



T-106.3100 Ohjelmoinnin jatkokurssi T2  
1.3.2007

## Quality Practices in a Small Development Team

Juha Itkonen

SoberIT

[Juha.Itkonen@hut.fi](mailto:Juha.Itkonen@hut.fi)



## What is Software Testing?





## Why do we need testing?

- ❑ To reveal defects in the software
- ❑ There are some risks associated with the unknown defects in the software
- ❑ Finding and fixing defects after delivery to the customer is expensive
  - ❑ Money
  - ❑ Lost data
  - ❑ Reputation
  - ❑ Physical damage
  - ❑ Safety critical consequences
  - ❑ ...
- ❑ We want to manage these risks by testing the software



## Definition of Software Testing

- ❑ Testing is the **process of exercising a software component**
- ❑ with the intent of **revealing defects** and **evaluating quality**
- ❑ and **demonstrating the difference between actual and required status**



## Testing – is not a phase at the end

- ❑ Often, testing is seen as some separate, last phase of software development process
  - ❑ That can be outsourced to separate testing team
  - ❑ That only needs to be done just before the release – if there is any time
- ❑ In practice, testing can not be separated from rest of the software development
  - ❑ Testing is much more than the final acceptance testing phase
- ❑ Testing has to be involved from the beginning
- ❑ Testing just reveals quality problems
  - ❑ there must be time and resources to react to the revealed problems

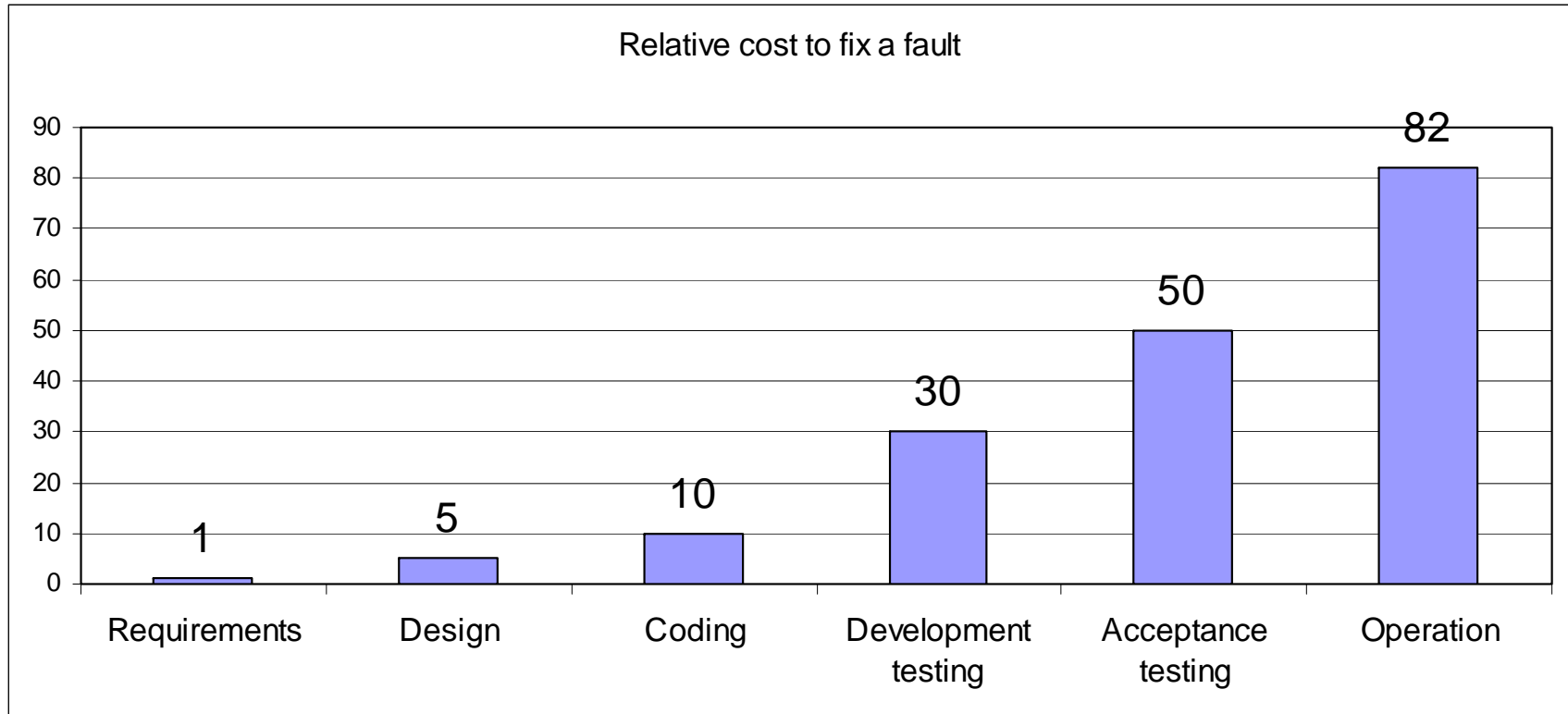


## Testing and Programmers ???

- ❑ Testing and testers don't create quality
  - ❑ Testing does not put the quality into the software
  - ❑ Programmers put the quality in
    - Good or bad quality
  - ❑ Testing reveals defects and measures the quality
- ❑ Testing is also the programmers' way to reveal defects and measure the achieved quality in their own code
- ❑ Catching defects in the late phases of development is expensive
  - ❑ Finding the defects in the coding phase is much cheaper than finding them in system testing or after delivery
  - ❑ Small and trivial coding defects can be very hard and tricky to debug based on system level symptoms



# Cost of fixing defects



*(Boehm. Software Engineering Economics. 1981)*



## Testing is destructive

- ❑ Finding defects is a destructive task
  - ❑ Requires destructive attitude and mindset
  - ❑ To find defects you must want to break the thing and find the nasty hidden problems there
- ❑ If you want to see the program work correctly, you most probably see it working correctly and miss the defects
- ❑ It is not easy for a programmer to achieve the destructive attitude towards his/her own code
- ❑ Independence is good for testing
  - ❑ Get somebody else to test your code



## Quality practices for programmers in small teams

- ❑ Coding conventions (or coding standards)
  - ❑ Consistent programming style
  - ❑ See e.g.
    - <http://java.sun.com/docs/codeconv/>
    - <http://www.gnu.org/prep/standards/>
- ❑ **Peer reviews**
  - ❑ Different document reviews
  - ❑ Pair programming
- ❑ **Unit testing**
  - ❑ Test-Driven Development
- ❑ **Defect reporting**
- ❑ Static code analysers
  - ❑ E.g. Lint
    - [http://en.wikipedia.org/wiki/Lint\\_programming\\_tool](http://en.wikipedia.org/wiki/Lint_programming_tool)



# Coding Convention

- ❑ `if(z<12&&i<60&&x<60){return true;}else{return false;}`
  
- ❑ `if (hours < 12 && minutes < 60 && seconds < 60)  
{  
 return true;  
}  
else  
{  
 return false;  
}`



## Introduction to peer reviews



# Reviews

- ❑ *A meeting or process at which an artifact is presented to peers, the user, customer, or other interested parties for comments and approval*
- ❑ Goals of a review include
  - ❑ Identifying defects
  - ❑ Educating and knowledge transfer
  - ❑ Presenting and discussing alternative solutions
- ❑ Reviews are effective way to do quality assurance
  - ❑ Reviews in general catch more than 50% of products defects



## Benefits of reviews

- ❑ Can be done as soon as the artifact is written
  - ❑ Easy to apply to parts of the system or incomplete components
- ❑ Can consider many different quality attributes as maintainability, reusability, security, etc.
- ❑ Distribution of knowledge
- ❑ Each defect can be considered in isolation
- ❑ Defect location time: Zero
- ❑ Increased awareness of quality issues
- ❑ Tool to improve entire development process
  - ❑ Data gathered as part of inspection process can be utilized to analyze and improve the entire software engineering process



# Team Review

- ❑ Team reviews are less rigorous than formal inspections
- ❑ Phases
  - ❑ Planning
    - Selecting participants, scheduling meeting, distributing the material
  - ❑ Individual preparation
    - Reviewing the artifact, logging all potential defects or issues
  - ❑ Meeting
    - Going through the artifact together
    - Logging all individual findings
    - Identifying new issues
  - ❑ Rework
    - Deciding how to act on each issue
    - Implementing the fixes
    - (checking the fixes, re-review)
- ❑ Goals
  - ❑ Reveal defects
  - ❑ Transfer knowledge
  - ❑ Discuss solutions



# Walkthrough

- ❑ Author presents a work product to peers
- ❑ No individual preparation
- ❑ Presentation of work product
  - ❑ Reading through
  - ❑ Program execution and debugging
  - ❑ Illustrations and diagrams of how program works under certain scenarios
- ❑ Goals
  - ❑ Maybe better used for other purposes than defect detection
  - ❑ Discuss ideas and brainstorm alternative approaches
  - ❑ Educate a larger group of people
- ❑ Understand the goals and results of the walkthrough
  - ❑ Which one was evaluated – the code or the presentation



## Pair review & Pass-around

- ❑ Also known as buddy-check
- ❑ Single individual checks a work product and returns with a list of defects
- ❑ Differences to prior mentioned techniques
  - ❑ Interaction is missing
    - less education, less discussion
- ❑ Multiple, concurrent buddy-checks -> Pass-around
- ❑ Pass-around often used e.g. for reviewing books and scientific articles



## Pair-programming

- ❑ Two programmers working tightly together
  - ❑ programming on a single computer
- ❑ Roles: Navigator & Driver
  - ❑ Roles and pair changed regularly
- ❑ Continuous education, peer-review, and discussion of better alternatives



## Proposed benefits of pair-programming

- ❑ Quality (defects, understandability, better design)
- ❑ Knowledge transfer within the development team
- ❑ Learning
- ❑ Enjoyment of work
  - ❑ but some developers resist using it
- ❑ Shortens elapsed time for development tasks
  - ❑ but spending two person's time may increase task effort
- ❑ Working harder
  - ❑ concentration
- ❑ Thinking aloud
- ❑ Negotiation about alternative solutions
- ❑ Mentoring
  - ❑ Learning form each other
- ❑ Courage
  - ❑ To make necessary changes and improvements to the code (refactoring)
- ❑ Continuous code review



## Introduction to unit testing



# Unit testing

- ❑ "A unit is the smallest possible testable software component"
  - ❑ Various interpretations exists
    - Procedure / function
    - Class / object / method
    - Small-sized component
- ❑ Goal is to ensure that software units are functioning according to specification (or as intended)
  - ❑ E.g. a function returns correct value when exercised with valid parameters
  - ❑ E.g. function behaves correctly when given an invalid input



## Unit testing level is a vital testing level

- ❑ Unit testing lays the foundation of good quality software
- ❑ Units have small size and simple function
  - ❑ Easy to design, execute, record, and analyze test results
- ❑ Unit level defects make integration and system testing too hard
  - ❑ Trivial bugs that prevent integration
  - ❑ Simple bugs on unit level can be hard to resolve on system level
- ❑ The cost of a defect is lowest when it is found in unit testing
  - ❑ Bug is found during integration, system or acceptance testing are harder to locate and repair
    - Debugging
    - Writing, prioritizing, and managing test and defect reports
  - ❑ Bug found during the use in production
    - Loss of money, loss of...



## How do you know that your code works correctly?

- ❑ Do you just trust your luck?
- ❑ Do you use "printf" outputs to see what happens in your code?
- ❑ Do you use debugger to find out how your code really works?
- ❑ Do you create small (temporary) widget to access the piece of code directly runtime?
- ❑ Do you use some ugly, cumbersome, time consuming practice that leaves trash into your code and is thrown up to the user's face when you least expect it...

```
"Getting here is not possible, something is really  
messed up in this code!!!"
```

```
"Fatal error 9981: Assertion failed : av_d_dx > 0"
```



# Automated unit testing

- ❑ According to modern unit testing ideas
  - ❑ All unit tests are automated
  - ❑ Programmers write unit tests
  - ❑ Automated tests are run as often as possible
- ❑ Cost-benefit break-even point reached after about 10 test runs...
- ❑ ...and you will probably do more than 10 runs a day!
  - ❑ Gather all your tests to a test suite and use it for regression testing
  - ❑ Integration, nightly, milestone
  - ❑ After every change to the code; new code or bugfix
- ❑ Write automated unit tests rather than use “printf” or debugger
  - ❑ Automated unit tests retain value over time
    - Tomorrow, after five years, ...
  - ❑ No need for human interaction
    - Write once – run with minimal effort



## Benefits of automated unit testing

- ❑ Programmer has better confidence on system
  - ❑ System works as developer has thought it should work
- ❑ Safe ground for adding new code
  - ❑ Confidence on the existing code
  - ❑ Regression testing suite
- ❑ Safe ground for refactoring your code
  - ❑ Refactoring = changing program code without adding new functionality
    - Needed to maintain quality of the system
  - ❑ "If it works, don't fix it" is history
- ❑ Unit tests document the system
  - ❑ Document the interface of unit under test
  - ❑ Unit tests are example code
  - ❑ Programmers understand code better than written documents!



## Automated unit testing in practice

- ❑ You might write more test code than production code
- ❑ Incremental build-up of test case base
  - ❑ Don't throw tests away
  - ❑ Maintain and add new tests
- ❑ All unit tests must pass 100 % all the time
  - ❑ When a test fails, fix it immediately
  - ❑ When programmers release any code at all, every unit test in the entire system must be running at 100 %
  - ❑ Not just that the new feature works, but that the changes haven't broken anything
  - ❑ Automated integration testing executed as often as possible



## ...but this is what you often hear from a developer

- ❑ "I have no time for testing" – the vicious circle
  - ❑ the greater the time pressure, the fewer tests
  - ❑ fewer tests -> unstable code
  - ❑ unstable code -> more debugging and fixing time needed
  - ❑ -> even greater time pressure
  
- ❑ Automated unit tests stabilize your code
  - ❑ Some time used for unit tests during coding
  - ❑ Less time needed for debugging, bug fixing etc.
  - ❑ Less time pressure -> better code and less defects



## Unit testing frameworks

- ❑ Convenient frameworks for writing unit tests as part of the programming task
  - ❑ In the same environment with the same language
- ❑ Tools of xUnit family are the most famous ones
  - ❑ jUnit
  - ❑ CUnit
  - ❑ NUnit
  - ❑ ...
- ❑ The unit testing frameworks provide a convenient way of writing and executing unit tests
  - ❑ Implementations for test cases, test suite, asserts, etc.
  - ❑ GUI and command-line based execution tools
    - Integrated to IDEs (e.g. Eclipse)
  - ❑ Convenient report generators
    - Xml, html



## Basic building blocks of a unit testing framework

- ❑ A way of writing test cases for each class, method, function, ...
  - ❑ Test case is a function or a class – a piece of code
- ❑ A way of managing suites of test cases
  - ❑ A data structure that is used to group test cases into logical collections
- ❑ Assert methods/functions that enable easy logging and reporting test results
  - ❑ The framework provides standard functions that are used to check the test results
  - ❑ Automated result logging and reporting provided by the framework
- ❑ Tools for executing test cases and reporting the results
  
- ❑ Initializing and cleaning up
  - ❑ Tests need to run against a known set of data structures
  - ❑ Initialization (or set up) function to initialize the variables, test data, etc. before executing each test case
  - ❑ Clean up (or tear down) function to release the used resources and clean up after executing each test case
  - ❑ The framework ensures that init and clean-up functions are properly executed
  - ❑ A one test suite has typically the same init and clean up functions for all its test cases

```

#include <stdio.h>
#include <string.h>
#include "CUnit/Basic.h"

/* Pointer to the file used by the tests. */
static FILE* temp_file = NULL;

/* The suite initialization function.
 * Opens the temporary file used by the tests.
 * Returns zero on success, non-zero otherwise.
 */
int init_suite1(void)
{
    if (NULL == (temp_file = fopen("temp.txt", "w+"))) {
        return -1;
    }
    else {
        return 0;
    }
}

```

**Example** of how things look like in CUnit  
<http://cunit.sourceforge.net/example.html>

```

    if (0 != fclose(temp_file)) {
        return -1;
    }
    else {
        temp_file = NULL;
        return 0;
    }
}

/* Simple test of fprintf().
 * Writes test data to the temporary file and checks
 * whether the expected number of bytes were written.
 */
void testFPRINTF(void)
{
    int i1 = 10;

    if (NULL != temp_file) {
        CU_ASSERT(0 == fprintf(temp_file, ""));
        CU_ASSERT(2 == fprintf(temp_file, "Q\n"));
        CU_ASSERT(7 == fprintf(temp_file, "i1 = %d", i1));
    }
}

/* Simple test of fread().
 * Reads the data previously written by testFPRINTF()
 * and checks whether the expected characters are present.
 * Must be run after testFPRINTF().
 */
void testFREAD(void)
{
    unsigned char buffer[20];

    if (NULL != temp_file) {
        rewind(temp_file);
        CU_ASSERT(9 == fread(buffer, sizeof(unsigned char), 20, temp_file));
    }
}

```



## Test-Driven Development

*(Test-first development)*



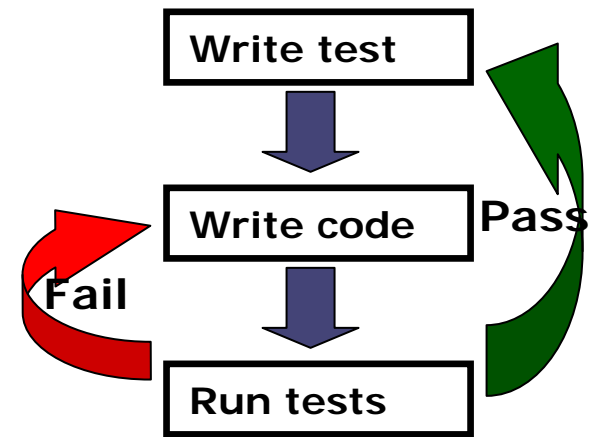
# Test-Driven Development

- ❑ Writing test before the code to be tested
  - ❑ "a little test, a little code, a little test, a little code, ..."
  - ❑ Tests are added gradually during implementation – not in large lump afterwards
- ❑ Process of writing tests drives low-level design and programming
  - ❑ Tests specify what code should do
  - ❑ Tests validate that code does what it should
- ❑ Actually, a design and coding practice
- ❑ One of the core practices of Extreme Programming
  - ❑ Developers have been applying TDD for several decades



## TDD episode (1/2)

- ❑ Proceeds step by step
  1. Write a test
  2. Design and implement just enough to make the test pass
  3. Run the tests
  4. Repeat
- ❑ Testing and coding alternate in very small steps
  - ❑ Duration of one cycle should be a few minutes
  - ❑ Small steps – difficult to make mistake





## TDD episode (2/2)

- ❑ Episode is over when you can't write a failing test anymore
  - ❑ Write test for each requirement of the code
  - ❑ Write test for each point that can possibly break
- ❑ One cycle at a time
  - ❑ Don't write a bunch of tests at once
- ❑ Refactor if you ever see the chance to make the design simpler
- ❑ Run all tests after finishing episode
  - ❑ Make sure you did not brake anything else



# Test-Driven Development – claimed benefits (1/2)

- ❑ Close feedback loop
  - ❑ TDD cycle is very short – know if code is working right after you programmed it
  - ❑ Easy and fast to fix
- ❑ Task-orientation
  - ❑ Encourage programmer to decompose problem into manageable programming tasks
  - ❑ Helps to maintain focus
  - ❑ Easier to see when you are done with a feature
- ❑ Low-level design
  - ❑ Programmer is forced to think which classes and methods to create, how they are used, how to name them, what arguments does a method take, what does a method return



## Test-Driven Development – claimed benefits (2/2)

- ❑ Results better code
  - ❑ If the test is too hard to write, the code being tested is too complicated
- ❑ Results testable code
  - ❑ Because the tests are there before the code
- ❑ Effect on quality
  - ❑ Testing becomes part of the development process and gets done
  - ❑ Side effect of TDD is that code gets thoroughly unit tested



## Try it!

- ❑ The only way to know!
- ❑ Experiences from our lab
  - ❑ Good feeling about the code written
    - General confidence that your code does what you have intended it to do
    - Good feeling when checking your code into version control with all green
  - ❑ Tests really get written when they are written beforehand
    - You always have an up-to-date regression testing suite
  - ❑ TDD helps you to keep focus on the current task
    - Program only what is needed to see the green light
  - ❑ Promote good practices
    - `printf` is used for displaying messages for user – not for developer
    - Debugger is used for debugging



## Unit testing best practices (1/2)

- ❑ Write tests in small increments
- ❑ Keep all tests running and passing all the time
- ❑ Run all your unit tests as often as possible, ideally every time the code is changed
- ❑ If it can't break on its own, it's too simple to break → otherwise, write unit tests
  - ❑ E.g. testing `setX()` and `getX()` usually ignored and tested as part of the tests for other functions
- ❑ Treat test code like production code
  - ❑ Remember readability and clarity
  - ❑ Remember coding conventions



## Unit testing best practices (2/2)

- ❑ Test code should be short and simple
  - ❑ Introducing more logic will introduce more errors
  - ❑ Kent Beck: Long tests indicate the likelihood of a design problem
- ❑ Document your unit tests
  - ❑ Use naming conventions
  - ❑ Test name should describe the intent of the test
    - E.g. `test_AddUser_WithAlreadyExistingUser`
  - ❑ Write comments for test methods
  - ❑ Include clear descriptions and messages
- ❑ Create independent unit tests
  - ❑ Don't assume a specific execution order
  - ❑ Don't assume specific execution time
  - ❑ ... or document your assumptions clearly



## Defect Reporting



## Why defects have to be reported and managed?

- ❑ To make sure found defects are not forgotten
- ❑ To get found defects fixed
- ❑ To enable defect management
  - ❑ Prioritization
  - ❑ Scheduling for fixing
  - ❑ Trade-off decisions
  - ❑ Tracking trends
    - Evaluating quality
    - Estimating effort and schedules
  - ❑ Estimating risks
    - Defect counts and severities



## Defect report

- ❑ A technical document written to describe the symptoms of a defect to
  - ❑ Communicate the impact and circumstances of a quality problem
  - ❑ Prioritize the defect for repair
  - ❑ Help a programmer to locate and fix the underlying fault
- ❑ Defect reports are the main communication channel from testing to development
- ❑ Defect reports are challenging to write
  - ❑ Bearing bad news
  - ❑ Explaining complicated behaviour
  - ❑ Communicating to people with different mindset using as few words as possible
  - ❑ Goal is to make people fix their mess instead of creating some new fancy functionality



# Defect report

1. **Defect-report identifier**
2. **Title:** A short description of the defect
3. **Date, time and finder**
4. **Test item and environment** including version and build numbers
5. **Description:** A detailed description of the defect
  1. Expected results
  2. Actual results
  3. Repeatability (whether repeated; whether occurring always, occasionally or just once)
  4. Additional information that may help to isolate and correct the cause of the incident
6. **Severity** of the defect



## Reporting the found defects

- ❑ Report the defects immediately
  - ❑ Don't leave it until the end of the day
- ❑ Make sure the defect has not been previously reported
- ❑ Find out how to reproduce the defect
  - ❑ Easier to isolate and get fixed
- ❑ Write specific and clear defect reports
  - ❑ Spend some time to find out what is the actual defect and under which conditions it occurs
  - ❑ What were the specific expected values and the actual values
- ❑ Be non-judgmental in reporting bugs
  - ❑ Bug reports need to be non-judgmental and non-personal
  - ❑ Reports should be written against the product, not the person, and state only the facts.



## What you can do for quality on this course

- ❑ *Overall test objectives* (why)
- ❑ What will and won't be tested (what)
- ❑ Test approach (how)
  - ❑ Testing in different phases
  - ❑ Test strategy, techniques, tools, ...
  - ❑ Defect reporting
  - ❑ Metrics and reporting
- ❑ Resource requirements (who)
  - ❑ Tester assignments and responsibilities
  - ❑ Test environments
- ❑ Test tasks and schedule (when)



# Manual functional testing

- ❑ System level testing
  - ❑ On GUI level
- ❑ Plan your test cases
  - ❑ Focus on the question “WHAT is tested?”
    - Not on the exact steps how to test it
  - ❑ List relevant test ‘ideas’ for each feature
  - ❑ Describe the expected results
    - If not obvious
- ❑ In this exercise your test cases are for guiding your own testing
  - ❑ and showing the assistants what you have done
- ❑ Report
  - ❑ What tests were executed
  - ❑ What were the results
    - Passed, failed, found defects
    - **Remaining known defects**



## More information...

- ❑ Programming style
  - ❑ [http://en.wikipedia.org/wiki/Programming\\_style](http://en.wikipedia.org/wiki/Programming_style)
- ❑ CUnit framework
  - ❑ <http://cunit.sourceforge.net/>
- ❑ Links to other xUnit frameworks
  - ❑ <http://en.wikipedia.org/wiki/XUnit>
- ❑ Introduction to TDD and pointers to further information
  - ❑ Janzen, D. and Saiedian, H. 2005. "Test-Driven Development: Concepts, Taxonomy, and Future Direction". In *Computer*, vol. 38(9), pp. 43 – 50.
  - ❑ [http://en.wikipedia.org/wiki/Test\\_driven\\_development](http://en.wikipedia.org/wiki/Test_driven_development)
- ❑ Links to static code analysis tools
  - ❑ [http://en.wikipedia.org/wiki/Lint\\_programming\\_tool](http://en.wikipedia.org/wiki/Lint_programming_tool)
  - ❑ [http://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)