Software Engineering

Software Testing and Verification- Introduction

Srinivas Pinisetty

28.02.2024

Software is everywhere

Complexity, evolution, reuse, multiple domains/teams, $\cdots$

Software bug...

- Error
- Fault
- Failure
- ...

A software bug is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. – Wikipedia

Introduction: Testing, Debugging, (Specification) and Verification

Introduction to techniques to get (some) certainty that your program does what it is supposed to do.

- Does my program do what it's supposed to do?
  - If not, why?
  - Have I understood exactly what it is supposed to do?

Introduction: Testing, Debugging, (Specification) and Verification

Introduction to techniques to get (some) certainty that your program does what it is supposed to do.

- Does my program do what it's supposed to do?
  - If not, why?
  - Have I understood exactly what it is supposed to do?
- Can I give any guarantees that my program does the right thing?

Introduction: Testing, Debugging, (Specification) and Verification

Introduction to techniques to get (some) certainty that your program does what it is supposed to do.

- Does my program do what it's supposed to do?
  - If not, why?
  - Have I understood exactly what it is supposed to do?
- Can I give any guarantees that my program does the right thing?
- Introduction and overview of main techniques.
  - Orientation of main concepts.

Cost of Software Errors

$ 312 billion

(annual global cost)

Source: Cambridge University, Judge Business School 2013

`http://www.prweb.com/releases/2013/1/prweb10298185.htm`

Cost of Software Errors

estimated

50%

of programmers time spent on finding and fixing bugs.

Cost of Software Errors

$ 407 billion

Size of global software industry in 2013.

Source: Gartner, March 2014
`http://www.gartner.com/newsroom/id/2696317`

Cost of bugs approximately 3/4 of the size of the whole industry...

Software fault examples: Ariane 5 rocket

- Exploded right after launch
- Conversion of 64-bit float to 16-bit integer caused an exception (made it crash)
- European space agency spent 10 years and 7 billion USD to produce Ariane 5

Software fault examples: Pentium Floating Point Bug

- Incorrect result through floating point division
- Rarely encountered in practice
- 1 in 9 billion floating point divides with random parameters would produce inaccurate results (Byte magazine)
- 475 million dollars, reputation of Intel.

Cost of Software Errors: Conclusion

Huge gains can be realized in SW development by:

- systematic
- efficient
- tool-supported

testing, debugging, and verification methods

In addition ...

The earlier bugs can be removed, the better.

Errors in Safety Critical Systems

Not just economic loss...

Therac-25 Radiotherapy Machine (1985-87)

- Patients overdosed.
- Three dead, two severely injured.
- SW bug causing radiation level entry to be ignored.

Errors in Safety Critical Systems

Not just economic loss...

Therac-25 Radiotherapy Machine (1985-87)

- Patients overdosed.
- Three dead, two severely injured.
- SW bug causing radiation level entry to be ignored.

Toyota Unintended Acceleration (2000-05)

- Bugs in electronic throttle control system.
- Car kept accelerating on its own.
- May have caused up to 89 deaths in accidents.
- Recalls of 8 million vehicle.

Defects in software: Problem sources

Defects in software: Problem sources

- Requirements: Incomplete, inconsistent, $\cdots$

Defects in software: Problem sources

- Requirements: Incomplete, inconsistent, $\cdots$
- Design: Flaws in design

Defects in software: Problem sources

- Requirements: Incomplete, inconsistent, $\cdots$
- Design: Flaws in design
- Implementation: Programming errors, $\cdots$

Defects in software: Problem sources

- Requirements: Incomplete, inconsistent, $\cdots$
- Design: Flaws in design
- Implementation: Programming errors, $\cdots$
- Tools:Defects in support systems and tools used

Brainstorm

How can you get some assurance that a program does what you want it to do?

Brainstorm

How can you get some assurance that a program does what you want it to do?

Techniques for assurance

- Testing
- Pair programming, code review, $\cdots$
- Formal verification

Brainstorm

How can you get some assurance that a program does what you want it to do?

Techniques for assurance

- Testing
- Pair programming, code review, $\cdots$
- Formal verification

- Usually more assurance = more effort
- Research focus on more assurance for less effort

Testing Debugging and Verification

- What is Testing?

Testing Debugging and Verification

- What is Testing?
  - Evaluating software by observing its execution
  - Execute program with the intent of finding failures (try out inputs, see if outputs are correct)

Testing Debugging and Verification

- What is Testing?
  - Evaluating software by observing its execution
  - Execute program with the intent of finding failures (try out inputs, see if outputs are correct)
- What is Debugging?

Testing Debugging and Verification

- What is Testing?
  - Evaluating software by observing its execution
  - Execute program with the intent of finding failures (try out inputs, see if outputs are correct)
- What is Debugging?
  - Understand why a program does not do what it is supposed to, usually via tool support such as the Eclipse debugger
  - The process of finding a defect given a failure
  - Relating a failure to a defect

Testing Debugging and Verification

- What is Testing?
  - Evaluating software by observing its execution
  - Execute program with the intent of finding failures (try out inputs, see if outputs are correct)
- What is Debugging?
  - Understand why a program does not do what it is supposed to, usually via tool support such as the Eclipse debugger
  - The process of finding a defect given a failure
  - Relating a failure to a defect
- What is Verification?

Testing Debugging and Verification

- What is Testing?
  - Evaluating software by observing its execution
  - Execute program with the intent of finding failures (try out inputs, see if outputs are correct)
- What is Debugging?
  - Understand why a program does not do what it is supposed to, usually via tool support such as the Eclipse debugger
  - The process of finding a defect given a failure
  - Relating a failure to a defect
- What is Verification?
  - Determine whether a piece of software fulfils a set of formal requirements in every execution
  - Formally prove method correct (find evidence of absence of failure)

What is a Bug? Basic Terminology

Bug-Related Terminology

1. Defect (aka bug, fault) introduced into 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 into 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 into 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 into 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 into 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 into 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: a specification!

Specification: Intro

- Specification: An unambiguous description of what a program should do.
- Bug: Failure to meet specification.

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 on one side is brown!!

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()`:

- `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 sorted array*

Example Cont'd

Example

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

Specification

Requires: a *is an array of integers*

Ensures: *returns a sorted array*

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 a sorted array*

Is this a good specification?

$\texttt{sort}(\{2, 1, 2\}) == \{1, 2, 2, 17\}$ ✘

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`

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

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*

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

Client:	(Caller) implementer of calling method, or human user for `main()`
Supplier:	(callee) aka implementer of a method
Contract:	One or more pairs of requires / ensures clauses defining mutual obligations of supplier and client

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 callee `C.m()` ensures that the Postcondition holds after `C.m()` finishes."

Specification, Failure, Correctness

What constitutes a failure

A method fails when 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.

Specification, Failure, Correctness

What constitutes a failure

A method fails when 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.

A method is correct means:

whenever it is started in a state fulfilling the required precondition, then it terminates in a state fulfilling the postcondition to be ensured.

Correctness amounts to proving absence of failures! A correct method cannot fail!

This Module

Introduction to techniques to get (some) certainty that your program does what it is supposed to.

Testing

Test: try out inputs, see if outputs are correct

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

This course: terminology, testing levels, unit testing, black box vs white box, principles of test-set construction/coverage, automated and repeatable testing (JUnit)

Debugging

Understand why a program does not do what it is supposed to, usually via tool support such as the Eclipse debugger

- Testing attempts exhibit new failures
- Debugging is a systematic process that finds (and eliminates) the defect that led to an observed failure

This course: Input minimisation, systematic debugging, logging, program dependencies (tracking cause and effect)

Verification

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

Verification: Mathematically prove method correct

- Goal: find evidence for absence of failures

Verification

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

Verification: Mathematically prove method correct

- Goal: find evidence for absence of failures

Code

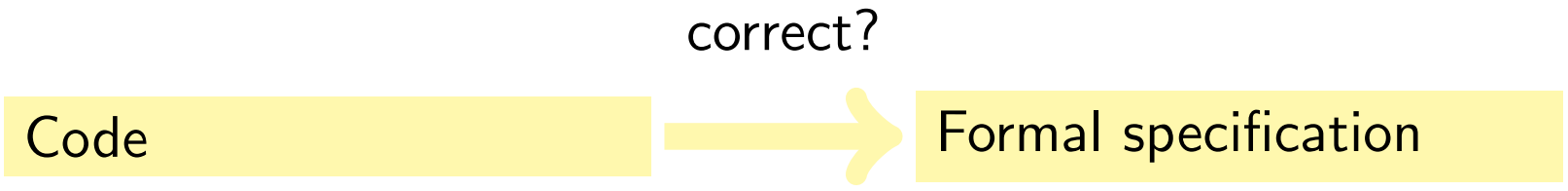
Formal specification

Verification

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

Verification: Mathematically prove method correct

- Goal: find evidence for absence of failures

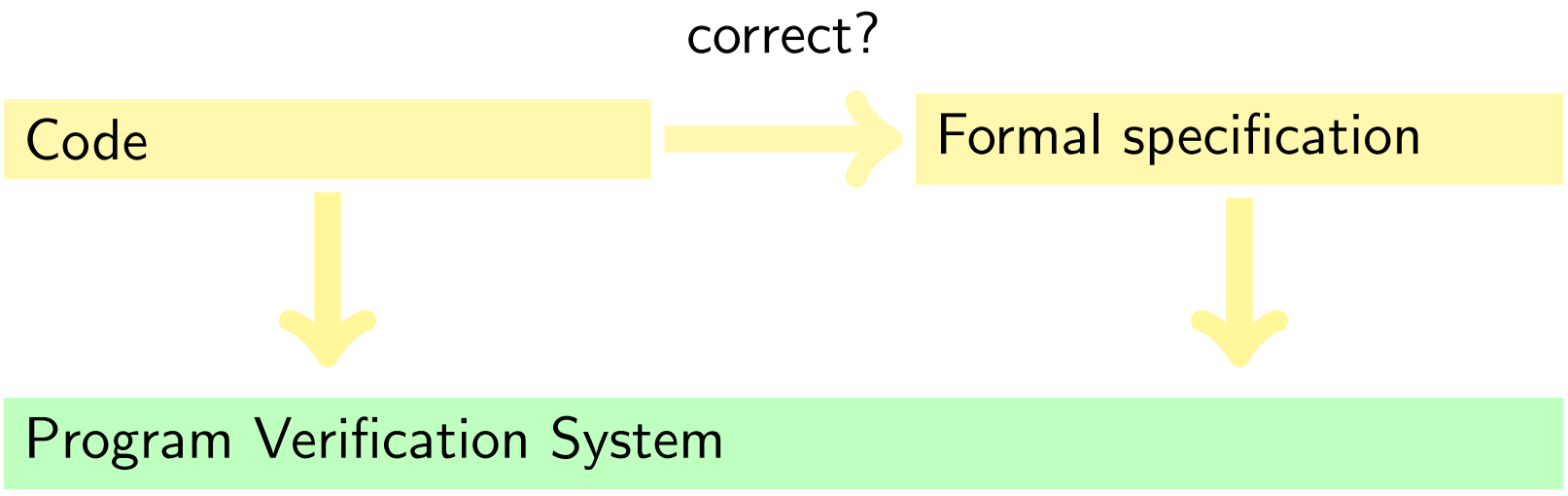


Verification

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

Verification: Mathematically prove method correct

- Goal: find evidence for absence of failures

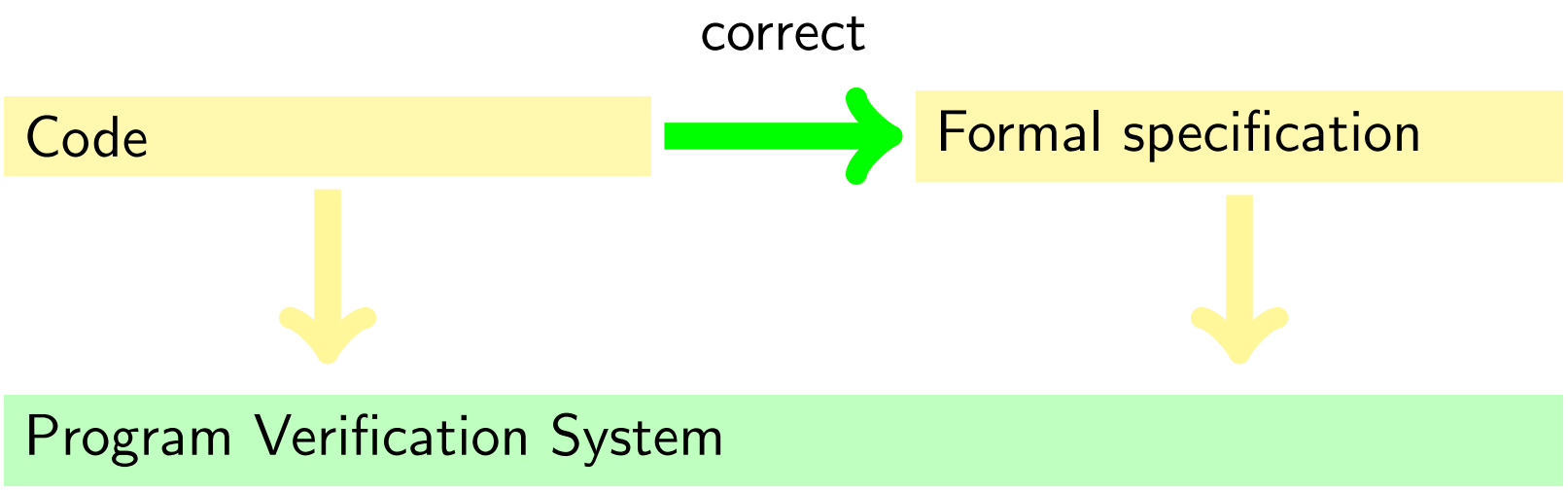


Verification

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

Verification: Mathematically prove method correct

- Goal: find evidence for absence of failures

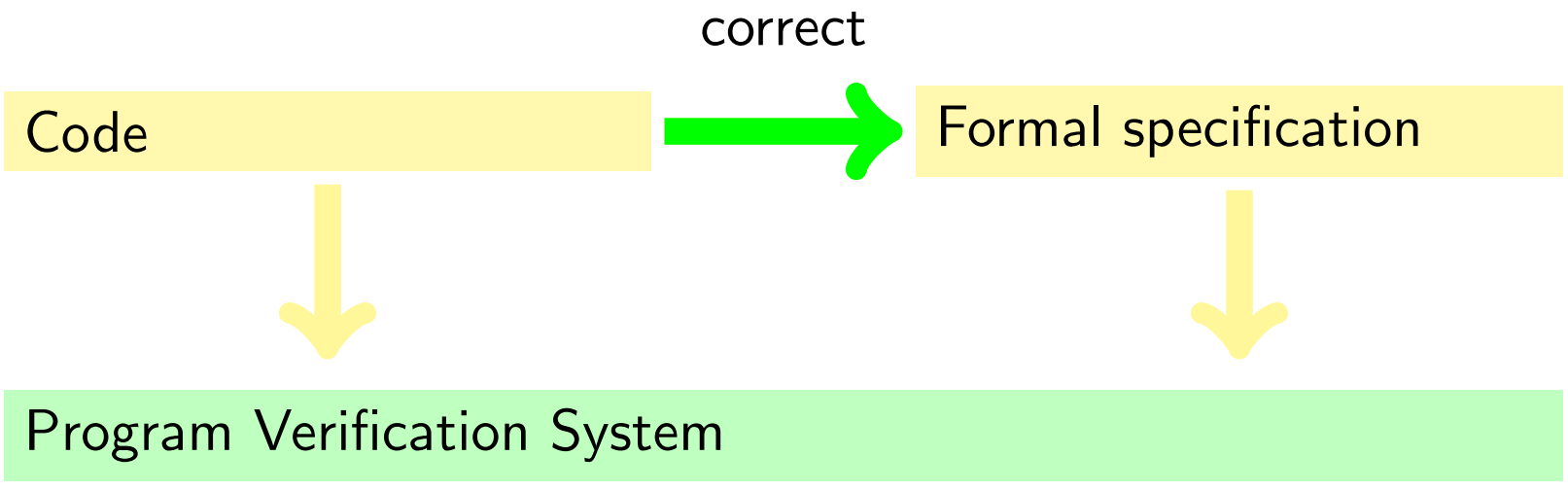


Verification

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

Verification: Mathematically prove method correct

- Goal: find evidence for absence of failures



This course: Formal verification (logics, tool support)

Contents

How do we get some certainty that your program does what it is supposed to?

- Testing: Try out inputs, does what you want?
  terminology, testing levels, unit testing, black box vs white box, principles of test-set construction/coverage, automated and repeatable testing (JUnit)
- Debugging: What to do when things go wrong
  Input minimisation, systematic debugging, logging, program dependencies (tracking cause and effect)
- Formal specification & verification: Prove that there are no bugs **Will be a part of the next module**
  Logic, define specification formally, assertions, invariants, formal verification tools, formal proofs

Contents

How do we get some certainty that your program does what it is supposed to?

- Testing: Try out inputs, does what you want?
  terminology, testing levels, unit testing, black box vs white box, principles of test-set construction/coverage, automated and repeatable testing (JUnit)
- Debugging: What to do when things go wrong
  Input minimisation, systematic debugging, logging, program dependencies (tracking cause and effect)
- Formal specification & verification: Prove that there are no bugs **Will be a part of the next module**
  Logic, define specification formally, assertions, invariants, formal verification tools, formal proofs

Tools

Tools Used

- Automated running of tests: JUnit
- Debugging: Eclipse debugger.
- Formal specification and verification: Dafny