Program testing

Projecte de Programació (PROP)

Autumn 2008
Program testing

• Process of execution and evaluation, either manual or automatic, of a computer system, with the following goals:
  - Elimination of the errors of specification, design and implementation
  - Verification of the functional requirements (what functionality is missing? Is this what the user wanted?)
  - Verification of the non-functional requirements

• Cost: up to 50% of the time!
Program testing

Updated software

Debugging

Errors

Test

Evaluation

Results

Reliability Model

Error rate

Test configuration (Test Set)

Correction

Software configuration to test

Error rate

Reliability Estimation
Program testing

• Detection vs. Correction (+ location)
• Detection -> Methods:
  - Formal verification (*proving*)
  - Tests (*testing*)

"Program testing can be used to show the presence of bugs, but never to show their absence" - Dijkstra
"Be careful about using the following code -- I've only proven that it works, I haven't tested it." - Knuth
"Test your software or your users will" - Hunt & Thomas
Formal verification

- Formal demonstration that the program satisfies the specification

Drawbacks:
- Complex techniques. Qualified staff
- It does not guarantee the absence of specification errors
- You rarely have a formal specification of the rest of the components of the system (libraries, input/output, ...)
- Use in critical pieces of the program and/or in parallel with the design
Testing

Input state

Program (predicate transformer)

Output state

Ideal: Battery of executions to test all the input states

Impracticable!

Testing: Choose a representative subset of all the states
Testing (2)

• The most used and cheapest technique
  - Programmer ability
  - systematic

• It does not guarantee the absence of errors of
  - specification  Manual monitoring
  - design
  - implementation  Systematic test

• It can be more or less exhaustive, according to the importance assigned to reliability
Testing (3)

Concepts:

- **Test case** (one test)
- **Test suite** (one test set)
- **Test plan** (one collection of test sets)
- **Scenario test** (one task that a hypothetical user may want to perform)

Task: Design the previous ones in order to cover *all* (???) the execution possibilities

Input: Files, DBs, values for parameters/cts., values to introduce interactively (including peripherals manoeuvres), ...

Predefine also the expected results!
Program testing

Levels:
• Unit/Component tests
• Integration tests
• System tests
• Acceptance tests
Unit tests

In O.O.: unit class

Techniques:
- White box
- Black box
- Internal structure
- Functionality
Unit tests: White box

- Necessary the access to the code of the component
- **Control flow** method:
  - Battery to provoke the execution, at least once, of each program instruction

  - **Options**: each branch of the “if” must be executed
  - **Loops**: execute limit cases (execute the loop or not) and interior
  - **Data structures**: internally (complete test according to its casuistry)

- More general: each possible path.
- More general: each path with different values
Unit tests: White box (2)

```java
{ 
    s1;
    while (c1) {
        if (c2) s2 else s3;
        s4;
    }
    if (c3) s5 else s6;
}
```

*Basis set method:*

1) Choose a set of paths that cover all the edges

2) Put a *test case* for each path
Unit tests: White box (3)

Data flow method:
- Locate what points of the program assign a value to a variable and which ones use it
- For each variable x define sets of instructions Def(x) (where it is assigned) and Use(x) (where it is read)
- For each instruction s, define Def(s) and Use(s) from the Def(x) and Use(x)
- Define DU chains (Def-Use):
  - (x,s1,s2) if x is in Def(s1) and in Use(s2) and the definition of x in s1 is still valid in s2
- Necessary to cover at least once each DU chain
Unit tests: Black box

- Complementary and normally after white box
- Only the interface of the component is used
- Types of error:
  - incorrect or not expected functionalities
  - interface errors
  - external D.S. errors
  - operation errors
  - initialization and completion errors
- Minimally complete test sets:
  - Divide the domain of each variable into “equivalence classes” (uniform behaviour - subjective...)
    - Ex: \( x \in \mathbb{N} \)
      - \( \{0\} \)
      - \( \{x | 0 < x < 7\} \)
      - \( \{x | x \geq 7\} \)
- Choose a representative of each partition
- Test suite: all the combinations of representatives (quotient set)
- Attention to boundary values
Concurrence / real time

Necessary to test, besides:
- Temporal sequence (*timing*) of data
- Parallelism / interaction among processes
- Deadlock and competition problems among processes/resources
- Reaction to external events/ relation with the execution environment
- Load distribution
- Error recovery (hardware, other programs, ...)
- Operative systems, production systems, networks, ...
- Error control code gets to be 70%
Testing strategy

Phases:
- Test planning: order of test of the units
- Design: cases to test and expected results
- Test: execution
- Obtaining results
- Evaluation
Integration testing

- Interactions among components are tested
  At testing level: not tested comp. <-> not built comp.
- Incremental integration: Not “Big Bang”
- Also “white box + black box”
- Approaches:
  - Bottom-up: Drivers
    Program that tests a module of a “lower level”
  - Top-down: Stubs
    Program that allows to test other module of a “higher level”

(Levels depend on the different types of relationships among components used)
Integration testing: Drivers

Drivers must be interactive!!

It must not be necessary to recompile to test with new data!

Input by keyboard of plain text files!

Attention:
Integration testing: Stubs

**Prop**

**Module to test**

Invoke methods

**Return results**

**Stub**

---

**Stub**: it represents a module $C$ with known interface but still not executable.

It implements a simpler version of the final functionality of $C$.

**E.g.**

- Write “method $m$ from $C$ has been called with parameters $x$”
- Return constant data
- ...
Particular case: Abstract classes

The specific stub implements the abstract methods from C in a simple way.

It allows to create instances of C to test the implemented methods in C (eventually by means of other stubs and drivers)
Integration testing

• Approaches:
  - Bottom-up: *Drivers*
    - The program does not exist until the end
    - Lower level modules go through more tests
  - Top-down (depth/width): *Stubs*
    - Construction of stubs
    - We already have the entire program - overlapping with validation testing
  - Hybrid methods: *Sandwich Testing*
    If it is better to start by testing the most critical modules:
    • Reuse
    • Visibility of the product
    • Degree of relation and use of the different modules
    • Acquisition of data from the user
Integration testing

• Stubs, drivers and test suites are the support (*scaffolding*) to build our system

• They will be taken out in the end

• But spending time designing them is NOT wasted time

• Write much more code than the one that will be in the final version
Tests

Advice:

• Test soon: you will save time and €
• Write tests that evaluate well-defined properties
• Write tests that are comprehensible
• Document tests (test set description!)
• Write tests before the code
• Write a lot of code to perform testing
Regression

• Parts that used to work do not work anymore
  - When we add functionality
  - When we redesign ("improve") a part

• Bugs supposedly eliminated reappear
  (*regression bugs*)

• Specially, in maintenance tasks
Regression (2)

- *Regression test:* Execute all batteries already passed in previous phases when changes are introduced

- ... in each “build”, or each night/week

- Important: add test cases for each bug already eliminated!

- Semi-automatic tools for testing
Validation/system testing

Instruments
• Complete product
• Specification of functional and operative requirements
• User documentation

System testing (real environment):
• Security
• Recovery (system failure)
• Power (extreme conditions: volume, frequency, rush hours, number of users)
• Efficiency (response time, resource consumption, ...)
• Interaction with other software
• Usability...
Acceptance test

- Performed by the client/end user
- Supervised acceptance tests:
  - alfa: with attendance of the programmer
  - beta: without attendance of the programmer

Until when can I test?
- Until money or time are over!
- When #detected errors/unit of time tends to 0
- End product = Pre-alfa version of the following product
Error debugging

Phases for each error:

- Detection (signs)
- Location (causes)
- Determination of the phase in which it has been produced (specification-design-implementation)
- Evaluation of the seriousness (minor-serious-critical) and of the cost of correcting it
- Correction
- Test (→ regression tests)
Error debugging (2)

Some heuristics:

• Brute Force Approach: Systematic monitoring of the performance of the program
• Backtracking: Go back from the point where the sign has been detected. Problem: side effect?
• Locate points of the program where the causes of the sign could happen: review variables involved and their assignments
• Check of contracts of the calls (pre and postconditions)
• Java’s Assert / PRAP’s InOut library
• Regression: fix a bug has generated others?
Experience

Advise:
- An error must be useful to avoid repeating it
- Reflect: why did I made this mistake?
- Share it with the rest of the group!

"Experience is the name everyone gives to his mistakes" - Oscar Wilde, Lady Windemere's Fan

"Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so" - Douglas Adams, Last Chance to See

"Experience is what you get when you don’t get what you want“ - Dan Stanford