Software Testing: Introduction

Mohammad Mousavi

Halmstad University, Sweden

http://ceres.hh.se/mediawiki/DIT085

Testing and Verification (DIT085),
Chalmers and GU, January 23, 2015

Contact information

Courses Web Pages

`http://ceres.hh.se/mediawiki/DIT085`

Check for news, updates, course material and much more!

Mohammad Mousavi

Office	Halmstad University, E 320 Fridays: Jupiter, 427
E-mail	M.R.Mousavi@hh.se
Telephone	(035 16) 7122
WWW	http://ceres.hh.se/mediawiki/Mohammad_Mousavi

Objectives and assessment

Learning objectives: Knowledge

1. understand the basic terminology of testing,
2. name and describe different testing techniques and approaches,
3. describe the connection between software development phases and kinds of testing, and
4. exemplify and describe a number of different test methods, and be able to use them in practical situations.

Objectives and assessment

Learning objectives: Skills

1. construct appropriate and meaningful test cases,
2. interpret and explain the results a test campaign,
3. plan and produce appropriate documentation for testing,
4. write models in at least one formal specification language, and
5. apply different testing techniques on realistic examples.

Objectives and assessment

Learning objectives: Judgment

1. compare different tools and techniques for testing software, and plan their use in appropriate contexts,
2. compare and judge alternatives to testing, such as model checking and runtime verification
3. identify and hypothesize about sources of program failures, and reflect on how to better verify the correctness of such programs.

Objectives and assessment

Evaluation method

- Practical project (P),
- Written exams (W), closed book

The final mark (VG, G, or U) = min (P, W)

Project: WhatsUpGU

General Description

- Server:
  - connection-based (TCP-IP-based) server,
  - to be implemented in Java,
  - multi-threaded,
  - XML interface for adding, editing, and fetching messages.
- Client:
  - Android-based,
  - able to deal with communication faults.

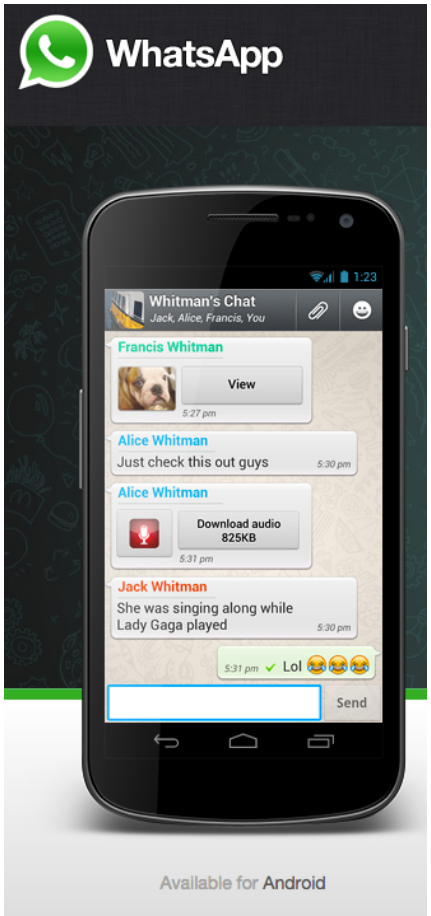


Photo: Copyright WhatsApp Inc.

Project: WhatsUpGU

Testing Perspective

- test-driven development,
- unit testing using jUnit,
- coverage metrics using Cobertura (or similar tools),
- integration testing, developing stubs using jMockIt (or similar tools),
- model checking using Uppaal, and
- UI testing using the Visual GUI Testing tool.

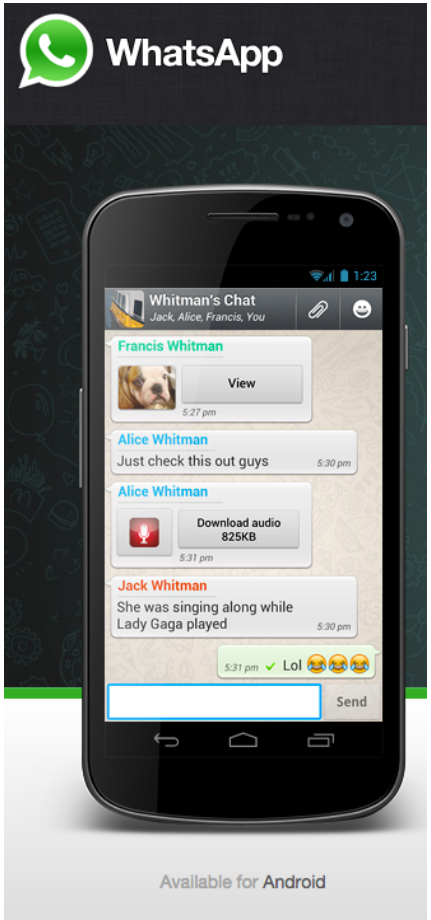


Photo: Copyright WhatsApp Inc.

Project: WhatsUpGU

Schedule and Deadlines

- Forming Groups: January 27 at 17:00
- Phase 1: TDD of a Unit: February 13 at 17:00,
- Phase 2: Integration (Testing) of the Server: February 20 at 17:00
- Phase 3: Specification and Model Checking: February 27 at 17:00
- Phase 4: UI Testing: March 13 at 17:00

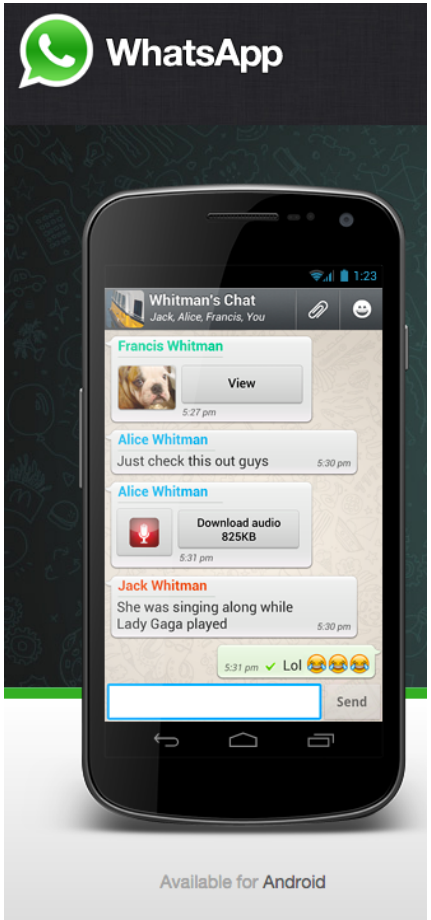


Photo: Copyright WhatsApp Inc.

Project: WhatsUpGU

Schedule and Deadlines

By the deadline:

- Deliverable submitted on GUL,
- Oral presentation given by all group members to the instructor.

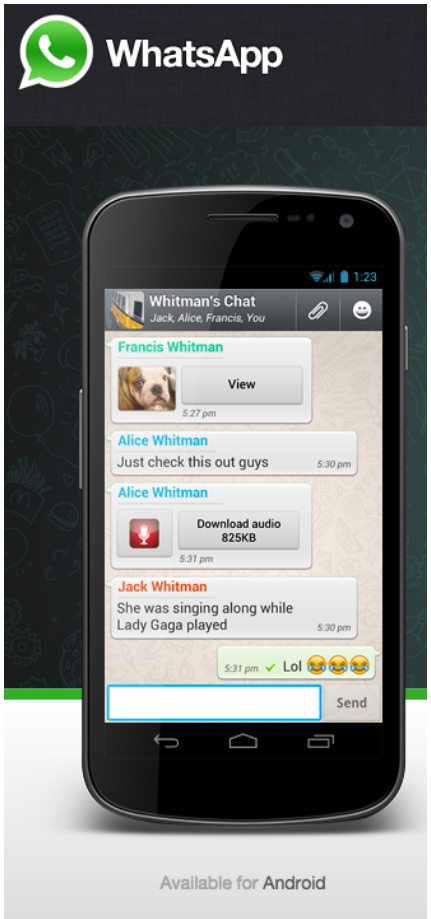


Photo: Copyright WhatsApp Inc.

Our Order of Business

- Terminology and Functional Testing (today)
- Test-Driven Development and jUnit (January 30)
- Coverage Criteria (February 6 and 13)
- Model Checking (February 20)
- GUI Testing (February 27)
- Slicing and Debugging (March 6)
- Reviewing Model Examination (March 13)
- Guest Lectures from Volvo (To be confirmed: March 4 and 12)

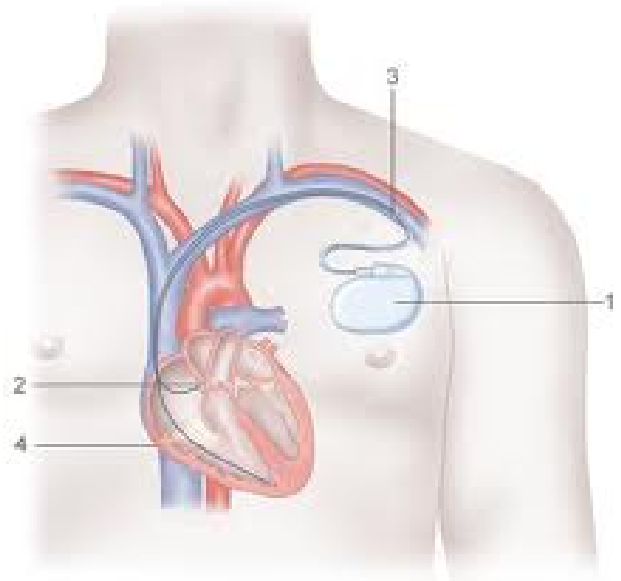
General Information

- Text book: P. Ammann and J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.
- Papers to be handed out during each lecture.
- Recommended books posted on the course page.

Software at Your Heart...

Software glitches in pacemakers

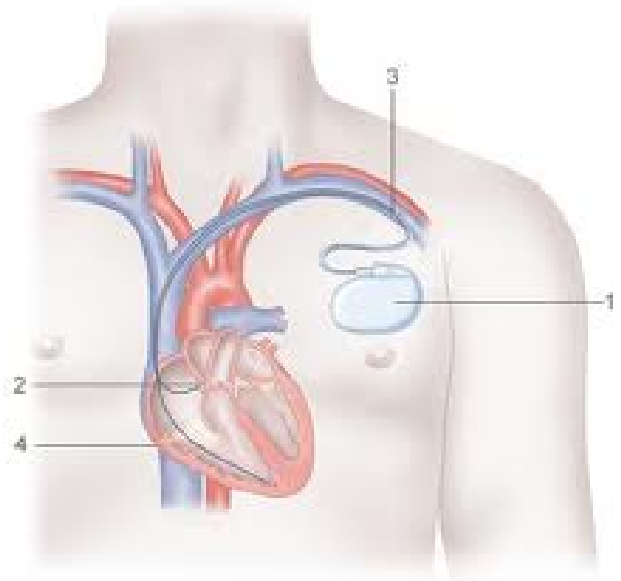
Company said it has not received any reports of deaths or clinical complications resulting from the glitch, which appears in about 53 out of every 199,100 cases.



Software at Your Heart. . .

At least 212 deaths from device failure in five different brands of implantable cardioverter-defibrillator (ICD) according to a study reported to the FDA . . . .

[Killed by Code, 2010]



Why?

"Bugs"

- Facts of life! (correct by construction: not always possible / affordable)
- Serious consequences (Pentium bug, OV Chipcard, etc.)

Why?

A Classic Bug

- Ariane 5 explosion report:

This loss of information was due to specification and design errors in the software ... caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented ...

Why?

The NorthWest Blackout "Bug"

- Widespread blackouts in 2003
- Affecting 8 US states and a part of Canada
- Traced back to a race condition bug
- Surfaced after 3 million hours of operation

Moral of the Story

If it can go wrong, it will go wrong.

Why?

"Bugs"

- 2002 Costs: 60 Billion USD (only USA).
- *Coders introduce bugs at the rate of 4.2 defects per hour of programming. If you crack the whip and force people to move more quickly, things get even worse.* *[Watts Humphreys]*

Why?

Quest for Quality

- *Software quality will become the dominant success criterion in the software industry.*
  [ACM Workshop on Strategic Directions in Software Quality]
- Testing:
  - a way to achieve better quality
  - >50% of the development costs

Why?

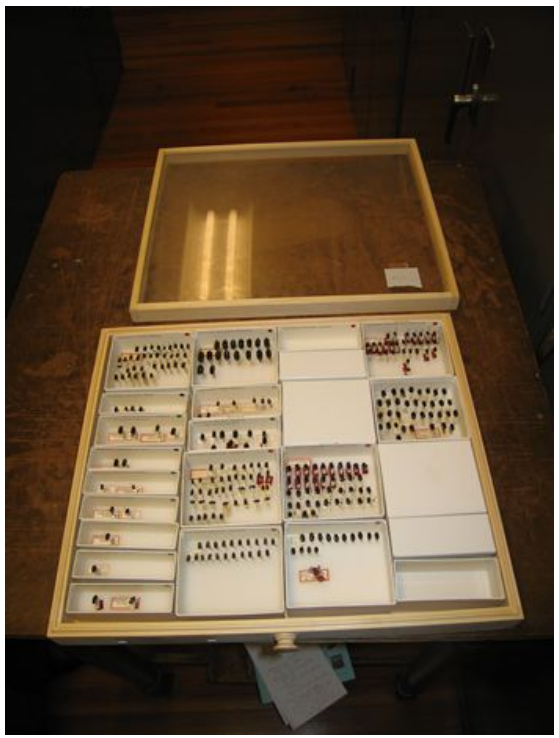
Bezier's Testing Levels

- L0 debugging (ad hoc, few input/outputs)
- L1 showing that software works (validating some behavior)
- L2 showing that software does not work (scrutinizing corner cases)
- L3 reducing risks (organizing and prioritizing test goals)
- L4 mental discipline for quality (central to development)

What?

Sorts of "Bug"

- Fault: incorrect implementation
  - commission: implement the wrong specification
  - omission: forget to implement a specification
    (the more difficult one to find and resolve)
- Error: incorrect system state (e.g., incorrect value for a variable)
- Failure (anomaly, incident) : visible error in the behavior



How?

Spec: A program that inputs an integer, and outputs $2 * i^3$.

```
int i;
i << cin;
i = 2 * i;
i = exp(i,3);
cout << i;
```

How?

Spec: A program that inputs an integer, and outputs $2 * i^3$.

```
1: int i;
2: i << cin;
3: i = 2 * i;
4: i = exp(i,3);
5: cout << i;
```

- Conceptual mistake: confusing the binding power of operators
- Fault: Statements 3 and 4 are in the wrong order
- Failure:
  Test-case: on input 1, the program must output 2.
  input 1 ... output 8!

What?

Validation vs. Verification

- Validation: Have we made the right product; compliance with the intended usage
  often: user-centered, manual process, on the end product
- Verification: Have we made the product right; compliance between artifacts of different phases
  often: artifact-driven, formalizable and mechanizable process among all phases

What?

Testing

- Planned experiments to:
  1. reveal bugs (turn faults into failures, test to fail),

     Testing can show the presence of bugs, but not the absence. *[Dijkstra]*

  2. gain confidence in software quality (test to pass)

What?

RIP Process

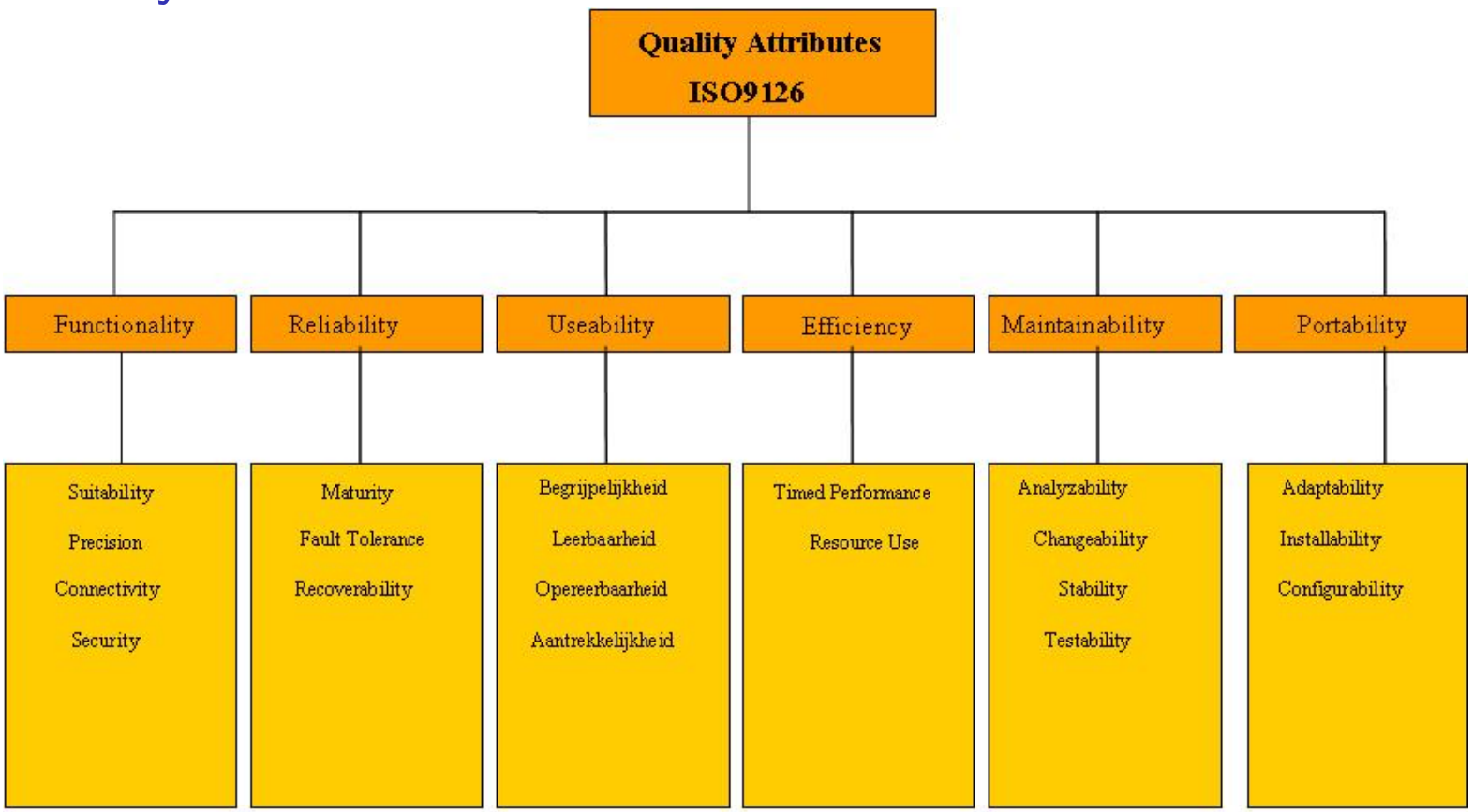
- Reachability: triggering the statements containing the fault,
- Infection: triggering the fault to produce incorrect state
- Propagation: carrying the fault to the visible behavior (output)

What?

- Test case (the plan):
  input (execution condition / behavior) and output (pass / fail conditions)
- Testing: planning and executing test-cases (how?).

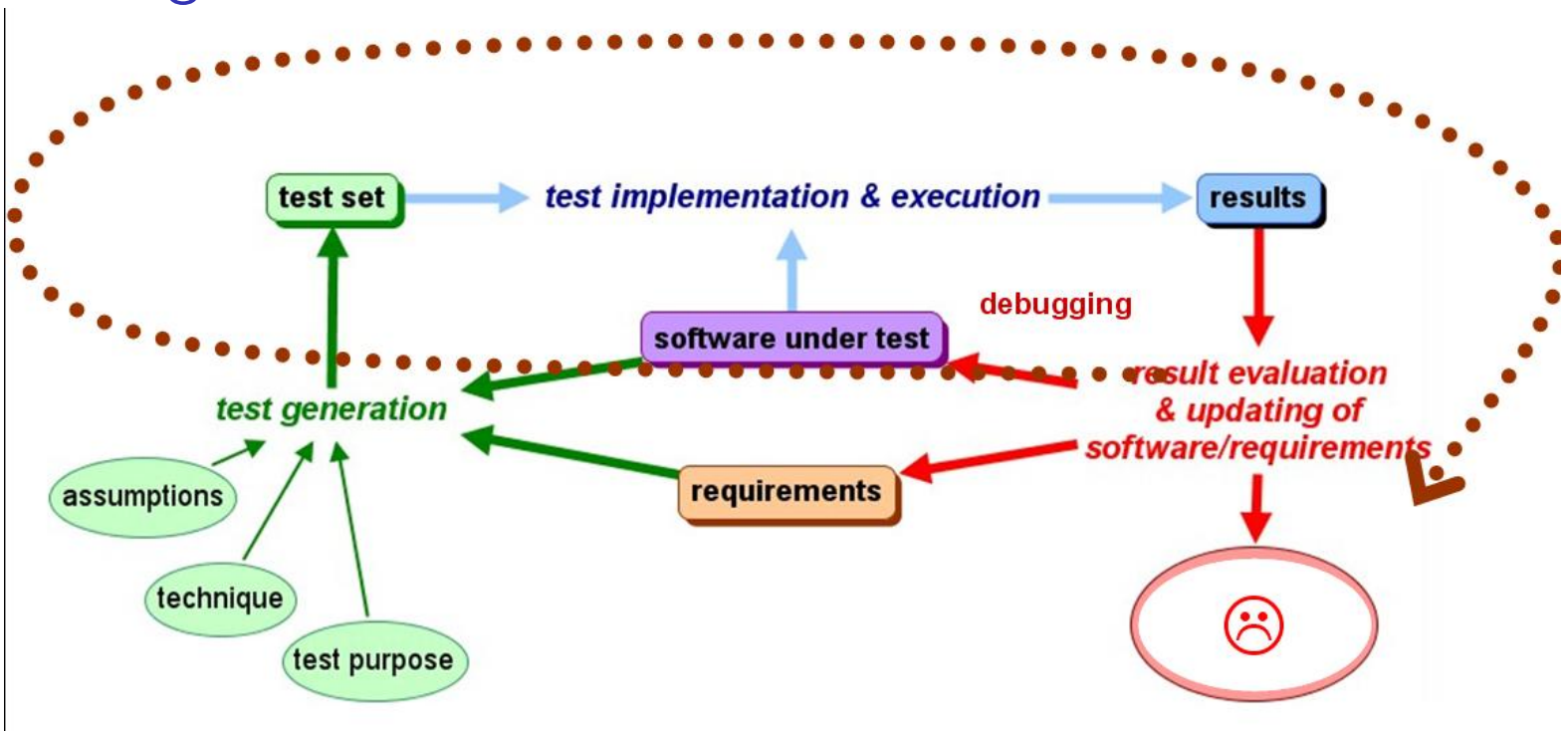
What?

Quality Attributes



How?

Testing


test set
test implementation & execution
results
debugging
software under test
result evaluation
& updating of
software/requirements
test generation
assumptions
technique
test purpose
requirements

How?

Testing


Test Policy
Test Management
Test Method
Test Tooling and Utilities

How?

- Testing: planning and executing test-cases.
  1. designing test-cases (manual, automatic: models, formal specs),
  2. executing them (manual or automatic: scaffolding, fixture),
  3. distinguishing failures or correct executions (manual: experts, automatic: oracles, models)
  4. giving feed back for debugging / changing specification

How?

Aspects of Testing

- Functional testing:
  assumption: software is a function from inputs to outputs
  covering aspects of specification
  suitable for black-box testing (but can be enhanced with information from the code)
  - \+ program independent: tests can be planned early
  - \+ tests are re-usable
  - \- gaps: untested pieces of software
  - \- redundancies: the same statements may be tested several times

Functional Testing: Mortgage Example

Spec. Write a program that takes
three inputs: gender (boolean), age([18-55]), salary ([0-10000])
and output the total mortgage for one person

Mortgage = salary * factor,
where factor is given by the following table.

Category	**Male**	**Female**
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

From: P.C. Jorgensen. Software Testing: A Craftsmans Approach.

An Implementation

Mortgage (male:Boolean, age:Integer, salary:Integer): Integer
if male **then**
  return $((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$
else {female}
  return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$
end if

Is this implementation correct? No way, 12 bugs!

Functional Testing

Mortgage (male:Boolean, age:Integer, salary:Integer): Integer
if male **then**
  return $((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$
else {female}
  return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$
end if

Possible test cases:
inputs: representatives from each age range and for each gender and salary 1,
output: factors as given by the table
(similar to equivalence testing; wait till next sessions!)

How?

Aspects of Testing

- Structural testing:
  covering aspects of program
  examples: code coverage, branch coverage
  - \+ giving insight to the effectiveness of test
  - \- more complicated than functional testing
  - \- incapable of detecting errors of omission

Structural Testing

Spec.: input: an integer x $[1..2^{16}]$
output: x incremented by two, if x is less than 50, x decremented by one, if x is greater than 50, and 50, otherwise.

```
if x < 50 then
   x = x + 1;
end if
if x > 50 then
   x = x - 1;
end if
return x
```

Structural Testing

```
if x < 50 then
   x = x + 1;
end if
if x > 50 then
   x = x - 1;
end if
return x
```

Test-cases: sufficiently many random inputs until all statements are at least executed once, manually check the outputs with the spec.

Input	Output	Pass/Fail
1540	1539	P
2783	2782	P
3222	3221	P
30	31	F

Structural Testing

First "Debugged" Version:

```
if x < 50 then
   x = x + 2;
end if
if x > 50 then
   x = x - 1;
end if
return x
```

Input	Output	Pass/Fail
1540	1539	P
2783	2999	P
3222	3221	P
30	32	P

Have we tested enough?

Structural Testing

```
if x < 50 then
   x = x + 2;
end if
if x > 50 then
   x = x - 1;
end if
return x
```

Input	Output	Pass/Fail
49	50	F

Pesticide paradox: debugging old faults may produce new bugs (or "wake" old bugs up).

How?

Moral of the Story

Testing aims at covering some (abstract) artifact:

- Functional testing: requirements (logical partitions, formulae, graphs, trees)
- Structural testing: program (control or data flow graphs)

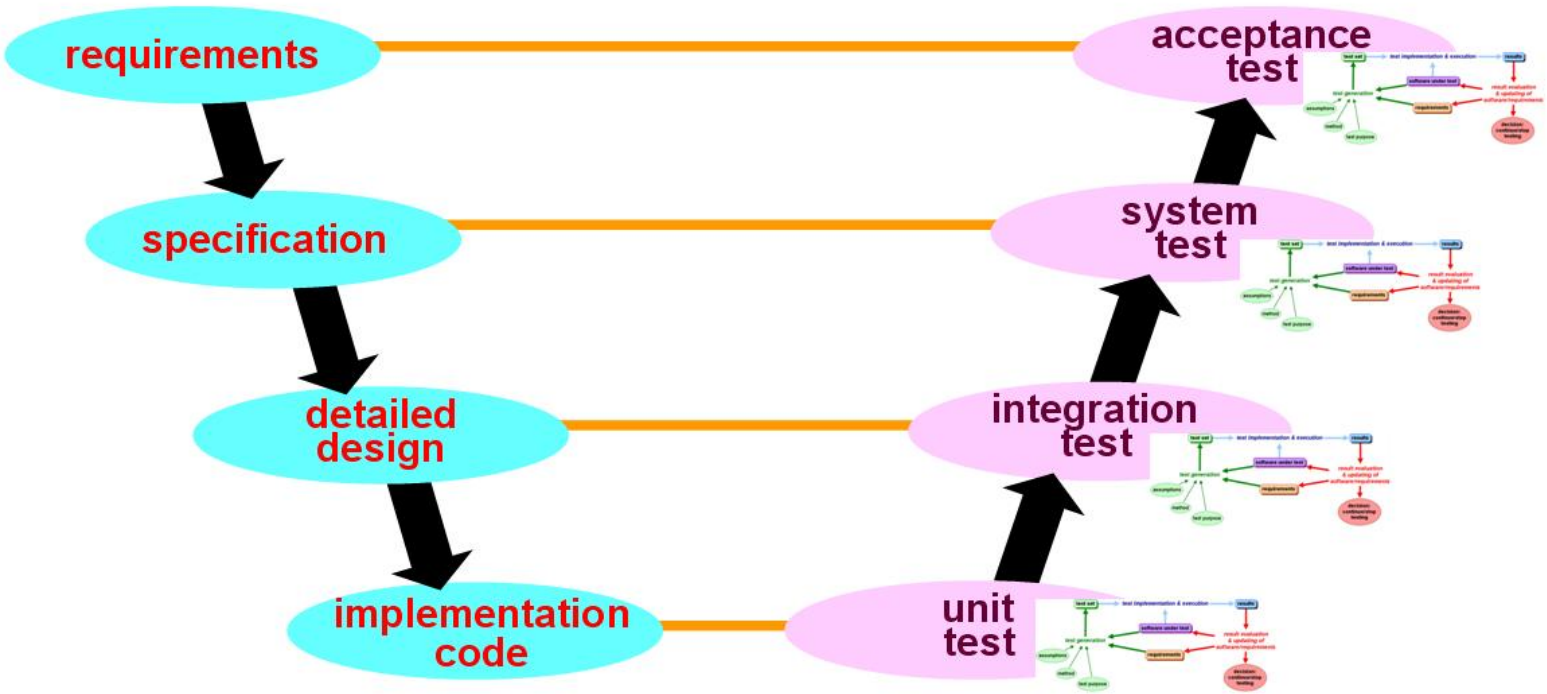
How?

Ideal Mix

- Functional and structural testing at various levels (unit, integration, system)
- Structural measures for the effectiveness of functional test-cases

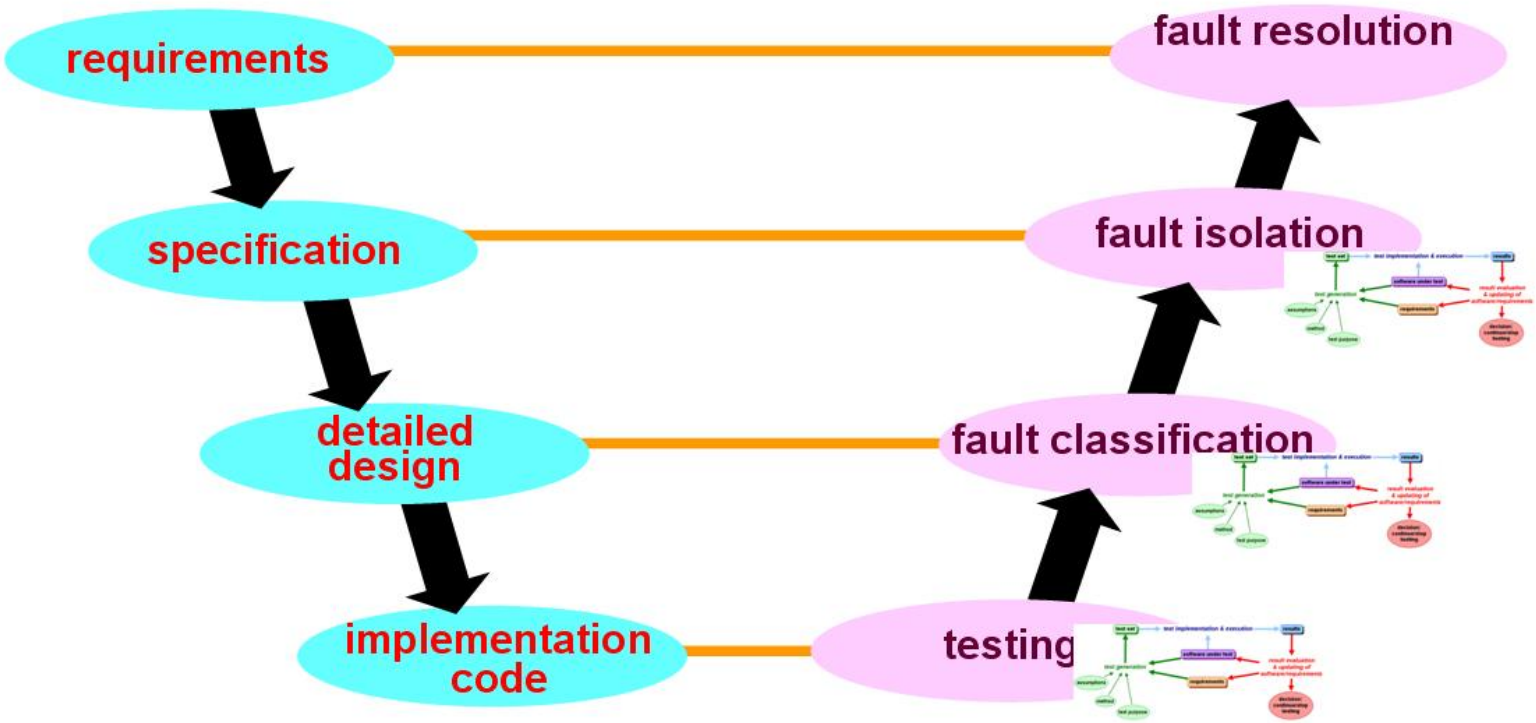
When?

V Model


requirements
specification
detailed design
implementation code
acceptance test
system test
integration test
unit test

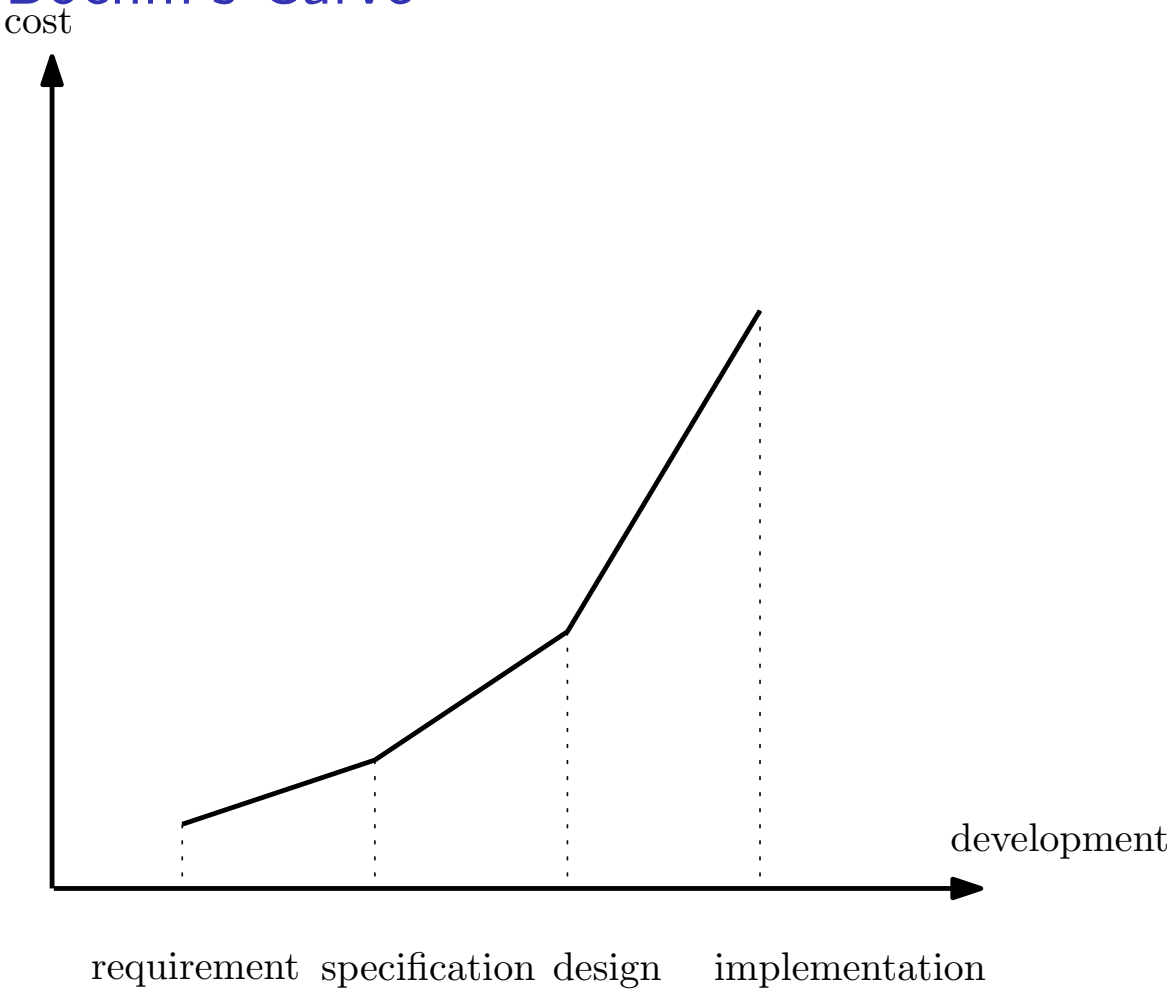
When?

V Model


requirements
specification
detailed design
implementation code
testing
fault classification
fault isolation
fault resolution

When?

Boehm's Curve



When?

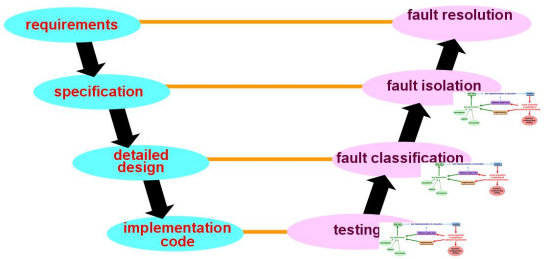
Dealing with Bugs

1-4 Putting errors in (producing bugs),

5-7 finding bugs:

- testing
- fault classification
- fault isolation

8 removing bugs



What Else?

Alternatives

- Static Analysis:
  test abstract properties without running the program,
  e.g., uninitialized/unused variables, empty/unspecified cases, coding standards, checking for design (anti)patterns.
  - \+ automatic and scalable for generic and abstract properties;
  - \+ existing powerful tools;
  - \- involves approximation (true negatives and false positives); complicated (may involve theorem proving) for concrete and specific properties (proving the abstraction function to be "correct")

What Else?

Alternatives

- Model Checking:
  test the state-space for formally specified properties.
  - \+ rigorous analysis, push-button technology;
  - \- not (yet) applicable to many industrial cases (state-space explosion)