6.3 Error seeding

- Conscious adding of errors to the source code
  - One can try to estimate the number of "real" errors in the code based on the number of seeded errors found
  - Evaluating the effectiveness of testing
- Let's mark with $V_R$ the number of real errors and with $V_S$ the number of seeded errors and $V_{RF}$ and $V_{SF}$ mark the number of found real and found seeded errors
- Additionally it is assumed that $V_R / V_{RF} = V_S / V_{SF}$
- The number of errors that remain is $(V_{RF} \times (V_S / V_{SF})) - V_{RF}$

Problems of seeding:
- What kinds of errors should be seeded?
- Have all the errors been removed that were added?
- How much extra work is caused by the method?
- This technique can also be used in document inspections
  - One can measure how well the document has been inspected
  - Requires much from the organization
Mutation testing

- A special case of error seeding
- Different versions of the program are produced each having a different error seeded
- Can be used to
  - Assess the effectiveness of testing (how many mutants were detected)
  - Test how well the program is able to recover from errors
- Earlier the generation of mutants has been used also for generating project assignments in the TUT testing course!

Pablo’s error seeding machine

- C++ program that gets as an input another C++ program and generates a mutant out of it
- The errors planted to the mutant are found by comparing it to the original program (in the Unix environment for example with the diff command)
- Defines for example sets of types and operators whose units are interchangeable
  - For example {+, *, /}: if the program has one operator out of the set it is changed to some other operator in the set
  - For example (short, int, long): the type is changed
Can also add and remove continue and break statements in loops

Problem: Not all produced mutants are legal C++ programs
- Solution: Call gcc and only those programs that pass the compiler are good enough to be mutants

A distributed seeding machine was able to produce 40 000 mutants in a short time, so every project group had their own

Excursion:
How to find a bug

- Adapted from [Kaner et al. 02]
- New things: Newest features might work wrong
- New technologies: New concepts lead to new errors
- New markets: Different users use the program in different ways
- Change: The changes might break a code that worked earlier
- A late change: Quick decisions and busy people lead to errors
- Busy work: When there is not enough time budgeted to the project, the quality of the work suffers
Learning curve: Errors of carelessness
Bad design or a system that is hard to maintain: Because of some design decisions the system is so hard to maintain that fixing problems results regularly in other faults
Tired programmers: Overtime for several weeks results in ineffectiveness and errors
Other issues in personnel: Personal problems, family issues, problems in the working community (if two programmers do not communicate with each other, probably neither does their code)

Just slipping it in: A developer’s favorite feature added to the software without the knowledge of the project management does not necessarily work with other parts of the code
Not invented here: External components can cause problems
Not budgeted: Tasks that are outside of the budget are done carelessly
Ambiguity: Ambiguous specifications can lead to errors or conflicted implementations
Conflicted requirements: Ambiguity covers a conflict
A moving target: The customer comes up with the real requirements only when the product is already being developed
• Number of bugs: Features that have a lot of known bugs can also contain a lot of unknown bugs
• Dependencies: Faults can cause other faults
• Untestability: It is hard (almost impossible) to test the system
• Minimal unit testing: The programmers should find and repair most of their own errors
• Previous concentration to narrow testing strategies: Regression and functional testing can leave the software with large number of errors that remain from one version to another
• Weak testing tools: If proper tools are not used for e.g. finding pointer errors, these errors can easily go undetected

Typical errors in programming languages: Examples in C

• A forgotten end symbol for a comment:
  a=b; /* comment begins
  c=d; /* Another comment */
• Assignment vs. comparison:
  if(a=b) ++c;
• Local variables have to be initialized:
  void foo(a) {
    int b;
    if (b) {}
  }
• What is the semantics of the statement i = i++ ?
• Fortunately, the weak parts of the C language are well documented and smart compilers warn about suspicious parts in the code
Typical errors for programming languages: examples in C++

- Although there is a Boolean type in the language, implicit conversion from bool->int is allowed:
  
  ```cpp
  if (-0.5 <= x <= 0.5) return 0;
  ```

- Function declaration and calling of a default constructor get often easily confused:
  ```cpp
  int main() {
      string a("Hello");
      string b(); // Function declaration...
      string c = string("World");
      // ...
      return 0;
  }
  ```

Typical errors for programming languages: examples in Java

- In older code the missing enumerator type (enum) has led to strange detours
- Not knowing which class files are loaded (jar, CLASSPATH, a period in CLASSPATH)
- All variables that refer to objects are actually implicit pointers:
  - myObject a, b; // variables are initialized to the value null
  - a = b; // copying of pointers
  - a = new myObject(b); // creation of a new object
7. Agile testing

*There is life outside of the V-model – one of the clearest trends in testing is the shifting towards more agile and less plan-driven processes.*

- In traditional software processes that follow the waterfall model, testing is done according to the V-model
  - The left side of the V-model describes the waterfall model, which goes top-down from requirement specification to functional specification and from there to architectural design and at the end to module design and to the implementation
- In each phase a test plan is made for each phase of testing (right side)
- When there exists some implementation, the testing can begin
  - The right side of the V-model bottom-up according to test plans
– In every level it is tested whether the implementation corresponds to the specifications
• The techniques used depend on which phase is currently active
  – For example, testing on the lower levels is usually more like white than black box testing and on the upper levels the other way around
• Traceability between different phases eases the tracking of the origins of errors
• When an error is detected, the testing process can be improved so that the errors of that kind can be detected earlier

• Often requirements are only discovered during the way
  – Everything cannot be known at the start of a project and changes are something that should be prepared for
• The traditional interpretation of the V-Model does not support iterative software development, where there are intermediate versions of the program
  – The goal is to create the whole system within one iteration
  – The phases of the model are repeated only when developing the next version of the system
• The model makes assumptions on what is documented, which documents the test cases of different levels are based on, and in which phase the created test cases are executed
Agile unit testing: Test-Driven Development

- A way to think, plan, communicate and write code
  - The goal: simple, clear and secure software
  - Ideas of eXtreme Programming
- As a side effect a wide collection of automated unit/integration level tests are created
- Not like this:
  - stress -> test less -> more errors -> more stress
- But like this:
  - stress -> testing -> fewer errors -> less stress

The method in short:
- Write a test
- Code the implementation
- Clean implementation (re-factoring)
- What about other modeling, design, documentation, implementation and testing methods?
  - They are used if necessary, if they help to create a working software
• Principles:
  – Willingness for change
  – Presence of a client
  – Metaphors: concepts of the project
  – Design game
    • User stories ≈ the requirements, use cases
  – Stand-up meetings
    • A daily meeting lasts only so long as the people can stand

– Simplest working solution is selected
– Pair-programming
  • Circulation of pairs
– Coding standards
  • Style guides etc.
  • From the code one should never be able to tell who wrote it
– Joint ownership
  • Everyone is responsible for all the code
– Continuous integration
  • Problems are noticed immediately and not after a week
– Cleaning of the code
  • Also test code must be cleaned
– Gradual delivery
  • Small increments
– In the limits of human capabilities
  • 40 hour work week

• Tests
  – The goals:
    • Achieving trust in the functionality, planning, design
  – Cycle:
    • Write a test, compile, run tests
    • Write code, compile, run tests
    • Clean implementation, compile, run tests, integrate, compile, run tests, check in the version control, write a test, ...
Also the test code must be cleaned up from time to time
The goal is to have as much test code as the code to be tested
• If there is less test code:
  – Something is probably left untested
• If there is more test code:
  – There is probably some redundancy in the test code (too much time spent on running the tests)

Questions
• How large should the tests be?
• What does not need to be tested?
• How do you know if tests are good?
• How many tests must there be?
• When should a test be removed?
• What is the effect of the programming language and the environment?
• Can large systems be made with test driven design?
• How is TDD deployed in the middle of a project?
• Testing of embedded systems?
• TDD as a whole
  – TDD divides a problem into smaller fragments and to the responsibility of the whole team
  – Can clever design, continuous integration and cleaning of the code make coding and testing feel like creative work that proceeds steadily?
  – In a work that proceeds in small steps, finding an error is much easier
    • A separate phase for cleaning gives a possibility to concentrate on one thing at a time
  – There has to be self discipline: tests must be written first instead of just testing the functionality
  – Not suited for all projects
  – May have negative effects on the software architecture

Continuous integration in practice

• Based on article Martin Fowler & Matthew Foemmel: Continuous Integration, www.martinfowler.com
• The requirements of continuous integration:
  – The version control has to be in order, the source code resides in one location, where it can be obtained from by anyone if needed
  – Creating a new build has to be automated so that anyone can make it with one command
  – Automated test cases should also be executable for the build with one command
  – The result of a successful build can be obtained by anyone
• What is the benefit of continuous integration?
• Let’s assume that two developers code their own components that should work together
  – Both test their own components and find no errors in them
  – When the components are integrated, it is noticed that they do not work together as they should
  – If there is a traditional process in use, finding the error may be really difficult if the code has been written weeks before
  – In continuous integration, the error has probably been made on the same working day, or the previous one, so finding it is much easier, because there is less new code and the developer still remembers what he has done

• How often should one then integrate?
  – Due to automation as often as is wanted, at least once a day
• What is a successful build?
• Build is successful if the following tasks are accomplished without errors or adjusting of the automation:
  – All the latest files are obtained from the version control
  – All files compile OK starting from clean
    • If compiling takes a long time, one can compile only the changes which means the full compilation is done only from time to time
Object files are linked to the binaries
  - For example Java class files to jar files
  - The program is started and the smoke tests are run successfully

Basic requirement is the working configuration and version control
  - Anyone should be able to plug in a “clean” machine and obtain the sources with one command for the build
  - Version control should include all the files needed in the build, for instance configuration files and scripts

Master build
  - Common build for the team
  - Is done in a centralized server
  - The system monitors the version control and when it notices a new version of the code, it starts building
  - After running the tests, the system sends email about the results to the developers whose code is in the build
    - The developers are assumed to be in readiness to fix their code until they have received a notice of a successful build
  - The system writes a log from the phases of the build, the logs can be observed through a web-page that describes the progress of the projects
  - After this, the system returns to observe the version control and wait for another build to make
Before starting a task the developers get the newest files from the version control
New code can be integrated locally when the whole task is done or part of it has been done, as long as unit tests have been run successfully
At the start of integration, newest versions of the other files are yet again taken from the version control
When a local build has been made and smoke tests run successfully, the developer puts his own code to the version control and waits for the master build
  – When a master build has been created successfully, he/she can move on to the next task

Agile acceptance testing: ATDD

In TDD, low level test cases drive the coding phase
In ATDD, high level test cases drive the whole process
How to define the “Definition of Done” from the perspective of the customer or end-user?
ATDD is based on defining executable system level tests before the corresponding implementation is even started
Ideally, the tests are done by the customer or end-user
When the test passes, the corresponding requirement is considered to be "Done"
ATDD tools enable easy test creation in a form understood by the customer or the end-user
ATDD Process (iterative) 
adapted from Niklas Collin, Kilosoft

- **Benefits**
  - Less ambiguity in requirements, since the customer or end-user has approved the tests
  - When a test passes we can move on to implement the next requirement

- The set of tests defines the scope of the project: if no unpassed tests exists, there is nothing to implement any more (as approved by the customer or end-user)

- Visibility of the progress for the management and the customer

- However, the tools available for ATDD do not necessarily support every domain

...and not all customers are willing to do this!
Agile manual testing

• Exploratory testing
  – The skills and experience of testers play a major role
  – Testers learn all the time new things about the test target, its
    risks, habits, how it has been working wrong in the previous
    tests, etc.
  – New test cases are created and used continuously
    • Not necessarily documented
  – New test cases are better than the old ones, because they are
    based on new information about the system
  – Unlike in ad-hoc testing, the testers have clear goals and they
    often focus in some limited part of the software

  – Exploratory testing can be prepared for example with charters
    • A charter consists of what is to be tested and why, how, what
      type of problems are looked for, knowledge about he risks,
      recommendation for tools, etc.
    – A normal regression test session lasts about two hours
    – The primary result are bug reports, but also other
      documentation, notes, etc. as necessary
    – After a session reporting is done as agreed
Problems

• How to
  – know which tests were done?
  – repeat them?
  – assure accountability?
  – train new explorative testers?
  – combine ET with other testing approaches?
  – get accurate, reliable and unbiased information about the approach?
  – allocate the work inside the team without duplicate work on same areas?

ET Variants

• Freestyle
• Session-based
• Tourist
• Guerrilla
• Usage scenario / Use case / User story based
8. Automation and tools

In this part we introduce test automation and the tools which can help testing. Due to the great number of tools available, the purpose is just to give a general view.

Few examples of the tools of agile testing

<table>
<thead>
<tr>
<th>TDD</th>
<th>Agile acceptance testing</th>
<th>Exploratory testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>System testing</td>
<td>PIT, FitNesse Robot Framework</td>
<td>Office tools</td>
</tr>
<tr>
<td>Integration testing</td>
<td>Continuous Integration (CI) tools</td>
<td></td>
</tr>
<tr>
<td>Unit testing</td>
<td>xUnit (<a href="http://www.junit.org">www.junit.org</a>)</td>
<td></td>
</tr>
</tbody>
</table>

+ different small tools are often helpful in different levels of testing
• The beginning slides are based on the material collected by Mika Maunumaa
• Different views
  – Software design describes how things should be
  – Testing questions if things are as they seem
• Testing has traditionally been handwork
• The promise of test automation is to remove handwork and solve testing problems
  – When a program is testing a program, money and time is saved
  – A program can do the testing without errors, faster and for a longer time
  – All cannot be tested by hand
    • stress and performance testing etc.

8.1 Test automation as a whole

• Test automation is a supplement for manual testing, and not a replacement
  – Tests have to be planned beforehand
  – Only the best tests are automated
  – The role of manual work changes, but is not removed completely
• A program whose purpose is to “run” another program
  – Both are created (partially) from the same requirements
  – There is a big difference in the point of view and the goals
• Manual testing and automation of tests require different kinds of skills
• Test automation is not the silver bullet of testing
Promises of test automation

- Regression testing becomes easier; more tests, more often
- Tests that are impossible to do manually
  - For example stress testing of web sites
- The errors made during testing are reduced, machine is more accurate than man
  - On the other hand new kinds of errors become possible
- More efficient usage of resources
- Repeatability and integrity of the tests
- The tests can be executed on different hardware or software platforms
- The reusability of tests when the test target remains the same
- Confidence towards correctness increases, shortening the time to market

Common problems of test automation

- Unrealistic expectations
  - Tools solve all the problems
- Bad ways of testing
  - "Automating chaos just gives faster chaos"
- Automation is expected to find a lot of new errors
- False sense of security
  - As automation did not find errors, there are none
- Maintenance of automated tests
- Technical problems
  - Buggy testing software, compatibility
- Problems in the test organization
  - Lack of support from the management, the culture of the organization, training
Limits of automation (1/2)

- Does not remove the need for manual testing
  - Not everything should be automated
  - Tests ran rarely
  - The software to be tested is a “moving target”
  - The results of a test are easy to be interpreted by man but very difficult for a machine (for example the quality of sound or picture)
  - Tests that require human activity
  - There is no need to automate everything
    - Only the best and most often repeated tests
  - Testability of the SUT needs special emphasis, the lack of it may result in failure

Limits of automation (2/2)

- Manual tests find more errors
  - An error is found first with a manual test that is then automated for regression testing
  - James Bach reports his experiences from Borland: automation found only less than 20% of all errors that were discovered during the project although there had been investments to automation for many years (James Bach, Test Automation Snake Oil, 1999)
    - The new features contain more errors than the old ones
    - A manual tester can find the new bugs while automation is only being adjusted to test new areas
• Test automation might limit the software development
  – Tests are sensitive to slight changes in the software
  • Initialization of an automated test may be a more complicated
    operation than in the case of a manual test
  • Maintenance requires work
• Tools have no imagination
  – Tools only do what they are programmed to do – a man can vary
    the test execution and act as an intelligent observer
• Automation does not increase effectiveness
  – The price of a test run and test execution time vs. the price of the
    design and maintenance

**Domains of automation**

• Traditionally in systems based on APIs and protocols
  – The goals are "unambiguous"
  – The interface is usually clear programming interface
• GUI testing has increased as graphical user interfaces have
  become more popular
  – New challenges
    • Input of data
    • The interpretation of the response
    • Digging out the response from the depths of the program
• Testing through the user interface poses a question on where we get the data needed in the comparisons
  – In the worst case the bitmaps captured from the screen have to be compared
    • The change in the color of one pixel can cause a false alarm in a test run
      – If the automation does not recover from errors, a result can be a lot of wasted time
  – Optical Character Recognition (OCR) can help with the comparison of the text fields
    • Although the technique is slow and prone to errors
  – In the best case the resources of the user interface library can be used directly
    • The window knows all text fields in it, whose values can be read directly as strings
    • Usually the GUI automation tools in Windows are based on this

Test automation – a different kind of software engineering

• The implementation of test automation is a software engineering project; the scripts require the following
  – Programming and design skills
  – Testing skills
  – Skills in documentation
  – Maintenance skills
• Who tests the test program?
A test automation project

- Starts often with great expectations
  - The tool shall solve all the problems in testing
  - Automated testing is cheap
  - When an automated GUI test is running on the screen one can easily get the euphoric feeling that the quality of the product increases tremendously – if one does not understand what is really happening

- Often breaks down on the fading of the illusion
  - Automating bad habits did not improve the situation
    - Wrong, bad, or erroneous tests with errors were automated
  - Wrong or incompatible (and expensive) tool was selected
  - There was no preparation for maintenance
    - Changing of the automation scripts requires work
    - The dependencies of the tests from the software version were not taken into account

- "Let’s automate all tests" and other unrealistic goals
  - Automation engineers are often more expensive than manual testers and with the same amount of money a lot would have been gained with manual testing
  - Automation does not recover from errors

- It is worthwhile to start lightly
  - It is good idea to try different tools
  - A pilot project
  - Let’s start for example by automating the smoke tests
  - Let’s find out if any other project or organization has used any corresponding tool in a similar project
  - A proper testing strategy and the commitment of the people is needed
    - Automation does not reduce the need for test planning
How to select a tool?

Test automation in the different phases of the process

- Unit testing:
  - The developers should take care of the unit testing
  - The developers are often not at all interested to test their own code manually
    - Manual testing is seen to be unexciting way to waste a lot of time
  - Tools are needed to make the automated unit testing as easy as possible
    - Preferably these tools should be integrated to the development tools of the developers as seamless as possible
      - Text editors, compilers, IDEs (Integrated Development Environment)
• Several commercial and free tools are available
• One example are the xUnit frameworks
  – The idea behind them is to move the responsibility to design, code, run, and analyze tests to the developers
  – Automated regression tests are gained as a “bonus”

• Integration testing:
  – Some unit testing tools can also be used in integration testing
  – Automatic generation of drivers and/or stubs
  – Before running actual integration test, it is wise to run a smoke test to the whole to find out if it is fit to even enter to integration testing
    • Compare to pre-integration
    • With the smoke test a lot of time can be saved in the integration testing
The generations of test automation

- Especially in GUI test automation, different generations can be identified
  - Ease of use, maintainability and the ability to find errors are improved
- In the following test automation is divided into five generations
  - Capture and replay
  - Structured scripts
  - Data driven
  - Keywords, action words
  - Model-based

The short history of (system level) automated testing

Capture Replay, Spaghetti Scripts

Structured Test Scripts

Data-Driven Scripts

Keywords, Action words

See for example *Software Test Automation: Effective use of test execution tools*  
By Mark Fewster and Dorothy Graham, Addison Wesley, 1999.
...and the future?

- Manual testing
- Automated test execution
- Model-based testing