Aalto University
School of Science

Testing

CS-C2105, Programming studio A
CS-C2120, Programming studio 2

10.2.2021

News

- Round 18 open
- Project topic selection due Thursday, 23:59
- Model project presentation on Friday 12th
- Project plans due:
  – General plan, Wednesday, Feb 17th
  – Technical plan, Friday, Feb 19th
- UML-task results should be published very soon.

Contents

- Software failures
- How to design tests
- Practical hints
- Other aspects of testing
- Software development process

Software failures

- Software can fail in many different ways
  - There is a logical error in the code and *program crashes*
    - e.g. null-pointer exception or divide by zero

      => exception handling can help detecting the error but not removing it.
  - There is a logical error and the program calculates *incorrect results*
    - You have seen a lot of these cases…

      => test results can help you to identify the reason for the error

Software can fail…

- The program handles well normal cases but fails to process *incorrect input data* or other *special cases*, like missing input files.
  - There is no way to avoid these situations, so you need to take care of them yourself
    - ⇒ exception handling can help here
- The program does not implement the *required features*.
  - E.g., some essential commands are missing or do not work.
    - => You just have to implement the missing parts

Software failures…

- The program works correctly, but is *far too slow* when working with realistic data…

  => Might be solved by changing to use more efficient data structures / algorithms.

- Other issues
  - The program may use *too much memory space*
  - The program may have serious *security problems*

Software failures/features

- Other issues
  - *Platform dependencies* may cause issues
    - Software may not be *portable*
  - Sometimes the program works correctly but in a surprising way
    - undocumented or *unexpected feature*, e.g., Excel in some cases interprets data as date values.

      => You just have to implement the fixes

Goal of testing

- Why should we test our programs?
  - ”*Program testing can be used to show the presence of bugs, but never to show their absence*!”
    - Edsger Dijkstra (1930-2002)
  - What else could we do to show that our software works?
    - Formal proofs of correctness have a very limited application area.

Some terms

- Bug
- Defect
- Error
- Failure
- Feature

Contents

- Software failures
- How to design tests
- Practical hints
- Other aspects of testing
- Software development process

How to design tests?

- *Equivalence partitioning*
  - Consider the space of possible input values
  - Split the space into areas and take test cases from each area.
  - For example:
    - coordinates from all quadrants
    - The Chess problem: input files having different ordering and selection of blocks
  - Makes more sense in unit testing of a one method instead of the whole program level

How to design tests?

- *Boundary value analysis*
  - Consider boundary cases of input or parameter values or data structures. Take test cases around them.
  - For example
    - Suppose some min / max values are specified for a parameter. What happens with values min, min-1, max, max+1.
    - *Off-by-one bugs*:
      - Check that array index remains within bounds
    - What happens with an empty collection (say List), or collection with just one item?
    - Consider searching/inserting/deleting items in a List. What happens, if the item is the first or the last one, or does not exist in the structure?

How to design tests?

- *Fuzz testing*
  - Consider what happens with wrong input values:
    - Illegal values
    - Wrong type of data (e.g., reading ”A” for Int)
    - Missing / empty data
    - Wrong format in data
    - Too large data sets
    - Missing input files / cannot access file

How to design tests?

- *Use case testing*
  - Consider typical user actions
  - How does she/he give commands?
  - What information is available for her/him?
  - What happens in each phase?
  - Can the user perform all subtasks?

Design your testing process

- Do NOT build your whole program before you start testing.
- Plan initially which parts of your program will you implement in each phase.
- How could you test each part (package / class / method) separately?
  – What do you need to be able to do it?
  – Where to use unit testing?

User interface testing

- You can build a visually complete user interface, including windows, panes, buttons and menus even though all logic behind them is still missing. For example:
  - Buttons and menus call Dummy methods.
  - Or they call Stub methods which return constant values just to show that the method is called appropriately.

File management testing

- Create a test class which can, e.g.,
  – open file
  – read file contents and display them
  – manage with end-of-file case
  – write contents of a given data set (generated for the test purpose only) to a file
  – close file
  – manage with errorneous content or format

Data structure testing

- Create a test class which calls methods of the tested data structure class or collection,
  – e.g., using unit testing
- Give generated data for the methods to build content in the structure, e.g. insert generated strings, ints, pairs, … into the structure to *initialize it for testing*.
- Build a method to *traverse the structure through* and print all values.
- Build the methods your program needs to manipulate the structure
  – Execute the methods with the test data structure and call the auxialiary method to print the content and thus allow you to monitor that the content is correct.
  – Test the special cases like empty structure, structure with one item, possible full structure

Contents

- Software failures
- How to design tests
- Practical hints (CONTINUE 15.13)
- Other aspects of testing
- Software development process

Asserts

- You can build your own asserts methods also without Scalatest library.
- Basically assert is a method, which receives as a parameter a logical expression (`exp==something`) to check that it holds.
  - `exp` is a variable in the tested method.
  - `something` is its expected value.
  - If the expression is not true, assert prints out a message for this (or throws an expection) and possibly quits the program.
  - The condition could also be some other comparison, like
    - assert(number > 0)
    - assert(x > 0 && x < 100)

```
class TestSupport {

  def assert(expression: Boolean, methodName : String) = {
    if (!expression) {
      println("Assert failed in method: ", methodName)
      System.exit(0)  // or something else
    }
  }
}
```

Debugging and user interfaces

- Debugger is a highly useful aid in many cases.
- However, debugging graphical user interfaces can be painful.
- Why?
  - Graphical user interface is based on processing *events* (mouse click, button click, key click, …) which are processed separately.
  - When you follow program execution, the program control jumps into event processing, which may be confusing.

Debugging and user interfaces…

- Jumping between uninteresting GUI methods and the actual logical code in unexpected ways is disturbing, if you try to follow progress step-by-step.
  - Setting breakpoints only in logical code is a partial solution.
  - But keeping track on which active method call you are investigating may be cumbersome.

Debugging and user interfaces…

- One option is to separate the GUI code as well as possible from the logical code, and test it separately
  - Use stubs or mocks to help you to provide minimal data for testing and the user interface can deliver and show data appropriately.
- And, implement a logical part of the program using command line interaction first (or stubs / mocks) to provide necessary UI data.
  - Test that the logic works properly before you integrate the parts, followed by *integration testing*

Printing values

- While debugger is a great tool to help you, printing variable values is a useful method, too, to follow program execution and checking that variable values are correct.
- Assert methods fit well together with this.

Hint: Toggle debugging mode

- Define a variable to toggle whether you are in debug mode or mode

```
val DEBUG_ON = true
```

```
class TestSupport {

  def assert(expression: Boolean, methodName : String) = {
    if (!expression) {
      println("Assert failed in method: ", methodName)
      System.exit(0)  // or something else
    }
  }
}
//-------------------
      …
      If (TestSupport.DEBUG_ON) println (…)
      …
      If (TestSupport.DEBUG_ON)
            TestSupport.assert(x > 0, ”calculation”)
```

Contents

- Software failures
- How to design tests
- Practical hints
- Various aspects of testing
- Software development process

What can we test?

- Program functionality
  - Software meets the given requirements
- Program correctness
  - Software gives correct responses to ***all kinds*** of inputs
- Performance testing
  - Performs its functionality in acceptable time
- Usability testing
  - User interaction with the software is acceptable

What can we test...

- Software works on the desired platforms
  - Different operating systems
  - Different devices
- Acceptance testing
  - Software meets the general requirements of the customer

Some more terminology

- Alpha testing
  - Testing the feasibility of the initial software (or prototype) among potential customers
- Beta testing
  - User acceptance testing for a limited audience
- Functional vs. Non-functional testing
  - Functional: what the program should do?
  - Non-functional: other aspects like performance, usability, scalability, …
- Installation testing
  - Whether the installation process works correctly

Some more terminology…

- Regression testing
  - Running a series of tests to discover if anything is broken after a major change in software
  - Typically ready-made regression test sets
- Smoke testing
  - Testing whether it is worthwhile to proceed with further testing
- Stress testing
  - Testing the limit capacity of operation, to discover when the performance breaks down.
- Internationalization and localization
  - Testing that the software works in different languages and geographical / cultural areas.

Different testing processes

- Static testing
  - Code reviews, walkthroughs in collaboration with a peer.
  - Identifying dead code
- Dynamic testing
  - Executing program with test cases

Different testing approaches

- White-box testing/glass box testing
  - Seeks to show that internal structures / algorithms within program / program unit work correctly.
  - Usually carried out in unit testing level
- Black-box testing
  - Seeks to show that the program / program unit produces correct output without considering how it does it (even with not access to it)
- Gray-box testing
  - Have access to source code but perform tests as in black-box testing.

Test quality

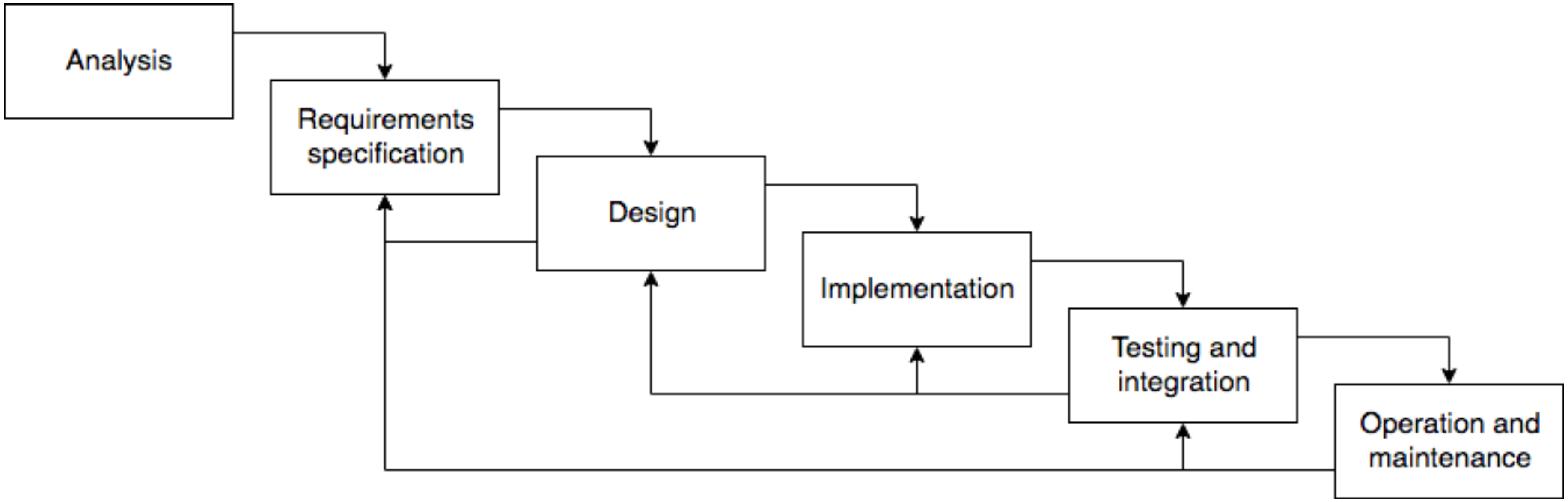
- How widely the test cases cover the code.
  - Function coverage
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Path coverage
- Fault injection
- Mutation testing

Contents

- Software failures
- How to design tests
- Practical hints
- Other aspects of testing
- Software development process

Software development processes

- Waterfall model



Software development processes

- Agile software development
  - Development is *iterative, incremental, evolutionary*
  - Works in short cycles covering planning, analysis, design, coding, unit testing, and acceptance testing.
  - Works in close collaboration with customers
  - *Scrum* is one agile framework having 2 week sprints (and there are many others)

Software development processes

- TDD (test driven development)
  - Turns requirements into tests
    1. Add a new test
    2. Run all tests and see if the new test fails
    3. Write code that addressed the new test
    4. Run tests and revise code until all tests pass
    5. Refactor code
    6. Goto 1

Some future courses

- CS-C3150 Software Engineering
- CS-C3180 Software Design and Modelling
- CS-C2130 Software Project 1
- CS-C2140 Software Project 2

Next

- Choose project topic
- Submit project plan (in two phases)
- Follow MyCourses / A+ announcements for project plan demos etc.