10 Verification

• All changes in the code have to be verified
  – refactoring
  – actualization

• Essential difficulties
  – programmers very often produce imperfect work
  – defects are called “bugs”

• Verification finds bugs
Verification techniques

• Many techniques of software verification have been researched and proposed
• Current practice
  – testing
  – code inspection
Testing

• Tests execute the program or its parts
  – specific input to the execution
  – compare the outputs of the execution with the previously specified outputs
  – report if there is a deviation

• Tests are usually organized into a test suite
  – several, often many, tests.
Incompleteness of the testing

• “Testing can demonstrate the presence of the bugs, but not their absence. “

• No matter how much testing has been done, residual bugs still can hide in the code
  – they have not been revealed by any tests
  – no test suite can guarantee that the program runs without errors
Turing’s *halting problem*

- Theoretical reason for testing incompleteness
- It is theoretically impossible to create a perfect test suite
- The programmers have been trying to do the best under the circumstances
  - techniques of the testing cannot guarantee a complete correctness of software
  - well designed tests come close to be adequate
New vs. old code tests

• Tests of the new code
  – new tests must be written with the new code

• Testing of the old code
  – the tests make sure the old code is not broken by the change
  – regression tests
  – prevent regression of what was already functioning in the software.
    – Merriam Webster: “regression” = “a trend or shift toward a lower or less perfect state”
Variety of software testing

• Setting
  – programmer’s workspace
  – team’s configuration management

• Strategy
  – structural
  – unit
  – functional

• Functionality
  – old (regression testing)
  – new
  – Combined (system testing)
Acceptance tests

• Final functional test
  – both the new and the old functionality
• Done during the phase of change conclusion
  – test the complete functionality of the software
  – software stakeholders are able to assess the progress of the software project
Composition of the test suite

• Unit tests
  – test for a specific class

• Functional tests
  – test a specific functionality of the whole program
    • user manual or graphics user interface guide the creation of the functional tests.
    • all features that are available to the user should be tested

• Structural tests
Harness (scaffolding)

- **Test drivers**
  - implement the support for the tests

- **Stubs**
  - implement the replacement for missing classes and subsystems

- **Environment simulation**

- **Harness = drivers + stubs + simulators**
  - production code vs. harness
  - developers vs. testers
Harness and production code

- Production code goes to the user
- Harness stays within the programming group
- Parallel evolution of both

© 2012 Václav Rajlich  
Software Engineering: The Current Practice  Ch. 10
Coverage

• We cannot guarantee a complete correctness of the code by testing
• We are going to guarantee that each unit of the tested code is executed at least once
  – this guarantees that at least some of the bugs are discovered
    • in particular, the bugs that are brought up by this single execution of the unit.
Granularity of methods

• calcSubTotal(), calcTotal(), getPrice(), setPrice(double)
  – each of them executed at least once
• Seems like a crude approach
  – systematic
  – guarantees correctness better than random selection of tests
Statement coverage

• Guarantees that every statement of the program is executed at least once

• *Minimal test suite* does not have any redundant tests
  – tests that cover only statements that are already covered by other tests

• Minimal test suite efficiently accomplishes the coverage
  – preferred approach
Example

read (x); read (y);
if x > 0 then write ("1");
else write ("2");
end if;
if y > 0 then write ("3");
else write ("4");
end if;

\{<x =2, y =3>\}
- incomplete coverage
- does not cover write ("2")
\{<x =2, y =3>,<x =9,y =1>,<x =0,y =0>\}
- complete coverage
- not minimal
- \(<x =9, y=1>\) is redundant
\{<x =2, y= 3>, <x =0,y =0>\}
- complete and minimal
Minimal complete test suite

• Easy to write the first test
  – no matter what it covers, it adds to the test suite coverage

• Increased number of tests
  – the statements not covered are fewer
  – it is increasingly hard to aim new tests at these remaining uncovered statements
  – increased coverage may not be economic

• 70% coverage is considered to be good
Unit testing

• Testing of individual modules
  – testing classes and methods

• Testing of the composite functionality
  – the class is being tested together will all classes that support it

• Testing local functionality
Testing driver

```
TestItem

Item
+calcSubTotal() : double
+calcTotal() : double

Price
+getPrice() : double
+setPrice() : double
```
public class TestItem {
    Item testItem;
    public void testCalcSubTotal() {
        assert(Item.calcSubTotal(2, 3) == 6);
        assert(Item.calcSubTotal(10, 20)==30);
    }
    public void testCalcTotal() {
        assert(Item.calcTotal(0, 5) == 5);
        assert(Item.calcTotal(15, 25) == 40);
    }
}
Testing local responsibility

• Driver + stub
  – stub simulates suppliers
    • part of harness

• Reasons: supplier classes
  – are not available
  – have not been tested
    • the confidence in them is low
  – support a limited contract (limited precondition)
    • the tested class to be used with other suppliers
Stubbing techniques

• Less effective algorithm
  – easier to implement
  – test becomes less efficient
    • developers do testing, acceptable impact

• Limited precondition of the stub
  – simplifies the code of the stub substantially
    • convert the date into a day of the week
      – the stub does that only for a selected month
    • inappropriate if the stubbing reason is to broaden the contract
Stubbing techniques, cont.

• User intervention
  – interrupts the test, the user provides the correct answer
    • practical only in if the stub is executed only few times during the test
    • human user may input incorrect values

• Replacement contract
  – quick but incorrect postcondition
  – the most controversial stubbing technique
  – still may provide valuable results
Functional testing

• Tests the functionality of the complete program
• Program with GUI: test every function
  – “tape recording” for future tests
• Coverage
  – percentage of the requirements tested
Regression Testing

• After change, programmers retest the code
  – reestablish the confidence that the old functionalities of the software still work
  – change may have inadvertently introduced stray bugs into the intact parts

• Tests from the past constitute the bulk of the regression test suite
  – test suite often grows
  – testing is often done overnight
Test suite evolution

- System test
- Regression test

- Add tests of new functionality
- Delete obsolete tests
Obsolete tests

- Broken test cases that cannot run
  - they do not interface with the software any more
- Tests that do not fulfill their purpose
  - a test case testing the limits of a range becomes obsolete when the range is changed
- Tests that no longer provide a deterministic result
  - a test case which may now impacted by the mouse
Code inspection

• Somebody else than the author reads the code
  – checks its correctness
  – reports bugs

• Code inspection does not require execution of a system
  – can be applied to incomplete code or to other artifacts
    • models, documentation
Effectiveness of code inspection

• “Habituation”
  – people become blind to their own mistakes

• After reading the code several times
  – programmers no longer read the code
    • recall from the memory what the code should contain
    • some errors repeatedly escape their attention
  – a different reader spots these errors easily and right away
Inspections and testing are complementary

• Some bugs are easily found by testing
  – they appear in each test
    • they are sometimes hard to spot by human
    • example: misspellings of long identifiers

• Some bugs are hard to find by testing
  – it is hard to create a test that finds them
    • human readers can find them easily
    • example: a possible division by zero
    • to create a test that causes such situation can be hard
Inspection of different artifacts

• Inspections can also check whether different artifacts agree with each other
  – does the code correspond to the change request?
  – does the UML model correspond to the actual code?

• Inspections cannot check non-functional characteristics
  – performance and usability
Walkthroughs

• Structured inspection process
• Walkthrough team
  – at least four members
    • the author of the inspected code
    • a moderator
    • at least two code reviewers.
Walkthrough process

• Preparation
  – the code or other work products are distributed to inspection team
  – one participant (reader) is selected to inspect the document thoroughly

• Walkthrough meeting
  – all team members participate
  – whole group walks through the document under direction of the reader
Walkthrough meeting

• The reader notes the errors, omissions and inconsistencies in the code
  – the other members of the team add their own observations
• The moderator chairs the meeting
  – notes the discovered errors
  – produces a report
    • recommendations, which documents have to be corrected or reworked