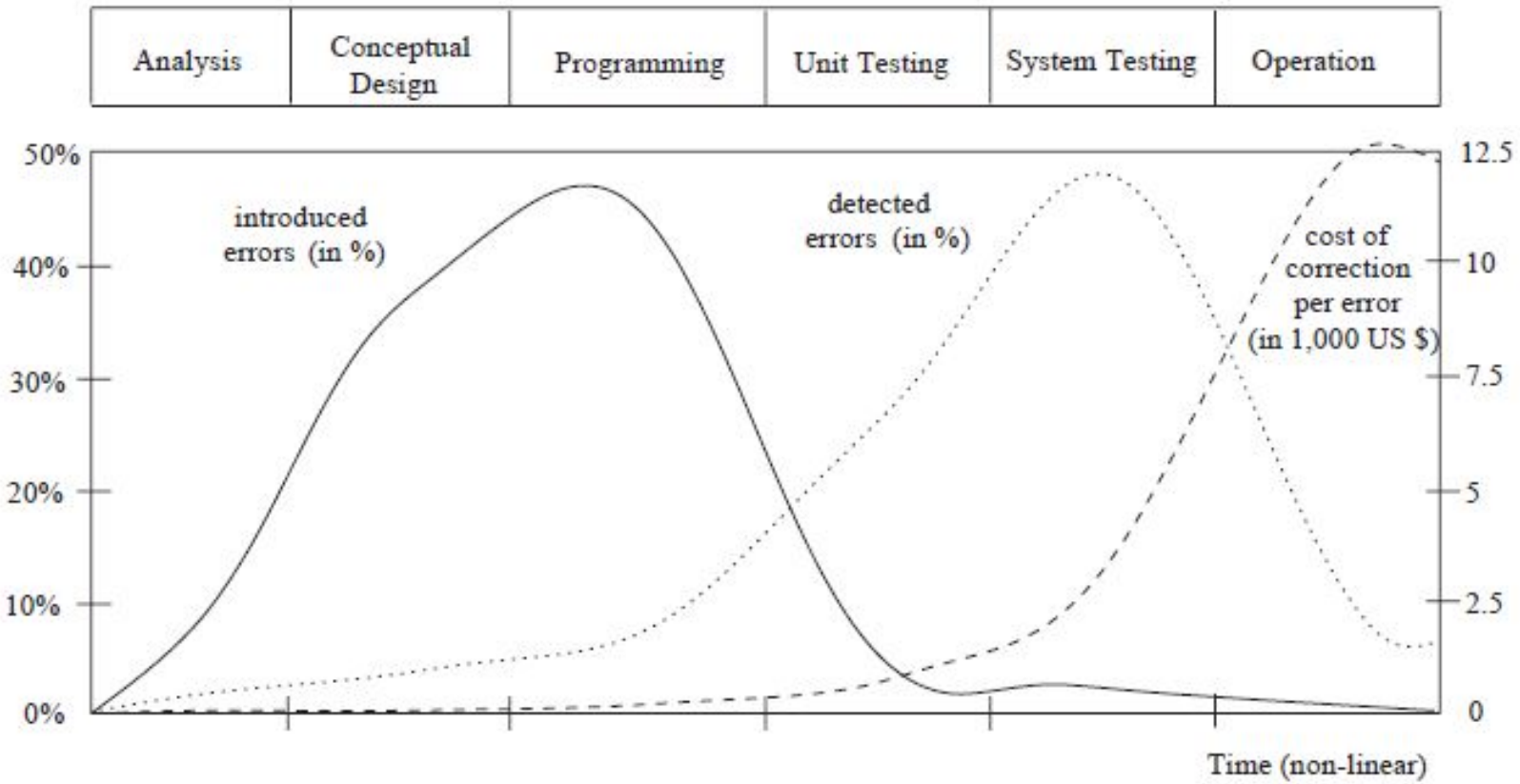


When are bugs introduced and detected?



Peter Liggesmeyer, Martin Rothfelder, Michael Rettelbach, and Thomas Ackermann. Qualitätssicherung Software-basiertertechnischer Systeme – Problembereiche und Lösungsansätze. *Informatik-Spektrum*, 21(5):249–258, October 1998

When are bugs introduced and detected?



Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press. 2008.

How are bugs detected?

- Peer review
- Simulation
- Testing
- Verification

Limitations of peer review

- Catches on average only 60% of the bugs.
- Is labour intensive (250 lines per hour).

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$
2. How many can we check in one second?

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$
2. How many can we check in one second?
   3×10^{9}

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$
2. How many can we check in one second?
   3×10^9
3. How many seconds does it take?

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$
2. How many can we check in one second?
   3×10^9
3. How many seconds does it take?
   $1.2 \times 10^{77} / 3 \times 10^9 = 4 \times 10^{67}$

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$
2. How many can we check in one second?
   3×10^9
3. How many seconds does it take?
   $1.2 \times 10^{77} / 3 \times 10^9 = 4 \times 10^{67}$
4. How many years is that?

Limitations of simulation

How long does it take to simulate a 128-bit multiplier on a 3 GHz machine?

1. How many cases need to be checked?
   $2^{128} \times 2^{128} = 2^{256} \approx 1.2 \times 10^{77}$
2. How many can we check in one second?
   3×10^9
3. How many seconds does it take?
   $1.2 \times 10^{77} / 3 \times 10^9 = 4 \times 10^{67}$
4. How many years is that?
   2×10^{59}

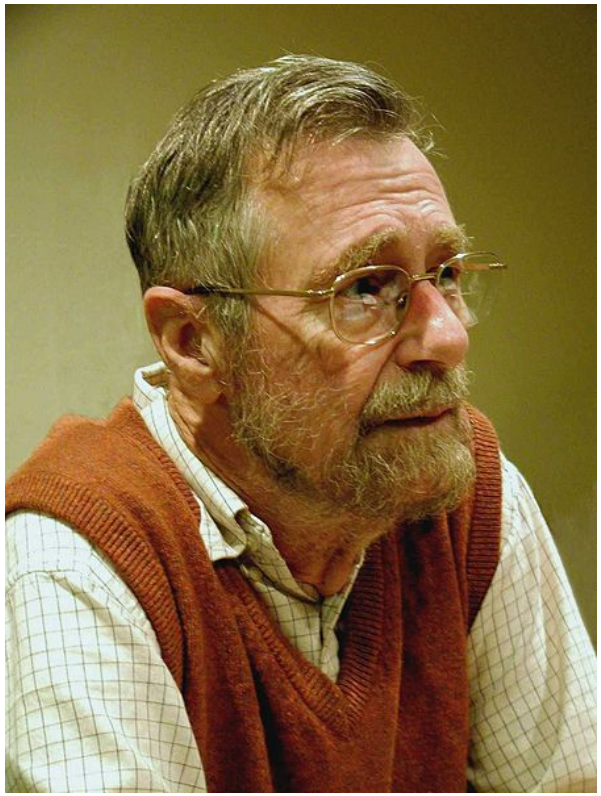
Limitations of testing

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

Edsger W. Dijkstra. Notes on structured programming. Report 70-WSK-03, Technological University Eindhoven, April 1970.

Edsger Wybe Dijkstra (1930–2002)

- Member of the Royal Netherlands Academy of Arts and Sciences (1971)
- Distinguished Fellow of the British Computer Society (1971)
- Recipient of the Turing Award (1972)
- Foreign Honorary Member of the American Academy of Arts and Sciences (1975)
- My scientific uncle (the supervisor of my supervisor was also Dijkstra's supervisor)



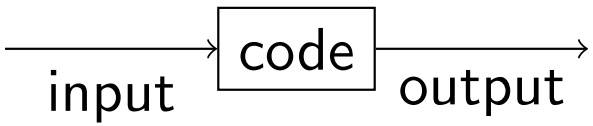
Source: Hamilton Richard

Bugs are everywhere

EECS 4315

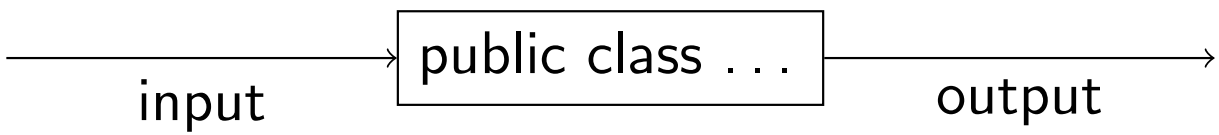
`www.eecs.yorku.ca/course/4315/`

How to test code?



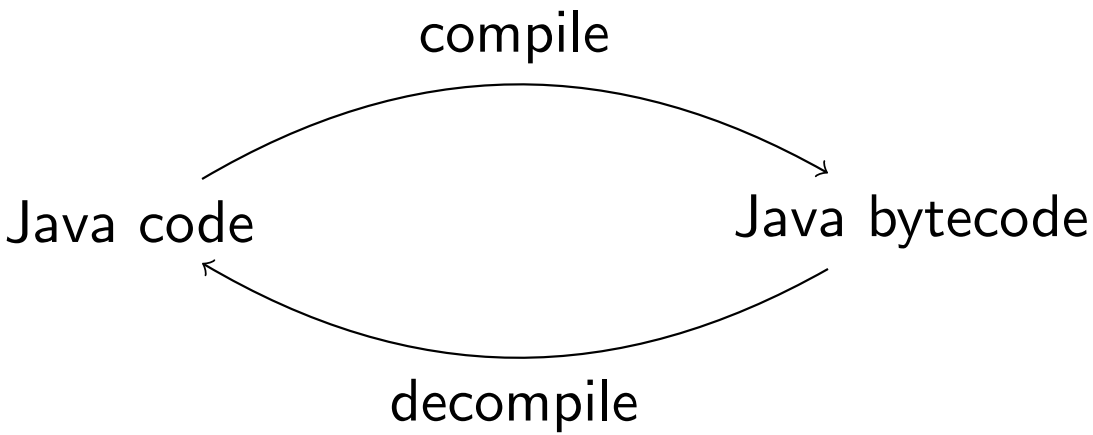
- Provide the input.
- Run the code.
- Compare the output with the expected output.

White box testing



Black box testing

Java code and Java bytecode



Why black box testing?

A Java archive (JAR) file usually only contains the bytecode and not the Java code.

Developers can obfuscate JAR files so that a user of the JAR file does not get much information regarding the original Java code.

Which test cases?

- Likely cases (black box and white box testing).
- Boundary cases (black box and white box testing).
- Cases that cover all branches (white box testing only).
- Cases that cover all execution paths (white box testing only).

Unit testing

A unit test is designed to test a single unit of code, for example, a method.

Such a test should be automated as much as possible; ideally, it should require no human interaction in order to run, should assess its own results, and notify the programmer only when it fails.

A class that contains unit tests is known as a test case.

The code to be tested is known as the unit under test.

JUnit

JUnit is a Java unit testing framework developed by Kent Beck and Erich Gamma.

JUnit is available at http://junit.org/junit5/.

Kent Beck

Kent Beck is an American software engineer and the creator of the Extreme Programming and Test Driven Development software development. He worked at Facebook.

source: Three Rivers Institute

Erich Gamma

Erich Gamma is a Swiss computer scientist and member of the "Gang of Four" who wrote the influential software engineering textbook "Design Patterns: Elements of Reusable Object-Oriented Software." He works at Microsoft.

source: Pearson

Java annotations

Annotations provide data about code that is not part of the code itself. Therefore, it is also called metadata.

In its simplest form, an annotation looks like

```
@Deprecated
```

(The annotation type `Deprecated` is part of `java.lang` and, therefore, need not be imported.)

JUnit contains annotations such as

```
@Test
```

(The annotation type `Test` is part of `org.junit.jupiter.api` and, therefore, needs to be imported.)

An annotation can include elements and their values:

```
@EnabledIfSystemProperty(named="os.arch", matches=".*64.*")
```

(The annotation type `EnabledIfSystemProperty` is part of `org.junit.jupiter.api.condition`.)

A test case

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class ... {
  @Test
  public void ...() {
    ...
  }

  @Test
  public void ...() {
    ...
  }
}
```

Names of test methods

It is good practice to use descriptive names for the test methods.
This makes tests more readable when they are looked at later.

Assertions in test methods

Each test method should contain (at least) one assertion: an invocation of a method of the Assertions class of the `org.junit.jupiter.api` package.

Do not confuse these assertions with Java's `assert` statement.

Methods of the Assertions class

`assertEquals(long, long)`

assert that the two are the same.

`assertEquals(long, long, String)`

assert that the two are the same; if not, the message is used.

Methods of the Assertions class

```
assertEquals(double, double, double)
assertEquals(double, double, double, String)
```

The method invocation

```
Assert.assertEquals(expectedValue, actualValue, delta)
```

asserts

$$|\texttt{expectedValue} - \texttt{actualValue}| < \texttt{delta}$$

Methods of the Assertions class

`assertEquals(Object, Object)`
`assertEquals(Object, Object, String)`

asserts that the objects are equal according to the `equals` method.

`assertSame(Object, Object)`
`assertSame(Object, Object, String)`

asserts that the objects are equal according to the `==` operator.

Methods of the Assertions class

`assertTrue(boolean)`
`assertTrue(boolean, String)`

asserts that the condition is true.

`assertFalse(boolean)`
`assertFalse(boolean, String)`

asserts that the condition is false.

Methods of the Assertions class

`assertNull(Object)`
`assertNull(Object, String)`

asserts that the object is null.

`assertNotNull(Object)`
`assertNotNull(Object, String)`

asserts that the object is not null.

Timeout

Cause a test to fail if it takes longer than a specified time in milliseconds:

```
@Test
public void ...() {
  Assertions.assertTimeout(ofMillis(1000),
    () -> {
      ...
    });
}
```

Exceptions

Cause a test to fail if a specified exception is not thrown:

```
@Test
public void ... () {
  Assertions.assertThrows(IOException.class,
    () -> {
      ...
    });
}
```

Body of unit test method

1. Create some objects.
2. Invoke methods on them.
3. Check the results using a method of the `Assertions` class.

Test case

For each method and constructor (from simplest to most complex)

1. Study its API.
2. Create unit tests.

Example

Write a JUnit test case to test the class `Color`, whose API can be found here. Its JAR can be found here.

Test the constructor

Question

What can we test about the constructor?

Test the constructor

Question

What can we test about the constructor?

Answer

That the created object is not null.

Test the constructor

Question

What can we test about the constructor?

Answer

That the created object is not null.

Question

How many “inputs” does the constructor have?

Test the constructor

Question

What can we test about the constructor?

Answer

That the created object is not null.

Question

How many “inputs” does the constructor have?

Answer

Three.

Test the constructor

Question

How many combinations of "inputs" for the constructor are there?

Test the constructor

Question

How many combinations of "inputs" for the constructor are there?

Answer

$256 \times 256 \times 256 = 16777216 \approx 10^7$.

Test the constructor

Question

How many combinations of "inputs" for the constructor are there?

Answer

$256 \times 256 \times 256 = 16777216 \approx 10^7$.

Question

Can we check all these combinations of "inputs"?

Test the constructor

Question

How many combinations of "inputs" for the constructor are there?

Answer

$256 \times 256 \times 256 = 16777216 \approx 10^7$.

Question

Can we check all these combinations of "inputs"?

Answer

Yes.

Test the acccessors

Question

What can we test about the accessors?

Test the acccessors

Question

What can we test about the accessors?

Answer

That they return the correct values.

Test the constant BLACK

Question

What can we test about the constant `Color.BLACK`?

Test the constant BLACK

Question

What can we test about the constant `Color.BLACK`?

Answer

That it is not null.

Test the constant BLACK

Question

What can we test about the constant `Color.BLACK`?

Answer

That it is not null.

Question

Should we test that the three accessors return 0 for the constant `Color.BLACK`?

Test the constant BLACK

Question

What can we test about the constant `Color.BLACK`?

Answer

That it is not null.

Question

Should we test that the three accessors return 0 for the constant `Color.BLACK`?

Answer

No. This has not been specified in the API.

Test the equals method

Question

What can we test about the `equals` method?

Test the equals method

Question

What can we test about the `equals` method?

Answer

- a `Color` object is equal to itself,
- a `Color` object is equal to a `Color` object with the same RGB values,
- a `Color` object is not equal to a `Color` object with the different RGB values,
- a `Color` object is not equal to `null`, and
- a `Color` object is not equal to an object of another type.

Test the equals method

Question

Can we test that a `Color` object is not equal to a `Color` object with the different RGB values for all possible combinations?

Test the equals method

Question

Can we test that a `Color` object is not equal to a `Color` object with the different RGB values for all possible combinations?

Answer

There are $256 \times 256 \times 256 \times 256 \times 256 \times 256 \approx 10^{14}$ and, hence, no.

Test the equals method

Question

Can we test that a `Color` object is not equal to a `Color` object with the different RGB values for all possible combinations?

Answer

There are $256 \times 256 \times 256 \times 256 \times 256 \times 256 \approx 10^{14}$ and, hence, no.

Question

Which combinations do we check?

Test the equals method

Question

Can we test that a `Color` object is not equal to a `Color` object with the different RGB values for all possible combinations?

Answer

There are $256 \times 256 \times 256 \times 256 \times 256 \times 256 \approx 10^{14}$ and, hence, no.

Question

Which combinations do we check?

Question

Random combinations.