

# Software Engineering Testing and Debugging — Overview

Prof. Dr. Peter Thiemann

Universität Freiburg

29.06.2009

## Essential Reading

- ▶ *Why Programs Fail: A Guide to Systematic Debugging*, A Zeller

## Essential Reading

- ▶ *Why Programs Fail: A Guide to Systematic Debugging*, A Zeller
- ▶ *The Art of Software Testing*, 2nd Edition, G J Myers

## Essential Reading

- ▶ *Why Programs Fail: A Guide to Systematic Debugging*, A Zeller
- ▶ *The Art of Software Testing*, 2nd Edition, G J Myers

## Further Reading

- ▶ *Code Complete*, 2nd Edition, S McConnell

# Cost of Software Errors

\$ 60 billion

**\$ 60 billion**

estimated cost of software errors for US economy per year [2002]

\$ 240 billion

**\$ 240 billion**

size of US software industry [2002]



**\$ 240 billion**

size of US software industry [2002]

incl. profit, sales, marketing, development (50% maybe)

# Cost of Software Errors

estimated

50%

# Cost of Software Errors

estimated

**50%**

of each software project spent on testing

# Cost of Software Errors

estimated

**50%**

of each software project spent on testing

(spans from 30% to 80%)

# Cost of Software Errors

very rough approximation

money  
spent on  
testing  $\approx$  cost of  
remaining  
errors

# Cost of Software Errors

very rough approximation

$$\begin{array}{rcc} \text{money} & & \text{cost of} \\ \text{spent on} & + & \text{remaining} \\ \text{testing} & & \text{errors} \\ & = & \end{array}$$

## Cost of Software Errors

very rough approximation

$$\begin{array}{rcc} \text{money} & & \text{cost of} \\ \text{spent on} & + & \text{remaining} \\ \text{testing} & & \text{errors} \\ & = & \\ 50\% \text{ of size of software} & & \\ \text{industry} & & \end{array}$$

# Brainstorming on Lecture Title

Collect opinions on:

- ▶ What is Testing?
- ▶ What is Debugging?



# A Quiz

A simple program

## Input

Read three integer values from the command line.

The three values represent the lengths of the sides of a triangle.

## Output

Tells whether the triangle is

**Scalene:** no two sides are equal

**Isosceles:** exactly two sides are equal

**Equilateral:** all sides are equal

# A Quiz

A simple program

## Input

Read three integer values from the command line.

The three values represent the lengths of the sides of a triangle.

## Output

Tells whether the triangle is

**Scalene:** no two sides are equal

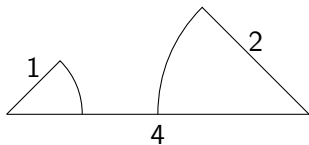
**Isosceles:** exactly two sides are equal

**Equilateral:** all sides are equal

Create a Set of **Test Cases** for this Program

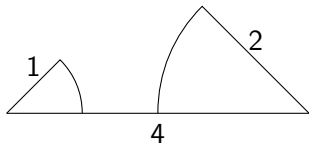
# Solution — 1 Point for each Correct Answer

Q 1: (4,1,2) a **invalid** triangle



## Solution — 1 Point for each Correct Answer

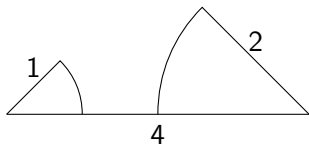
Q 1: (4,1,2) a **invalid** triangle



Why not a valid triangle?

## Solution — 1 Point for each Correct Answer

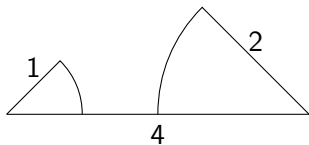
Q 1: (4,1,2) a **invalid** triangle



Why not a valid triangle?  $(a,b,c)$  with  $a > b + c$

## Solution — 1 Point for each Correct Answer

Q 1: (4,1,2) a **invalid** triangle



Why not a valid triangle?  $(a,b,c)$  with  $a > b + c$

Define valid triangles:  $a \leq b + c$

## Solution — 1 Point for each Correct Answer

Q 2: some permutations of previous  $(1,2,4)$ ,  $(2,1,4)$

## Solution — 1 Point for each Correct Answer

Q 2: some permutations of previous  $(1,2,4)$ ,  $(2,1,4)$

Fulfill above definition, but are still invalid.



## Solution — 1 Point for each Correct Answer

Q 2: some permutations of previous (1,2,4), (2,1,4)

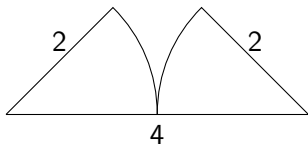
Fulfill above definition, but are still invalid.

Patch definition of valid triangles:

$$a \leq b + c \text{ and } b \leq a + c \text{ and } c \leq a + b$$

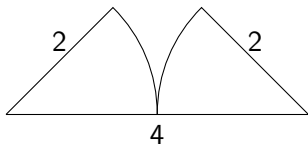
## Solution — 1 Point for each Correct Answer

Q 3: (4,2,2) a **invalid** triangle with **equal** sum



## Solution — 1 Point for each Correct Answer

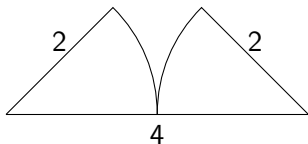
Q 3:  $(4,2,2)$  a **invalid** triangle with **equal** sum



Fulfills above definition, but is invalid (depending on what we want!).

## Solution — 1 Point for each Correct Answer

Q 3: (4,2,2) a **invalid** triangle with **equal** sum



Fulfills above definition, but is invalid (depending on what we want!).

Patch definition of valid triangles:

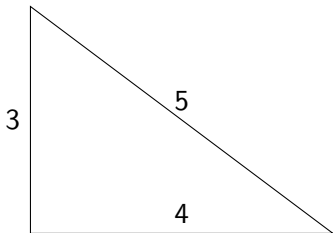
$$a < b + c \text{ and } b < a + c \text{ and } c < a + b$$

## Solution — 1 Point for each Correct Answer

Q 4: some permutations of previous  $(2,2,4)$ ,  $(2,4,2)$

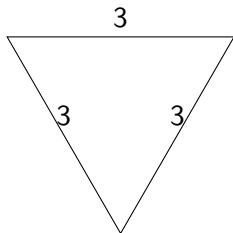
## Solution — 1 Point for each Correct Answer

Q 5: (3,4,5) a **valid scalene** triangle



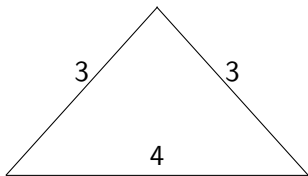
## Solution — 1 Point for each Correct Answer

Q 6: (3,3,3) an **equilateral** triangle



## Solution — 1 Point for each Correct Answer

Q 7: (3,4,3) valid isosceles t.





## Solution — 1 Point for each Correct Answer

Q 8: all permutations of valid isosceles triangle:

$(3,4,3)$ ,  $(3,3,4)$ ,  $(4,3,3)$

## Solution — 1 Point for each Correct Answer

Q 9: one side with **zero** value (0,4,3)

## Solution — 1 Point for each Correct Answer

Q 10: one side with **negative** value  $(-1,4,3)$

## Solution — 1 Point for each Correct Answer

Q 11: all sides zero (0,0,0)

## Solution — 1 Point for each Correct Answer

Q 12: at least one value is non-integer (1,3,2.5)

## Solution — 1 Point for each Correct Answer

Q 13: wrong number of arguments (2,4) or (1,2,3,3)

## Solution — 1 Point for each Correct Answer

Q 14 (the most important one):

Did you specify the expected output in each case?

## About the Quiz

- ▶ Q 1–13 correspond to failures that have actually occurred in implementations of the program
- ▶ How many questions did you answer?  
< 5? 5 – 7? 8 – 10? > 10? All?



## About the Quiz

- ▶ Q 1–13 correspond to failures that have actually occurred in implementations of the program
- ▶ How many questions did you answer?  
< 5? 5 – 7? 8 – 10? > 10? All?
- ▶ Highly qualified, experienced programmers score **7.8** on average

# First Conclusions

- ▶ Finding good and sufficiently many test cases is difficult
- ▶ Even a good set of test cases cannot exclude more failures
- ▶ Without a specification, it is not clear even what a failure **is**

# First Conclusions

- ▶ Finding good and sufficiently many test cases is difficult
- ▶ Even a good set of test cases cannot exclude more failures
- ▶ Without a specification, it is not clear even what a failure **is**

The discipline of Testing is all about Test Cases

# First Conclusions

- ▶ Finding good and sufficiently many test cases is difficult
- ▶ Even a good set of test cases cannot exclude more failures
- ▶ Without a specification, it is not clear even what a failure **is**

The discipline of Testing is all about Test Cases

well, almost ...

# First Conclusions

- ▶ Finding good and sufficiently many test cases is difficult
- ▶ Even a good set of test cases cannot exclude more failures
- ▶ Without a specification, it is not clear even what a failure **is**

The discipline of Testing is all about Test Cases

well, almost ...

Remark: At Ericsson: 35% of code is test cases!

# What is a Bug?


Photo # NH 96566-KN First Computer "Bug", 1945

9:2

9/9

0800 Antenn started  
 1000 " stopped - antenn ✓ { 1.2700 9.037 847 025  
 13<sup>00</sup> (032) MP-MC 2.130476415 ~~2.130476415~~ 4.615925059 (-2) correct  
 (033) PRO 2 2.130476415  
 correct 2.130476415  
 Relays 6-2 in 033 failed special speed test  
 in Relay " 11,000 test.

1100 Started Cosine Tape (Sine check)  
 1525 Started Multi-Adder Test.

1545  Relay #70 Panel F  
 (moth) in relay.

First actual case of bug being found.  
~~1630~~ 1630 Antennant started.  
 1700 closed down.

Relay 2145  
 Relay 3370

Harvard University, Mark II Aiken Relay Calculator

# What is a Bug? Basic Terminology

## Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer's fault, if, e.g., requirements changed)

# What is a Bug? Basic Terminology

## Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer's fault, if, e.g., requirements changed)
2. Defect may cause **infection** of program state during execution (not all defects cause infection)



# What is a Bug? Basic Terminology

## Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer's fault, if, e.g., requirements changed)
2. Defect may cause **infection** of program state during execution (not all defects cause infection)
3. Infected state **propagates** during execution (infected parts of states may be overwritten or corrected)

# What is a Bug? Basic Terminology

## Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer's fault, if, e.g., requirements changed)
2. Defect may cause **infection** of program state during execution (not all defects cause infection)
3. Infected state **propagates** during execution (infected parts of states may be overwritten or corrected)
4. Infection may cause a **failure**: an externally observable error (including, e.g., non-termination)

# What is a Bug? Basic Terminology

## Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer's fault, if, e.g., requirements changed)
2. Defect may cause **infection** of program state during execution (not all defects cause infection)
3. Infected state **propagates** during execution (infected parts of states may be overwritten or corrected)
4. Infection may cause a **failure**: an externally observable error (including, e.g., non-termination)

# What is a Bug? Basic Terminology

## Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer's fault, if, e.g., requirements changed)
2. Defect may cause **infection** of program state during execution (not all defects cause infection)
3. Infected state **propagates** during execution (infected parts of states may be overwritten or corrected)
4. Infection may cause a **failure**: an externally observable error (including, e.g., non-termination)

Defect — Infection — Propagation — Failure

# Failure and Specification

Some failures are obvious

- ▶ obviously wrong output/behaviour
- ▶ non-termination
- ▶ crash
- ▶ freeze

... but most are not!

# Failure and Specification

Some failures are obvious

- ▶ obviously wrong output/behaviour
- ▶ non-termination
- ▶ crash
- ▶ freeze

... but most are not!

In general, what constitutes a failure, is defined by

# Failure and Specification

Some failures are obvious

- ▶ obviously wrong output/behaviour
- ▶ non-termination
- ▶ crash
- ▶ freeze

... but most are not!

In general, what constitutes a failure, is defined by a **specification!**

# Failure and Specification

Some failures are obvious

- ▶ obviously wrong output/behaviour
- ▶ non-termination
- ▶ crash
- ▶ freeze

... but most are not!

In general, what constitutes a failure, is defined by a **specification!**

**Correctness is a relative notion**

— B. Meyer, 1997



# Failure and Specification

Some failures are obvious

- ▶ obviously wrong output/behaviour
- ▶ non-termination
- ▶ crash
- ▶ freeze

... but most are not!

In general, what constitutes a failure, is defined by a **specification!**

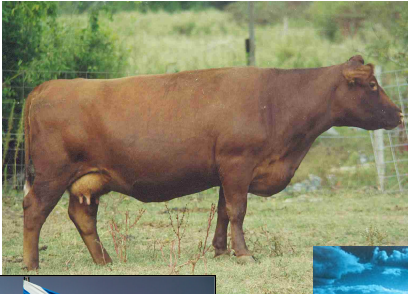
**Correctness is a relative notion**

— B. Meyer, 1997

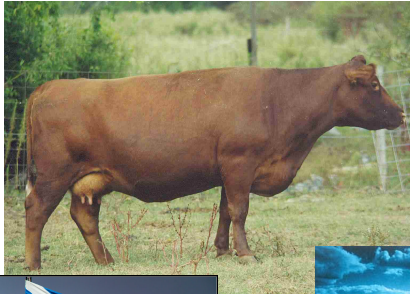
**Every program is correct with respect to SOME specification**

— myself, today

# Specification: Intro



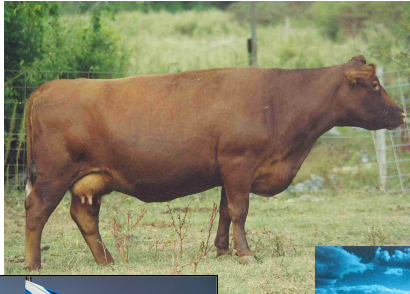
# Specification: Intro



**Economist:**

The cows in Scotland are brown

# Specification: Intro



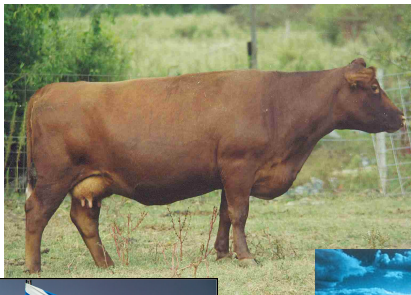
## Economist:

The cows in Scotland are brown

## Logician:

No, there are cows in Scotland of which one at least is brown!

# Specification: Intro



## Economist:

The cows in Scotland are brown

## Logician:

No, there are cows in Scotland of which one at least is brown!

## Computer Scientist:

No, there is at least one cow in Scotland, which is brown on one side!!

## Specification: Putting it into Practice

### Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

## Specification: Putting it into Practice

### Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing sort():

## Specification: Putting it into Practice

### Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing sort():

▶ `sort({3, 2, 5}) == {2, 3, 5}` ✓



# Specification: Putting it into Practice

## Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing sort():

- ▶ `sort({3, 2, 5}) == {2, 3, 5}` ✓
- ▶ `sort({}) == {}` ✓

# Specification: Putting it into Practice

## Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing sort():

- ▶ `sort({3, 2, 5}) == {2, 3, 5}` ✓
- ▶ `sort({}) == {}` ✓
- ▶ `sort({17}) == {17}` ✓

# Specification: Putting it into Practice

## Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing sort():

- ▶ `sort({3, 2, 5}) == {2, 3, 5}` ✓
- ▶ `sort({}) == {}` ✓
- ▶ `sort({17}) == {17}` ✓

# Specification: Putting it into Practice

## Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing sort():

- ▶ `sort({3, 2, 5}) == {2, 3, 5}` ✓
- ▶ `sort({}) == {}` ✓
- ▶ `sort({17}) == {17}` ✓

Specification?

# Specification: Putting it into Practice

## Example

A Sorting Program:

```
public static Integer[] sort(Integer[] a) { ...  
}
```

Testing `sort()`:

- ▶ `sort({3, 2, 5}) == {2, 3, 5}` ✓
- ▶ `sort({}) == {}` ✓
- ▶ `sort({17}) == {17}` ✓

Specification

*Requires:* *a is an array of integers*

*Ensures:* *returns the sorted argument array a*

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is an array of integers

*Ensures:* returns the sorted argument array a

*Is this a good specification?*

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is an array of integers

*Ensures:* returns the sorted argument array a

*Is this a good specification?*

`sort({2, 1, 2}) == {1, 2, 2, 17} ❌`

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is an array of integers

*Ensures:* returns a sorted array with *only elements from*  
a



## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is an array of integers

*Ensures:* returns a sorted array with *only elements from*  
a

$\text{sort}(\{2, 1, 2\}) == \{1, 1, 2\}$  ✘

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is an array of integers

*Ensures:* returns a *permutation* of a that is sorted

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is an array of integers

*Ensures:* returns a *permutation* of a that is sorted

`sort(null)` throws `NullPointerException` ❌

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is a *non-null* array of integers

*Ensures:* returns a permutation of a that is sorted

## Example Cont'd

### Example

```
public static Integer[] sort(Integer[] a) { ...  
}
```

### Specification

*Requires:* a is a *non-null* array of integers

*Ensures:* returns the *unchanged* reference a containing a permutation of the *old* contents of a that is sorted

# The Contract Metaphor

**Contract** is preferred specification metaphor for procedural and OO PLs

first propagated by B. Meyer, *Computer* 25(10)40–51, 1992

Same Principles as Legal Contract between a Client and Supplier

**Supplier** aka **Implementer**, in JAVA, a class or method

**Client** Mostly a caller object, or human user for `main()`

**Contract** One or more pairs of **ensures/requires** clauses defining mutual obligations of client and implementer

# The Meaning of a Contract

Specification (of method  $C::m()$ )

*Requires:*    *Precondition*

*Ensures:*    *Postcondition*

*"If a caller of  $C::m()$  fulfills the **required Precondition**, then the class  $C$  **ensures** that the **Postcondition** holds after  $m()$  finishes."*

# The Meaning of a Contract

Specification (of method  $C::m()$ )

*Requires:* Precondition

*Ensures:* Postcondition

*"If a caller of  $C::m()$  fulfills the **required Precondition**, then the class  $C$  **ensures** that the **Postcondition** holds after  $m()$  finishes."*

Often the following **wrong** interpretations of contracts are seen:

**Wrong!**

*"Any caller of  $C::m()$  must fulfill the **required Precondition**."*

**Wrong!**

*"Whenever the **required Precondition** holds, then  $C::m()$  is executed."*



# Specification, Failure, Correctness

Define precisely what constitutes a **failure**

A method **fails** whenever it is called in a state fulfilling the required precondition of its contract and it does not terminate in a state fulfilling the postcondition to be ensured.

Non-termination, abnormal termination considered as failures here

# Specification, Failure, Correctness

Define precisely what constitutes a **failure**

A method **fails** whenever it is called in a state fulfilling the required precondition of its contract and it does not terminate in a state fulfilling the postcondition to be ensured.

Non-termination, abnormal termination considered as failures here

Define precisely what **correctness** means

A method is **correct** if in all cases when it is started in a state fulfilling the required precondition it terminates in a state fulfilling the postcondition to be ensured.

This amounts to proving **Absence of Failures!**

# Testing vs Verification

## TESTING

Goal: find evidence for **presence** of failures

Testing means to execute a program with the intent of detecting failure

Related techniques: code reviews, program inspections

# Testing vs Verification

## TESTING

Goal: find evidence for **presence** of failures

Testing means to execute a program with the intent of detecting failure

Related techniques: code reviews, program inspections

## VERIFICATION

Goal: find evidence for **absence** of failures

Testing cannot guarantee correctness, i.e., absence of failures

Related techniques: code generation, program synthesis (from spec)

# Debugging: from Failures to Defects

- ▶ Both, testing and verification attempts exhibit **new** failures
- ▶ **Debugging** is a systematic process that finds and eliminates the defect that led to an observed failure
- ▶ Programs without **known** failures may still contain defects:
  - ▶ if they have not been verified
  - ▶ if they **have been** verified,  
but the failure is not covered by the specification

## Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing

# Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing

Test cases



Code under test



Code checking success

# Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing

Test cases

Test case generator



Code under test



Code checking success

Test oracle

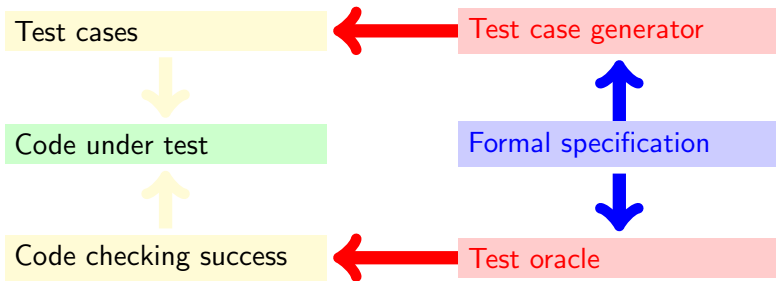




# Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing

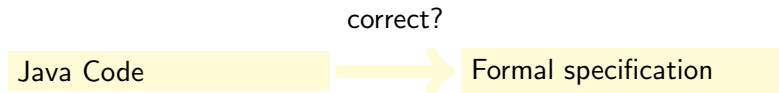


# Formal Verification of Program Correctness

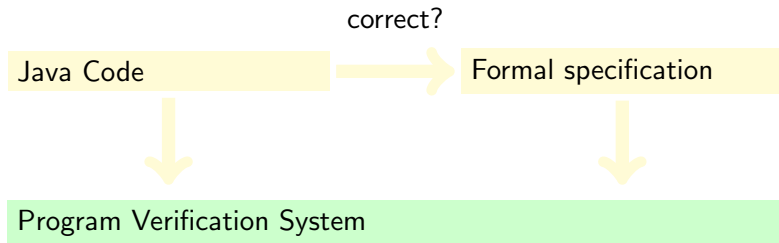
Java Code

Formal specification

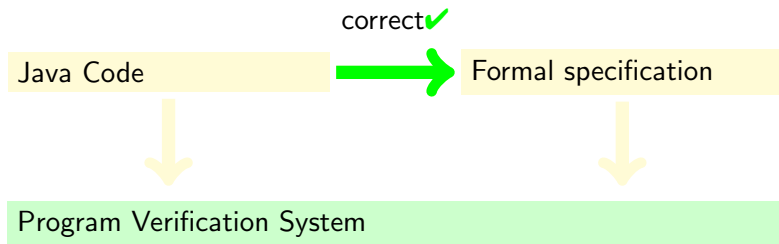
# Formal Verification of Program Correctness



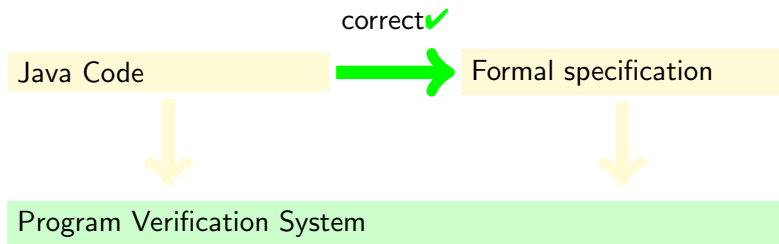
# Formal Verification of Program Correctness



# Formal Verification of Program Correctness



# Formal Verification of Program Correctness



Computer support essential for verification of real programs

`synchronized java.lang.StringBuffer append(char c)`

- ▶ ca. 15.000 proof steps
- ▶ ca. 200 case distinctions
- ▶ Two human interactions, ca. 1 minute computing time

# Tool Support is Essential

## Some Reasons for Using Tools

- ▶ Automate repetitive tasks
- ▶ Avoid typos, etc.
- ▶ Cope with large programs

# Tool Support is Essential

## Some Reasons for Using Tools

- ▶ Automate repetitive tasks
- ▶ Avoid typos, etc.
- ▶ Cope with large programs

## Tools Used

- ▶ Automated running of tests: `JUNIT`
- ▶ Debugging: `ECLIPSE` debugger