Chapter 5

Software Testing

Prof. Mirco Tribastone, Ph.D.

24 January 2013

Validation and Verification

- Validation: Building the right product.
  - Does the software meet the expectations of the customer?
- Verification: Building the product *right*.
  - Does the software conform to its specification?

When to check quality:

- In some software development processes, V&V is done as early as possible (e.g., prototyping, agile).
- It is understood that problems discovered early are easier and less expensive to fix.
- However, there are parts of the specification that can be checked only when the system is ready to be deployed.

Functional and Nonfunctional Properties

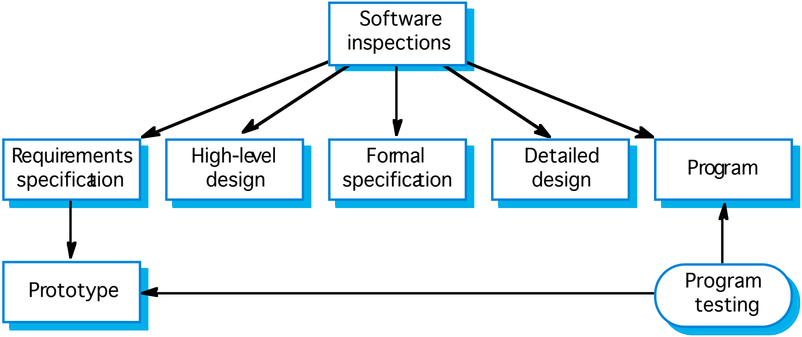
- Functional properties are related to *what* a system (or a part of it) is supposed to do.
    - Use cases in the UML.
- Nonfunctional (or *extrafunctional*) properties are related to *how* the system carries out an operation.
    - Performance; e.g., response time or throughput.
    - Security.
    - Availability; e.g., uptime 99.999%.
- Some nonfunctional properties are more difficult to check during early stages of the development process.

Tools for Validation and Verification

- *Software inspection* analyses requirement documents, designs, and source code (the latter, often automatically).
  - It is a **static** method: It does not require an executable artefact, hence it can be applied throughout all the stages of software development.

- *Software testing* uses an executable representation of the system (**dynamic** method).
  - The product is exercised with test input data
  - The resulting output is checked against the specification.
  - If there is no agreement, an error is found which must be fixed.
  - Different forms according to the knowledge assumed for the system under study: *black-box* or *white-box*.

V&V and the Development Processes



from http://www.cs.st-andrews.ac.uk/ifs/Books/SE7/Presentations/index.html

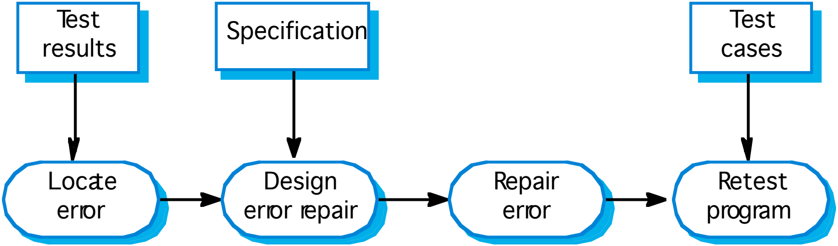
Important Points

- Software inspections can only check the agreement between a program and its specification.
- They cannot show that the software is operationally useful.
- Nor can they check nonfunctional properties (but may give hints).
- Software testing can only detect errors, **not** prove their absence.
- Testing all possible execution paths for nontrivial programs is **impossible**.
- They are not competing techniques, rather they are complementary.

Related Activity: Debugging

- Defect testing and debugging are distinct processes.
- Verification and validation is concerned with establishing the existence of defects in a program.
- Debugging is concerned with locating and repairing these errors.
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.

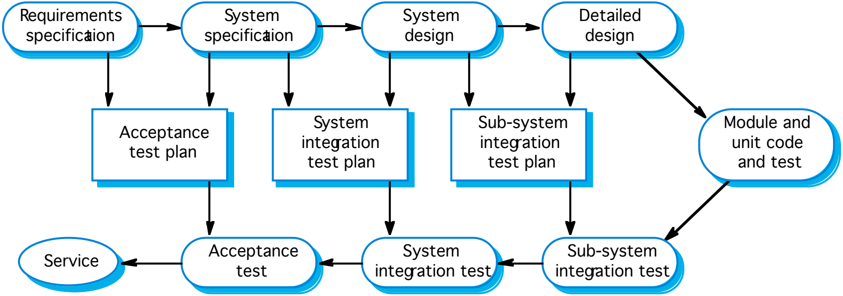
The Debugging Process



Key activity: regression testing

- Re-run the tests (or a subset of them) after a problem is fixed.
- It is not uncommon that a fix introduces errors elsewhere!

The V-Model of Development


Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

- For instance, in an object-oriented design:

  classes ⟶ **components** ⟶ **overall system**

Structure of a Software Test Plan

- **Testing process**
- **Requirements traceability**
  Tests should cover at least all the requirements provided by the users.
- **Tested items**
  Complete coverage of all artefacts is in general very difficult (too expensive). Items to be tested should be listed here.
- **Testing schedule**
- **Test recording procedures**
  Results must be recorded to give the possibility of checking later whether tests have been done correctly.
- **Hardware and software requirements**
- **Constraints**
  For example, staff shortages, deadlines, . . .

Software Inspections

- Empirical studies have shown that they are effective in detecting large amounts of errors in software.
- Many errors may be detected in a single inspection.
    - Recall, it is a static methods which does not require a running system.
    - With software testing, usually only one defect at a time may be discovered: the system usually crashes when an error occurs.
- They reuse domain and programming language knowledge: reviewers are likely to have seen the types of error that commonly arise.

Program Inspection

- It is a formal methodology for reviewing documents.
- It looks for defects such as logical errors, anomalies in the code, or non-compliance with standards.
- The process may have different variants according to the organisation in which it is performed.

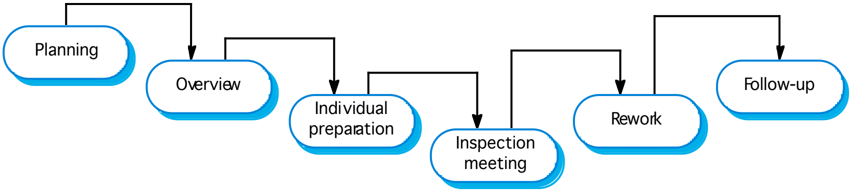
Typical pre-conditions

- Availability of a precise specification.
- Availability of syntactically correct code (or design).
- An error check-list.
    - This is dependent on the programming language. The weaker the typing, the longer the list.

Composition of the Reviewing Team

- **Author**
  Responsible for fixing defects discovered during the review.
- **Inspector**
- **Reader**
  Paraphrases the code during an inspection meeting.
- **Scribe**
  Records the outcome of the inspection meeting.
- **Moderator**
  Manages the process. Responsible for scheduling possible follow-up meetings.

The Program Inspection Process



- Planning is the responsibility of the moderator: choose a team, fix dates, . . .
- At the overview the author presents the program under inspection.
- At the inspection meeting errors are reported. Meetings should be kept relatively short (e.g., under 2 h).
- Rework is the author's responsibility.
- Follow-up may be needed to assess the code in case of major changes required.

Typical Checks

- **Data faults**
  Base indices for arrays? Possibility of buffer overflows?
- **Control faults**
  For each conditional statement, is the condition correct? Are loops guaranteed to terminate? Are compound statements correctly bracketed?
- **Input/output faults**
  Are all input variables used? Are output variables used? Can unexpected inputs cause corruption (e.g., null pointers)?
- **Exception management**
  Have all possible error conditions been taken into account?

Automated Static Analysis

- Performed by software tools which process the source code in search of potentially dangerous situations.
- Does not replace program inspection by humans, as it checks for more *mechanical* errors:
  - Variables used before initialisations, variables declared but never used, variables never used between two successive assignments.
  - Unreachable code.
  - Return values of functions/methods that are not used.
- Static analysers are typically available in Integrated Development Environments.
- Much more useful for weakly typed languages.

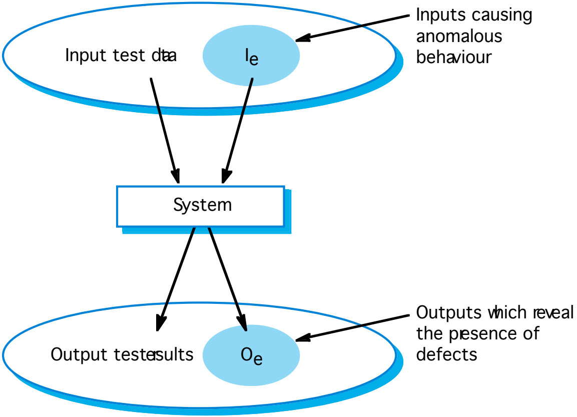
Software Testing

- **Component** (or unit) testing
  - Testing of individual program components.
    The notion of *component* depends on the programming language under consideration.
  - Usually under the responsibility of the authors.
  - Tests are based on the developers' experience.
- **System** testing
  - Testing of integrated components that form a (sub-)system.
  - Usually under the responsibility of an independent team.
  - Tests are based on a system specification.

Goals of Software Testing

- **Validation testing**
  - Demonstrates that the software meets the requirements.
  - It is successful when the system operates as intended.
  - The system is exercised using typical input data.
  - Does not reveal the absence of faults though!
- **Defect testing**
  - Discover faults that may lead to unintended behaviour or failure.
  - It is successful when the test makes the system perform incorrectly.
  - Revels the presence, not the absence of faults!
  - Guidelines on what to test
    - Functionality accessed from menus.
    - Combinations of functions accessed through the same menu (e.g., text formatting).
    - User input forms with correct and incorrect input.

Functional (Black-Box) Defect Testing


Input test dta
I_e
Inputs causing anomalous behaviour
System
Output testesults
O_e
Outputs which reveal the presence of defects

- The system (or component) is treated as a black box.
- Behaviour understood by relating inputs to outputs.
- It is only concerned with the functionality, not its actual implementation.

Other General Testing Guidelines

- Choose inputs that force the system to general all error messages.
  (It is important to have a specification at hand)
- Design inputs that cause buffers to overflow.
- Repeat the same input or input series several times.
- Force invalid outputs to be generated.
- Force computation results to bee too large or too small.

Partitioning

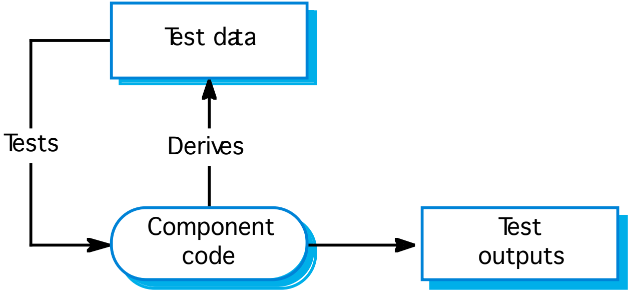
- Selecting relevant input data for testing.
- Based on the assumption that some inputs are somewhat similar: if one is troublesome, so will be all the others belonging to the same *class*

Example

```
class Account {
  public float getBalance() { ... }
  public void withdraw(float amount) { ... }
}
```

- Partition the floats into:
  - Negative values
  - Zero
  - Positive values:
    - $<$ getBalance()
    - $=$ getBalance()
    - $>$ getBalance()
  - Another dimension: more than two decimal digits!

Structural Testing

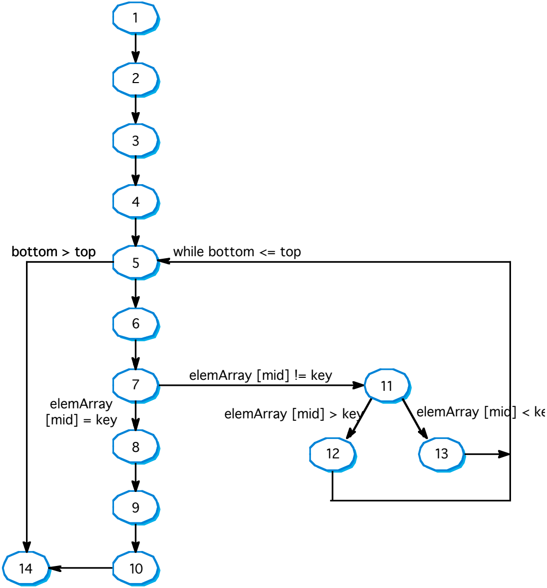


- Also called *white-box* testing.
- Test cases are inferred from the program structure, which is required to be known.
- Can be done incrementally, knowledge of the program can be used to add further test cases.
- The objective is to test all program statements (not all path combinations).

Path Testing

- Ensures that each test input covers a different path in the control flow of the system
- May use a high-level representation with a graph where nodes represent statements, and arcs denote the flow of control.
- Exhaustive path coverage may be expensive to guarantee in realistic scenarios.

Path Testing



Testing Nonfunctional Properties

Performance Testing

- Nonfunctional requirements of software systems are typically expressed as Service Level Agreements (SLAs) between clients and software developers.
- In most cases SLAs concern **performance**, i.e., how well the functionality is performed with respect to time.
- For instance:
    - In 95% of the cases, the **response time** must be less than 250 ms.
    - The system must support 100,000 transactions per seconds.
    - ...
- These concerns are increasingly important for distributed systems.

Stress Testing

- It is a typical technique to gradually increase the system load.
- For each load level, the tester measures the achieved quality of service (e.g., response time) and compares it against the relevant SLA.
- The test may also highlight functional problems:
    - Increasing loads may cause system malfunctions.
- Well-written applications exhibit a graceful degradation of performance at excessive loads.
- Perfectly functional systems may have serious performance problems.
- Fixing a performance problem may introduce serious functional errors.
- Regression testing should take place.

Example: A Simple Distributed Java App

```
public class Server extends Thread {
  public void run() {
    try {
      ServerSocket s = new ServerSocket(8081);
      while (true) {
        Socket client = s.accept();
        Thread.sleep(1000);
        client.getOutputStream().write("OK\n".getBytes());
        client.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Example: A Simple Distributed Java App

```
public class Client extends Thread {

  public void run() {
    Socket s;
    try {
      s = new Socket("localhost", 8081);
      BufferedReader r = new BufferedReader(
        new InputStreamReader(
          s.getInputStream()));
      System.out.println(r.readLine());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Example: A Simple Distributed Java App

```
 public static void main(String[] args)
     throws InterruptedException {
  new Server().start();
  int N = ...;
  Client[] clients = new Client[N];
  for (int i = 0; i < N; i++) {
    clients[i] = new Client();
    clients[i].start();
  }
  for (int i = 0; i < N; i++) {
    clients[i].join();
  }
  System.out.println("DONE.");
}
```

- For $N = 1$ it executes in about 1 sec.
- What is the expected total response time as a function of N?

Performance Fix

```
class FixedServer extends Thread {
 class Worker extends Thread {
  private Socket s;
  Worker(Socket s) { this.s = s; }
  public void run() {
   try {
    Thread.sleep(1000);
    s.getOutputStream().write("OK\n".getBytes());
    s.close();
   } catch (Exception e) { e.printStackTrace(); }
  }
 }
 public void run() {
  try {
   ServerSocket s = new ServerSocket(8081);
   while (true) {
    Socket client = s.accept();
    new Worker(client).start();
   }
  } catch (Exception e) { e.printStackTrace(); }
 }
}
```