Testing, Debugging, and Verification

TDA566/DIT082

Introduction

Wolfgang Ahrendt, Vladimir Klebanov, Moa Johansson

29 October 2012

Organisational Stuff

Course Home Page

`www.cse.chalmers.se/edu/course/TDA566/`

Google News Group

- Sign up via course home page (follow News link) entering
  - real name
  - person number (not necessary if you use `@student` address)
- Changes, updates, questions, discussions. Don't post solutions!

Passing Criteria

- Written exam 17 Dec 2012; re-exam Apr 2013
- Three lab hand-ins
- Exam and labs can be passed separately

Team

Teacher

- Moa Johansson (`jomoa`), MJ

Course Assistant

- Gabriele Paganelli (`gabpag`), GP

office hours: see course page

...append `@chalmers.se` to obtain email address

Structure

Course Structure

Topic	# Lectures	Exercises	Lab
Intro	1	✘	✘
Testing	3	✔	✔
Debugging	2	✔	✘
Formal Specification	3	✔	✔
Verification	3	✔	✔
Test Generation	2	✔	✘

Course Literature

Essential Reading

- *Why Programs Fail: A Guide to Systematic Debugging*[1)], 2nd edition, A Zeller
- *The Art of Software Testing*[1)], 2nd Edition, G J Myers

Further Reading

- *Introduction to Software Testing*, P Ammann & J Offutt
- *Code Complete*, 2nd Edition, S McConnell

Additional important references, papers on course page

1) available online as e-books via Chalmers library, navigate to 'E-book collections', 'Books24x7', and register

Labs, Exercises

Labs

- Submission via Fire, linked from course home page
- You must team up in groups of two
  1. team up with the partner of your choice
  2. if you can't find one, call for a partner via Google group
  3. if the above does not work, contact Gabriele (gabpag)
- If submission get returned, ca. one week for correction
- Testing 16 Nov, Formal Spec 30 Nov, Verification 13 Dec

Exercises

- One exercise session for each topic (5 in all)
- Before each session:
  - we post exercise questions on web page
  - you try to solve them (as much as possible, might not have covered all in lectures)
- During each exercise session:
  - we solve remaining questions and discuss solutions together

Course Evaluation

5 student representatives (choosen randomly)

- feedback meetings with teachers
- course evaluation

Mathias Forsén	`forssenm`
Hans Lämås	`lamas`
Kasper Karlsson	`kasperk`
Markus Johansson	`jmarcus`
Jonas Åström	`jonasas`

For email address append: `@student.chalmers.se`

All participants: web questionnaire after the course

Cost of Software Errors

$ 60 billion

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

Cost of Software Errors

$ 240 billion

Size of US software industry [2002]

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

Cost of Software Errors

estimated

50%

of each software project spent on testing

(spans from 30% to 80%)

Cost of Software Errors

Very rough estimate:

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

Cost of Software Errors

Very rough estimate:

$$\text{money spent on testing} + \text{cost of remaining errors}$$

$$=$$

50% of size of software industry

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 . . .

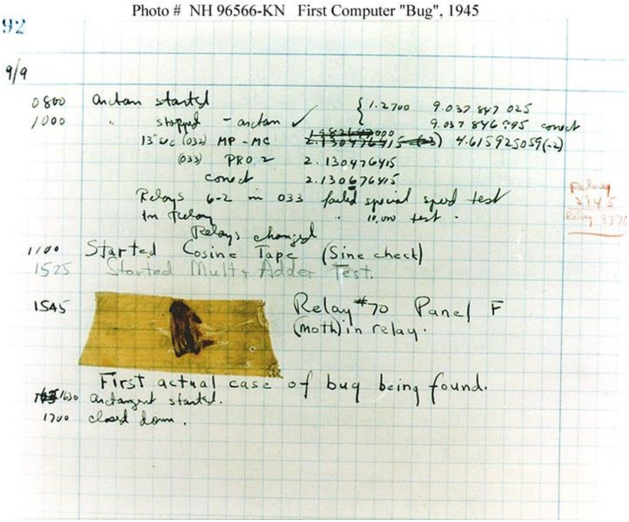
The earlier bugs can be removed, the better.

Brainstorming on Course Title

Collect opinions on:

- What is Testing?
  - Evaluating software by observing its execution
  - A mental discipline that helps IT professionals develop better software
- What is Debugging?
  - The process of finding a defect given a failure
  - Relating a failure to a defect and subsequent fixing of the defect
- What is Verification?
  - Determine whether the products of a given phase in SW development fulfill requirements established in previous phase
  - Determine whether a piece of software fulfills a set of formal requirements in every execution

What is a Bug? Basic Terminology



Harvard University, Mark II Aiken Relay Calculator

see `www.jamesshuggins.com/h/tek1/first_computer_bug.htm`

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

— Bertrand Meyer, 1997

Each program is correct with respect to SOME specification

—Wolfgang A.

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) { ... }
```

Testing `sort()`:

- $\texttt{sort}(\{3, 2, 5\}) == \{2, 3, 5\}$ ✔
- $\texttt{sort}(\{\}) == \{\}$ ✔
- $\texttt{sort}(\{17\}) == \{17\}$ ✔

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?

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

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`

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

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*

`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*

cf. the cow joke — unfortunately, in programming the unexpected happens

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 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 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

Define precisely what correctness means

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.

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

VERIFICATION

Goal: find evidence for absence of failures, contract being honoured

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

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

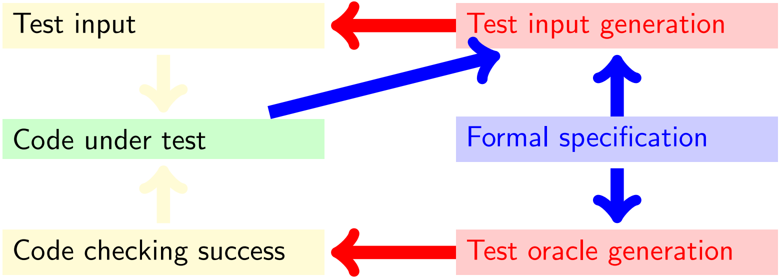
Debugging: from Failures to Defects

- 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 manually/informally verified, but the defect has been overlooked
  - if they have been verified, but the failure is not covered by the specification

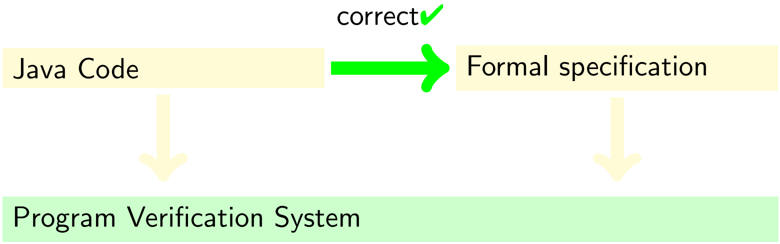
Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing



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

Course Contents

- Testing
  terminology, black box vs white box, test generation, coverage
- Debugging
  terminology, tracking, execution control, inspection, localisation
- Formal specification
  contracts, assertions, invariants, JML, logic
- Automatic test case generation
  partitions, symbolic execution, coverage
- Formal verification
  Hoare calculus, formal proofs, loop invariants

Tool Support is Essential

Some Reasons for Using Tools

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

Tools Used in This Course

- Automated running of tests: JUnit
- Debugging: Eclipse debugger.
- Formal specification: JML tools
- Automatic test case generation: JML tools, KeY/TestGen
- Formal verification: KeY verification system