Aalto University
School of Science

Testing

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

17.1.2024

Learning goals for the lecture

- Get an understanding of different kinds of software failures
- Understand how they could be detected with different testing approaches
- Understand different ways of planning test cases
- Learn some key terminology and some practical hints.

Learning goals for the course

- In round 15 exercises, you will learn to apply basic unit testing in Scala.
- You will learn basics test driven development
- In the project, you will plan your testing and carry out testing in practice.

Contents

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

Software failures

- Software failure denotes here some problems which are related to executing the program.
- Syntax errors, which prevent program compilation, are not considered failures here. Program is not executed.

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 remove 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
- Program has not enough execution resources
  - Running out of memory
  - Failing to access a class or the class file has damages.

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 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

Finally, software can fail…

- 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

Goal of testing

- Testing aims at identifying problems in software.
- As failures are different, testing approaches must be different

Paradox

- ”*Program testing can be used to show the presence of bugs, but never to show their absence*!”
  - Edsger Dijkstra (1930-2002)
  - Even rigorous testing cannot prove that software is faultless.
  - But:
    - 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 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 a 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
  - This is something that your project needs to consider, when assistants test your code.

How to design tests?

- *Use case testing*
  - Consider typical user actions
  - How does the user give commands?
  - What information is available for the user?
  - What happens in each phase?
  - Can the user perform all subtasks?
    - How complex it is (usability)

Design your testing process

- Planning testing is not just about planning test cases
- You need to design the process, when and how you test the program.

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?
  - When to use unit testing?
  - When to use testing through the user interface?

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.
  - Using REPL could help here

Data structure testing

- 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

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
  - cope and recover, when there is errorneous content or format in the file

Graphical user interface testing

- You can build a visually complete user interface, GUI, including windows, panes, buttons and menus even though all logic behind them is still missing. For example:
  - Buttons and menus call Dummy methods which do not do anything (???)
  - Or they call Stub methods which return constant values just to show that the method is called appropriately, and the GUI responds.

Contents

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

A+ resourses

- At the bottom of course A+ table of contents is a chapter
  - "Finding and fixing errors"
  - It has much useful information. Read it!
- Chapter 15.1 present *unit testing*, which is an important method for checking the correctness of small parts of a program.

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)

Example of own assert-function

```
class TestSupport {

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

Debuggers

- Debuggers allow you to execute the program in a controlled way. Most importantly:
  - Execute code step by step
  - Set *breakpoints* into the code, where execution stops
  - Explore variable values, when program is stopped for a while
  - Often more elaborated features available, such as exploring collections or data structures.
- See A+ page “Finding and fixing errors” for basics.

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”)
```

Note

- All these are related to software failures which concern bugs.
- Different strategies are needed to address other types of software failures.
- We discuss them later.

Break