Ingegneria del Software
Corso di Laurea in Informatica per in Management

Software testing

Davide Rossi
Dipartimento di Informatica – Scienze e Ingegneria
Università di Bologna

Validation and verification

- Software testing is part of verification and validation activities
- Verification: evaluation of artifacts (meeting, reviews), software testing
  - Are we building the system right?
- Validation: software meets the expectations, acceptance testing
  - Are we building the right system?

What is software testing about?

- *If you don’t care about quality, you can meet any other requirement*. [G. M. Weinberg]
- *Testing shows the presence, NOT the absence of bugs*. [E. W. Dijkstra]
- Software testing is about revealing as many defect as it is reasonable

Testing and software

- Testing in software cannot, in general, make use of results gathered from other engineering disciplines.
  - A bridge tested for 1000 tons will surely handle anything inferior, programs do not benefit from this kind of continuity properties.
- *The number of states in software systems is one order of magnitude larger than the number of states in the nonrepetitive parts of computers*. [D. Parnas]

Testing and costs

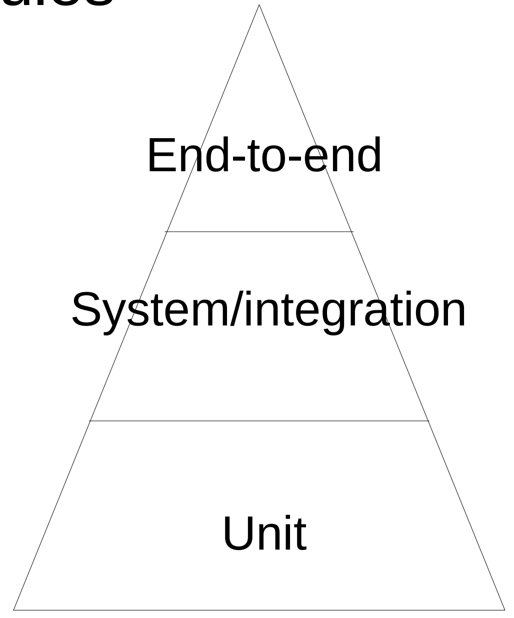
- The cost for correcting defects in software system is proportional to the time the defect is laying in the system (there are graphs and figures in the literature, albeit most are just wrong).
- As a consequence we want to **test early and often**.

Jargon

- Defect (bug)
  - The result of a mistake
- Failure
  - Happens when the mistake is exposed
- Issue
  - Describes the failure
- Test case
  - Describes the expected outcome of a run, it includes data, (pre/post)conditions and (expected) results
- Test set
  - A collection of test cases

Testing levels

- Unit → Class/method
- Integration → Group of software modules
- End-to-end (E2E) → The whole system



Testing pyramid

Static/dynamic testing

- Static testing
  - The code is **analyzed** in order to find bugs
- Dynamic testing
  - The code is **executed** in order to find bugs

Static testing

- Mostly based on formal methods approach
  - model checking
  - data-flow analysis
  - abstract interpretation
  - symbolic execution
- Noteworthy tool: Polyspace (marks code on the basis of static analysis: reliable code, faulty code, unreachable code, unproven code, code violating rules)
- Human reviews are a kind of static testing too.

Software quality tools

- Use *bug patterns* to assess the quality of code
- Mostly simple rules to check adherence to safe coding styles
- Some static analysis rule (e.g. non-reachable code, collection added to itself, ...)
- Examples: FindBugs/SpotBugs, SonarQube, ...

Black-box testing

In black-box testing functionality is examined without any knowledge of internal implementation (*what*, not *how*)

Test design techniques:

- Boundary value analysis
- Equivalence partitioning
- Decision table testing
- All-pairs testing
- State transition tables

Boundary value analysis and equivalence partitioning

- Find the discontinuity points in the input values
- Test them (at that point and, if it makes sense, right before and right after that)
- Test one random sample for the intervals around discontinuity points and in domain validity intervals

Boundary value analysis example

`discounted` receives the order amount and returns a discounted value applying a 5% discount if 1000<amount<5000 and 10% discount if amount >= 5000

- Discontinuity values for the input:
  - 1000, 5000.
- Tests:
  - 1000, 1001, 4999, 5000, 50001

Equivalence partitioning example

`discounted` receives the order amount and returns a discounted value applying a 5% discount if 1000<amount<5000 and 10% discount if amount >= 5000

- Partitions:
  - [INF, -1] because of input domain
  - [0,1000][1000,5000][5000,SUP]
- Possible values:
  - -10, 100, 3000, 8000

White-box testing

The internal structure of the code is used to define the test cases.

Testing methods include:

- Data flow
- Control flow - identify a test set that allows to achieve (*full*?):
  - Code coverage
  - Branch coverage
  - Path coverage

Coverage example

Code coverage

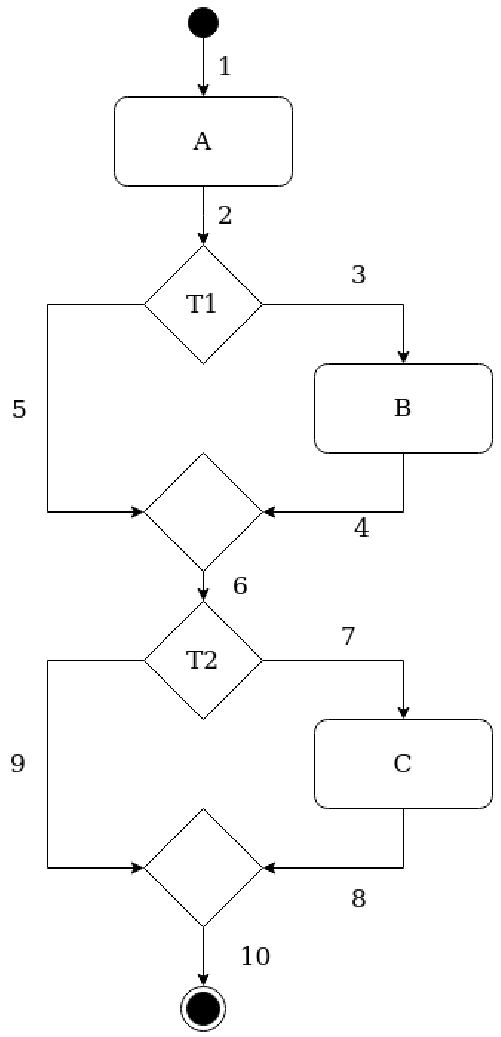
```
1;A;2;3;B;4;6;7;C;8;10
```

Branch coverage

```
1;A;2;3;B;4;6;7;C;8;10
1;A;2;5;6;9;10
```

Path coverage

```
1;A;2;3;B;4;6;7;C;8;10
1;A;2;3;B;4;6;9;10
1;A;2;5;6;7;C;8;10
1;A;2;5;6;9;10
```



White vs Black pros and cons

- White pro: knowledge on the code is acquired while building the test cases
- White pro: higher coverage
- White con: complex
- Black pro: testers who are not coders
- Black pro: closer to requirements
- Black con: unknown coverage

Test the tests

- How do I assess the quality of my test set?
- Mutation testing
  - Create mutants of your code
  - Run your tests on the mutants
  - If a test passes you have a problem
- (Actually usable) mutation testing for Java: PIT

Structure of a test

Tests, regardless of their level, are structured around three stages

1) Put the SUT in a desired state
2) Interact with the SUT
3) Verify that the results of the interactions are the expected ones

A useful mnemonics: **AAA** – Arrange / Act / Assert

Unit testing

- In unit testing single units of code (functions, methods) are tested.
- Unit testing is used to ensure that code meets expectations and that code continues to meet expectations (***regression testing***).
- Units have to be tested in isolation.
- The test set for each unit contains independent cases.

Unit test expectations

- Return value
- State
- Collaboration with other objects (*behavioral testing*)

Isolation?

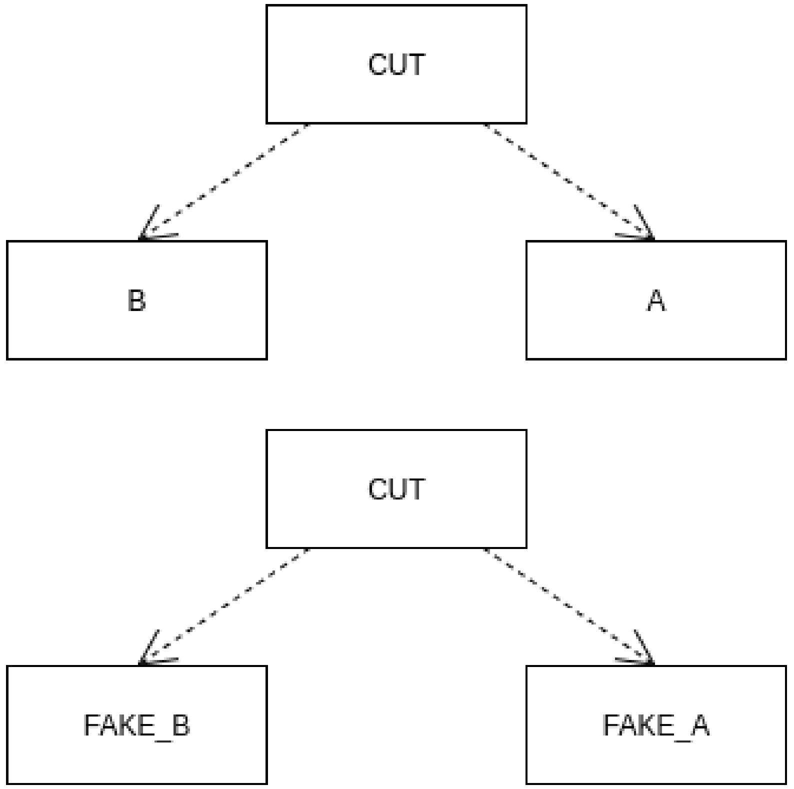
- How do we untangle the code from its dependencies?
- Use *test doubles*.
- They provide the same interface with alternate code.
- Dependency injection greatly simplifies isolation.

Test doubles

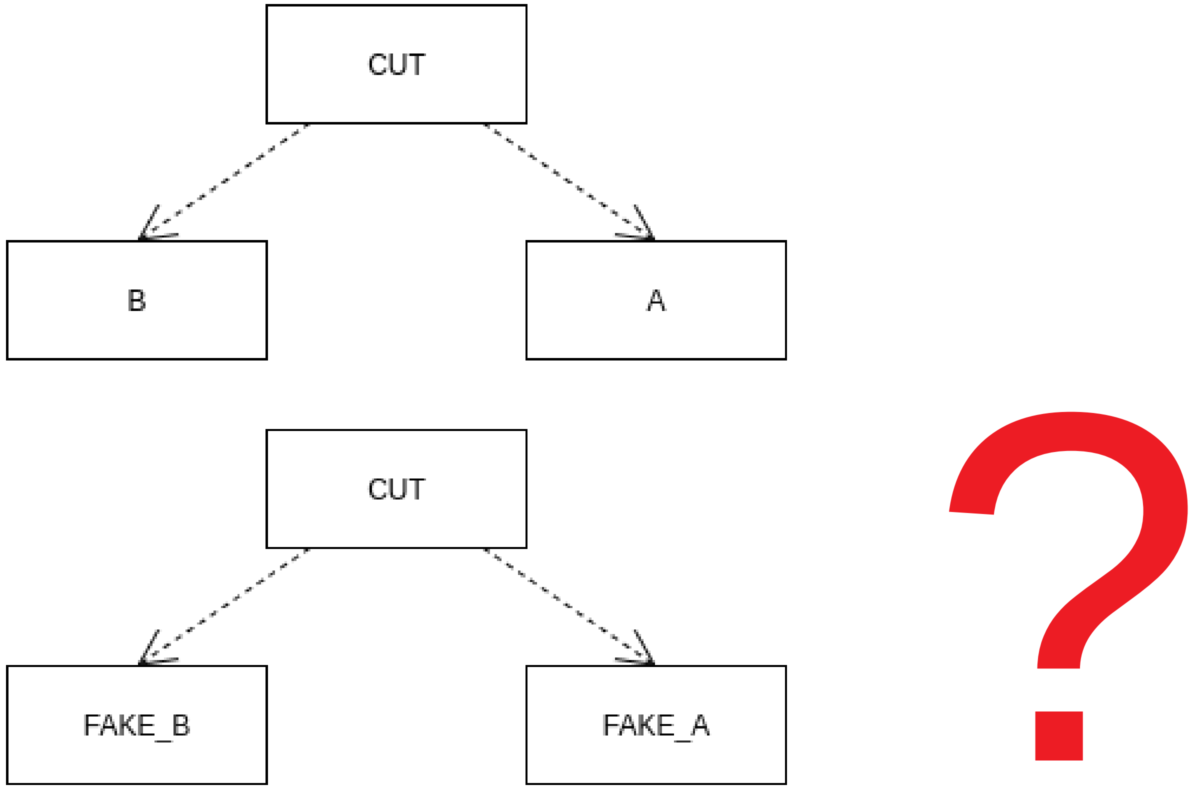
Test doubles replace the collaborators (dependencies) of an object to improve isolation and/or to check collaborations.

- Dummy – unused parameters
- Fake – simplified working implementation
- Stub – provide hard-wired responses
- Spy – stub that record interactions
- Mock – double with expectations (about calls it receives)

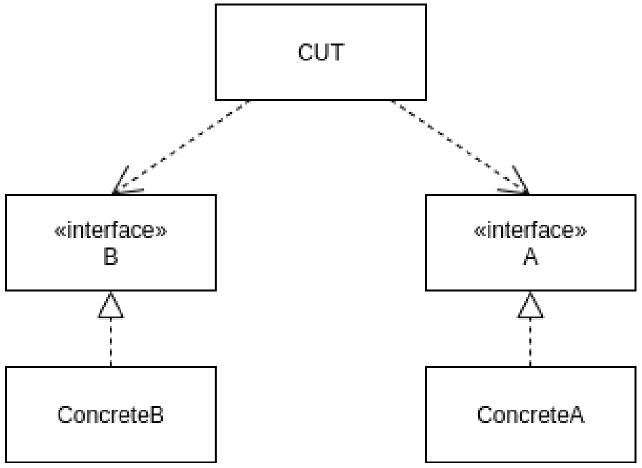
Test double example

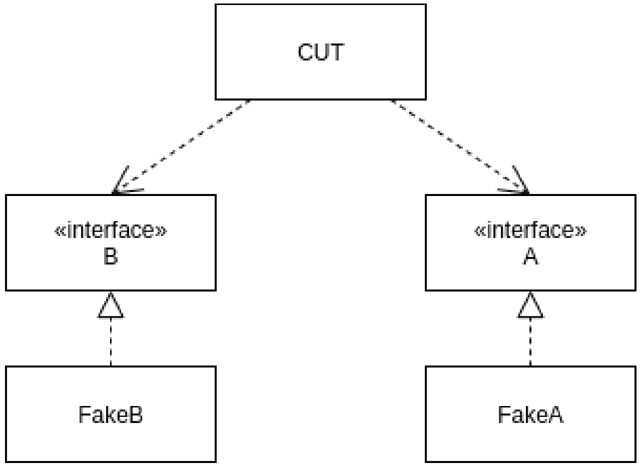


Test double example



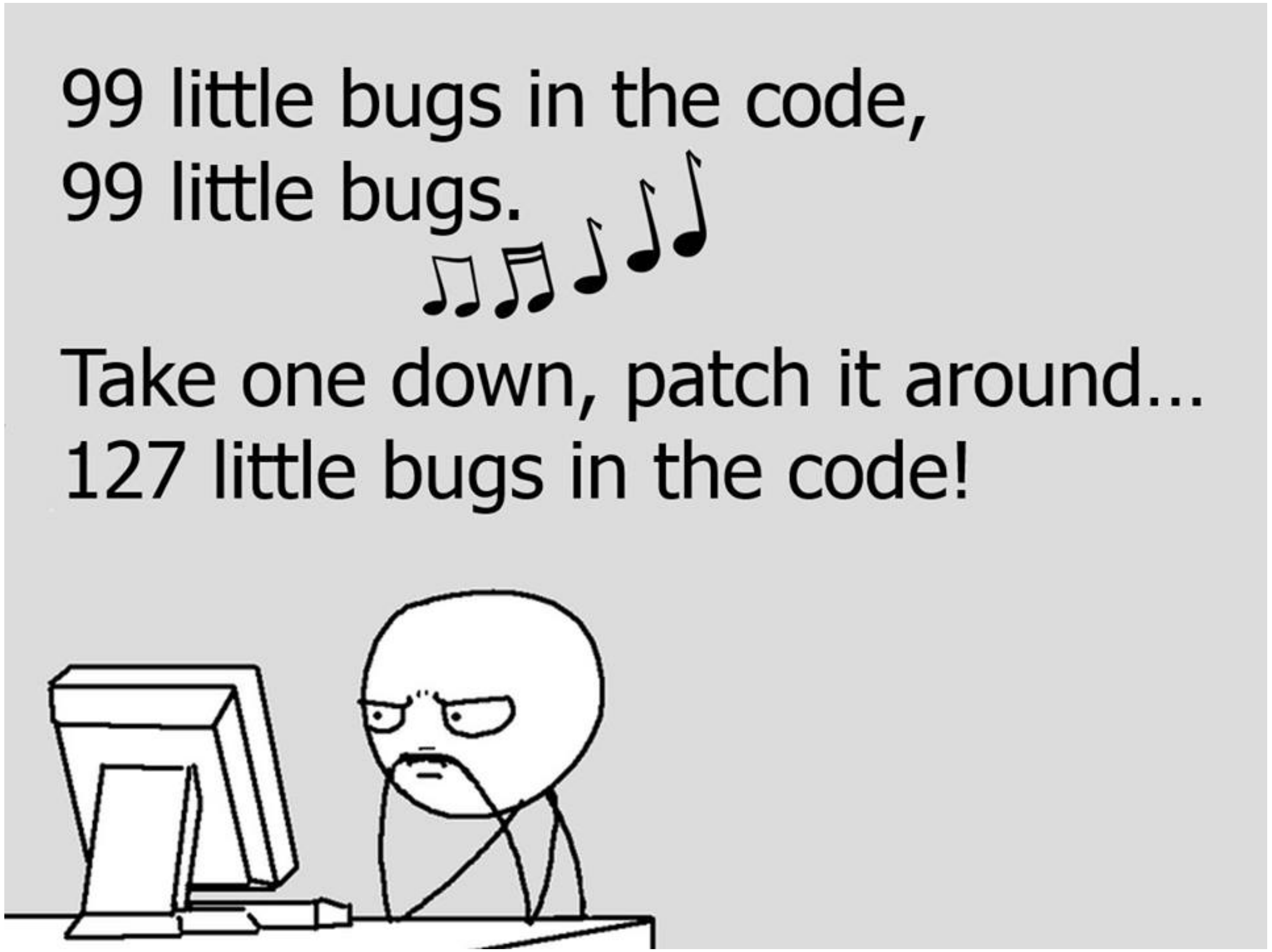
Test double example





XUnit

- A framework for unit testing originally designed by K. Beck.
- Most widely used implementation: JUnit [E. Gamma]
- The components of XUnit’s architecture are:
  - Test runner
  - Test case (uses assertions – one per case)
  - Test fixtures (or contexts)
  - Test suites
  - Test execution


99 little bugs in the code,
99 little bugs.
Take one down, patch it around…
127 little bugs in the code!