

CS3213 Project – Week 7

More Testing | 02-03-2022

- ☐ Recap: Equivalence Class Analysis
- ☐ Testing - Best Practices
- ☐ Assignment 7 - "Unit Testing"
- ☐ Introduction to Debugging

Short Announcements

- ❑ LumiNUS > Survey > Presentations in Week 13

- ❑ So far: 4 responses.

- ❑ Please add your availability so that we can plan accordingly.

- ❑ **Checkstyle** in Projects

- ❑ Version 8.44, **google_checks.xml**

- https://checkstyle.sourceforge.io/google_style.html

- ❑ Use **IDE plugin** to use checkstyle

- <https://plugins.jetbrains.com/plugin/1065-checkstyle-idea>

- ❑ or use the jar to run checkstyle from the **command line**

- <https://github.com/checkstyle/checkstyle/releases/tag/checkstyle-8.44>

```
java -jar checkstyle-8.44-all.jar -c google_checks.xml <project folder>
```

Equivalence Class Analysis (2/2)

Recap

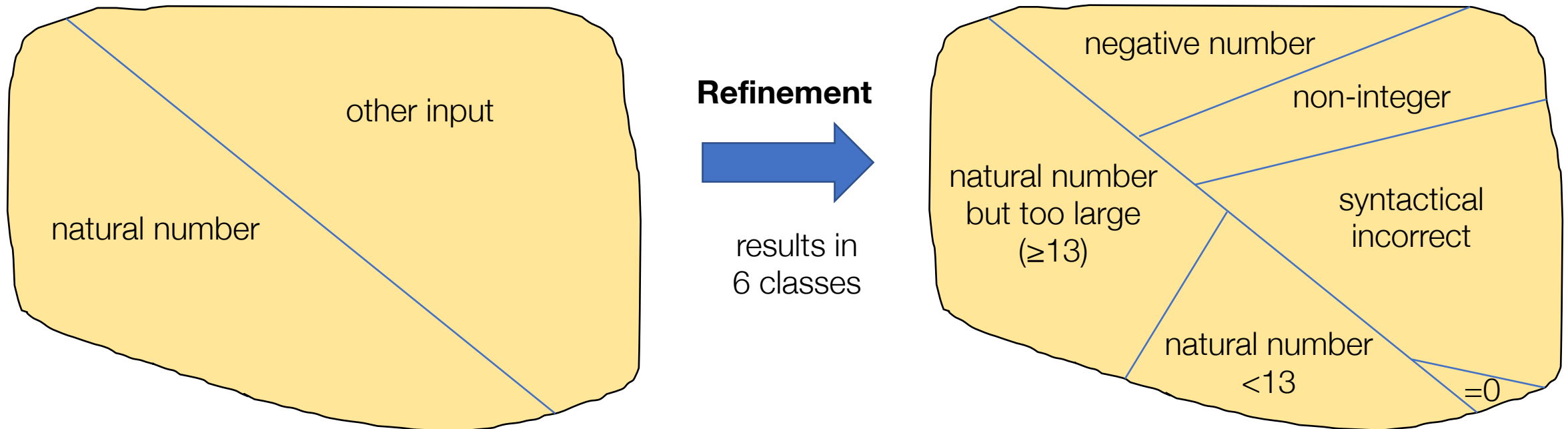
- ❑ Procedure (attributed to [Myers, 1979]): Equivalence Class Partitioning
 1. Identify: input variables, equivalence classes for valid and invalid inputs
 2. Divide classes further intuitively, if necessary
 3. Select input data for each class, determine expected outcomes
- ❑ The equivalence classes are to be numbered unambiguously. The generation of test cases from the equivalence classes requires two rules:
 - ❑ The test cases for valid equivalence classes are formed by selecting test data from as many valid equivalence classes as possible.
 - ❑ The test cases for invalid equivalence classes are formed by selecting a test data from an invalid equivalence class. It is combined with values taken exclusively from valid equivalence classes.
- ❑ Often used: Test of equivalence class boundaries (Boundary Value Analysis).

Equivalence Class Partitioning

(Example 2/4)

Recap

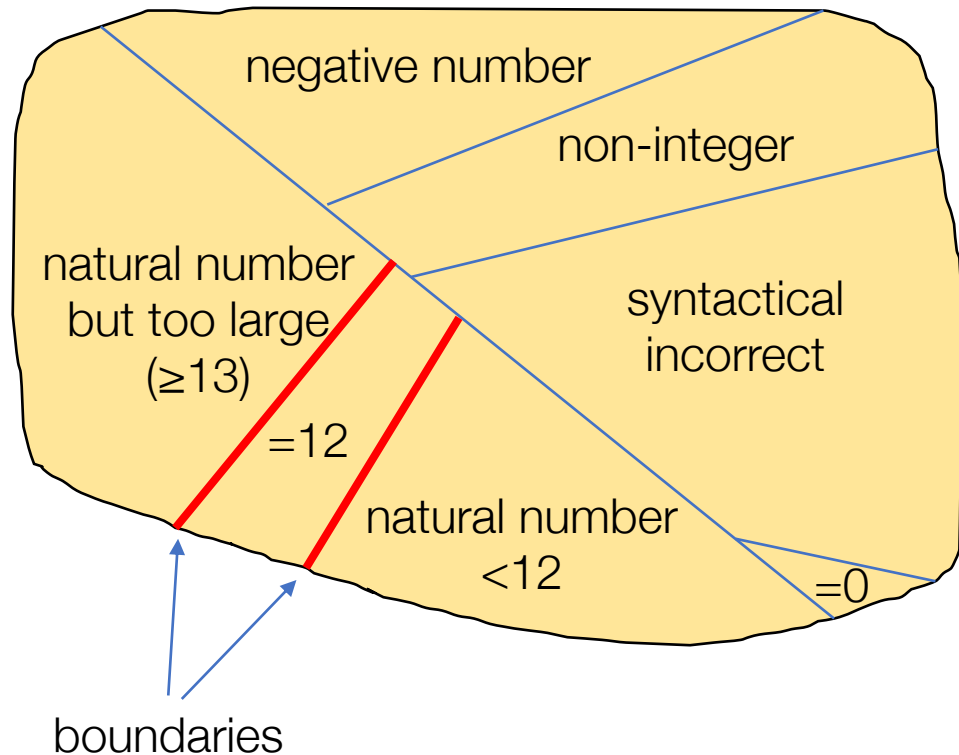
- ❑ A program to calculate the **factorial** of n .
- ❑ A program that is to calculate the factorial of n must reject (1) negative numbers, (2) real fractions, (3) numbers whose factorial is too large ($n \geq 13$), and (4) syntactically incorrect inputs. Special case: 0!



Boundary Value Analysis

(Example 5/5)

Recap



Class	Input	Expected Outcome
Negative number	-5	Error message
Non-integer	3.14	Error message
Too large number	100	Error message
Syntactical Incorrect input	"ABC"	Error message
Normal/expected input	7	5040
Zero	0	1
Boundary Value	12	479001600
Boundary Value -1	11	39916800
Boundary Value +1	13	Error message

Exercise: Equivalence Class Testing



A program for the warehouse management of a building materials store has an input option for the registration of deliveries. If wooden boards are delivered, the **type** of wood is entered. The program knows the wood types **oak**, **beech** and **pine**. Furthermore, the **length** in centimeters is specified, which is always between **100 and 500**. A value between **1 and 9999** can be entered as the delivered **number of items**. In addition, the delivery is given an **order number**. Each order number for wood deliveries starts with the letter **“H”**.

- (a) Derive **equivalence classes** using the above specification. Note:
- For each of the four function parameters there exists at least one valid and one invalid equivalence class.
 - You can assume that type conformity of the function parameters is guaranteed, i.e., invalid equivalence classes for non-type conform input values need not be considered.

Exercise: Equivalence Class Testing

(Part a: Equivalence Classes)

A program for the warehouse management of a building materials store has an input option for the registration of deliveries. If wooden boards are delivered, the **type** of wood is entered. The program knows the wood types **oak**, **beech** and **pine**. Furthermore, the **length** in centimeters is specified, which is always between **100 and 500**. A value between **1 and 9999** can be entered as the delivered **number of items**. In addition, the delivery is given an **order number**. Each order number for wood deliveries starts with the letter **“H”**.

Input Variable	Valid Classes	Invalid Classes
Type of Wood	1) Oak 2) Beech 3) Pine	4) Others, e.g., Steel
Length	5) $100 \leq \text{Length} \leq 500$	6) $\text{Length} < 100$ 7) $\text{Length} > 500$
#Items	8) $1 \leq \text{\#Items} \leq 9999$	9) $\text{\#Items} < 1$ 10) $\text{\#Items} > 9999$
Order Number	11) First Character is “H”	12) First Character is not “H”

Exercise: Equivalence Class Testing



- (b) Now derive a **minimal** set of test cases so that each equivalence class is tested by **at least one** representative. For this example, it is okay to ignore the expected outcome and only name the inputs for each test case. The expected outcome cannot be inferred from the specification above.

Exercise: Equivalence Class Testing

(Part b: Test Cases)

- ❑ The test cases for **valid** equivalence classes are formed by selecting test data from **as many valid equivalence classes as possible**.
- ❑ The test cases for **invalid** equivalence classes are formed by selecting a test data from an invalid equivalence class. It is combined with values taken **exclusively** from valid equivalence classes.

Input Variable	Valid Classes	Invalid Classes
Type of Wood	1) Oak 2) Beech 3) Pine	4) Others, e.g., Steel
Length	5) $100 \leq \text{Length} \leq 500$	6) Length < 100 7) Length > 500
#Items	8) $1 \leq \text{\#Items} \leq 9999$	9) #Items < 1 10) #Items > 9999
Order Number	11) First Character is "H"	12) First Character is not "H"

Test Case	1	2	3	4	5	6	7	8	9
<i>(additional) tested equivalence classes</i>	1), 5), 8), 11)	2)	3)	4)	6)	7)	9)	10)	12)
Type of Wood	Oak	Beech	Pine	Steel	Beech	Beech	Beech	Beech	Beech
Length	200	300	300	300	50	1000	200	200	200
#Items	100	200	100	100	100	100	0	50000	100
Order Number	"H1"	"H1"	"H1"	"H1"	"H1"	"H1"	"H1"	"H1"	"J2"

How to
further
improve?

What are the attributes of a “good” Test Case?



Attribute	Explanation
Powerful	Test case discovers existing failures
Valid	Test cases discovers actual failures
Relevant	Reveals test cases which are important for stakeholders
Reliable	
Executable	
Representative	
Non-redundant	There is no other test case which reveals the failure
Maintainable, repeatable, cheap, ...	

Testing – Best Practices ^(1/2)

- ❑ Test cases should be **independent**!
 - ❑ The JUnit execution model executes test cases in arbitrary order (unless explicitly defined).
 - ❑ Use @Before.. Annotations to define test case preparations! Do not assume that another test case already created some sort of test data or program state.
 - ❑ Dependent test cases can cause **flaky tests**: sometimes they pass, sometimes they fail, depending on the test execution order. General reasons for flaky tests:
 - ❑ an issue with the test itself
 - ❑ some external factor compromising the test results
 - ❑ an issue with the newly-written code

August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. “*IFixFlakies: a framework for automatically fixing order-dependent flaky tests*”. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019.

Wing Lam, Reed Oei, August Shi, Darko Marinov and Tao Xie, “*iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests*”. 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019.

Testing – Best Practices (2/2)

- ❑ One test case for one feature (→ Single Responsibility for Tests).
Keep things simple!
- ❑ 5LOC Rule: Strive to write test cases 5LOC long.
- ❑ Choose meaningful test method names!
- ❑ Use same package structure as for source code.
→ Test code is separate, but you can access methods with package accessibility
- ❑ Test cases should have the end user or defined requirements in mind.
- ❑ Peer review is important!

(un)Testable Code ^(1/3)

Based on “Guide: Writing Testable Code” written by Google developers
<http://misko.hevery.com/code-reviewers-guide/>

Flaw #1 – Constructor does Real Work

*“When your constructor has to instantiate and initialize its collaborators, the result tends to be an inflexible and prematurely **coupled design**. Such constructors shut off the ability to inject test collaborators when testing.”*

- ❑ violates the **S**ingle Responsibility Principle
- ❑ testing directly is **difficult**

Before: Hard to Test

```
// Basic new operators called directly in the  
// class' constructor.
```

```
public class House {  
    Kitchen kitchen = new Kitchen();  
    Bedroom bedroom;  
  
    public House() {  
        bedroom = new Bedroom();  
    }  
  
    // ...  
}
```

```
// An attempted test that becomes pretty hard
```

```
public class HouseTest {  
  
    @Test  
    public void testThisIsReallyHard() {  
        House house = new House();  
  
        // Darn! I'm stuck with those Kitchen and  
        // Bedroom objects created in the  
        // constructor.  
    }  
}
```

After: Testable and Flexible Design

```
public class House {  
    Kitchen kitchen;  
    Bedroom bedroom;  
  
    public House(Kitchen k, Bedroom b) {  
        kitchen = k;  
        bedroom = b;  
    }  
    // ...  
}
```

```
// New and Improved is trivially testable, with any test-double  
// object.
```

```
public class HouseTest {  
  
    @Test  
    public void testThisIsEasyAndFlexible() {  
        Kitchen dummyKitchen = new DummyKitchen();  
        Bedroom dummyBedroom = new DummyBedroom();  
  
        House house = new House(dummyKitchen, dummyBedroom);  
  
        // Awesome, I can use test doubles that are lighter weight.  
    }  
}
```

(based on "Guide: Writing Testable Code" – <http://misko.hevery.com/code-reviewers-guide/>)

(un)Testable Code ^(2/3)

Flaw #2 – Digging into Collaborators

*“If you have to test a method that takes a **context object**, when you exercise that method it’s hard to guess **what is pulled out of the context**, and what isn’t cared about.”*

Example violations:

```
getUserManager().getUser(123).getProfile().isAdmin()
```

→ all you need to know if the user is an admin

```
context.getCommonDataStore().find(1234)
```

→ no need to have the complete data store object...

Before: Hard to Test

```
public class SalesTaxCalculator {
    TaxTable taxTable;
    SalesTaxCalculator(TaxTable taxTable) {
        this.taxTable = taxTable;
    }

    float computeSalesTax(User user, Invoice invoice) {
        Address address = user.getAddress();
        float amount = invoice.getSubTotal();
        return amount * taxTable.getTaxRate(address);
    }
}
```

```
public class SalesTaxCalculatorTest {
    // ...

    SalesTaxCalculator calc =
        new SalesTaxCalculator(new TaxTable());
    Address address = new Address("1600 Amphitheatre");
    User user = new User(address);
    Invoice invoice = new Invoice(1, new ProductX(95.00));

    assertEquals(0.09,
        calc.computeSalesTax(user, invoice), 0.05);

    // ...
}
```

After: Testable and Flexible Design

```
public class SalesTaxCalculator {
    TaxTable taxTable;
    SalesTaxCalculator(TaxTable taxTable) {
        this.taxTable = taxTable;
    }

    float computeSalesTax(Address address, float amount) {
        return amount * taxTable.getTaxRate(address);
    }
}
```

```
public class SalesTaxCalculatorTest {
    // ...

    SalesTaxCalculator calc =
        new SalesTaxCalculator(new TaxTable());
    Address address = new Address("1600 Amphitheatre");

    assertEquals(0.09,
        calc.computeSalesTax(address, 95.00), 0.05);

    // ...
}
```

(based on "Guide: Writing Testable Code" – <http://misko.hevery.com/code-reviewers-guide/>)

(un)Testable Code ^(3/3)

Flaw #3 – Brittle Global State & Singletons

*“The problem with using a Singleton is that it introduces a certain amount of **coupling** into a system — coupling that is almost always unnecessary. ... [U]nless you change your design, you are forced to rely on the correct behavior of the Singleton in order to test any of its clients.”*

[J.B. Rainsberger, Junit Recipes, Recipe 14.4]

Flaw #4 – Class Does Too Much

→ violates the **S**ingle Responsibility Principle

- ☐ Hard to debug
- ☐ Hard to test
- ☐ Non-extensible system

Testable Code & Writing Unit Tests

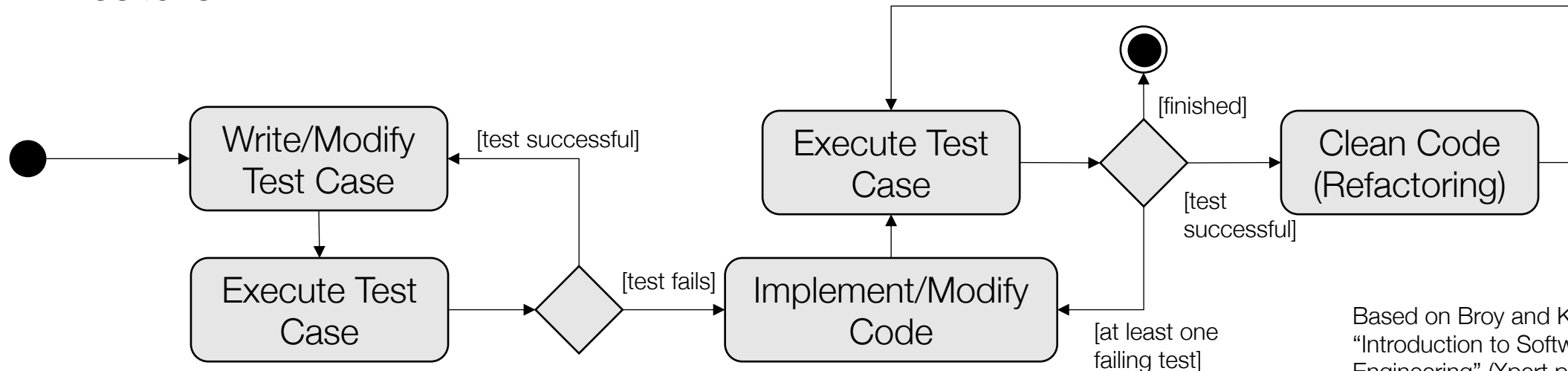
– Closing Remarks

- ❑ **Easy to write.** It should be easy to implement unit test without enormous effort.
- ❑ **Readable.** With a good unit test, you can fix a bug without actually debugging the code!
- ❑ **Reliable.** Unit tests should fail only if there's a bug in the system under test. Good unit tests should be reproducible and independent from external factors such as the environment or running order.
- ❑ **Fast.** Running regression test suites should be feasible.
- ❑ **Truly unit, not integration.** Eliminate the influence of external factors.

Test Driven Development (TDD)

Recap

- ❑ Refers to a **style** of software development that focuses on testing.
- ❑ The three core tasks of **coding**, **testing** and **design** are carried out in an interactive manner.
- ❑ The procedure described below maps the simple rules of Test-Driven Development in an incremental/iterative process for the implementation of one feature.



Based on Broy and Kuhrmann.
"Introduction to Software
Engineering" (Xpert.press), 2021.

Three Laws of TDD (by Kent Beck)

Rule 1:

You may not write production code until you have written a failing unit test.

Rule 2:

You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Rule 3:

You may not write more production code than is sufficient to pass the currently failing test.

K. Beck. "Test Driven Development: By Example." Addison-Wesley Longman, 2002.

TDD – Implications



Change of perspective

When creating test cases, developers become "users" of the code, which means that developers now have to deal with the interface of the respective module.



Testability



Documentation

The tests can also be seen as part of the documentation of the software. E.g., how to instantiate objects, how to use software and components.

It is important to note that it is **explicitly not** in the spirit of TDD to create all test cases before starting to write production code. “Clean Code” recommends writing tests and production code alternately and switching between these two activities in "micro-iterations" lasting only a few minutes.

R. C. Martin. “Clean Code: A Handbook of Agile Software Craftsmanship.” Prentice Hall, 2008. 46.

Try Test Driven Development (TDD) yourself!



We can discuss your experience in the lab
and at the end of the lecture!



Assignment 7: Unit Testing



Assignment 7: Unit Testing

CS3213 Foundations of Software Engineering (AY21/22 Sem2)

Submission Deadline: **Tue 22/03/2022, 10 pm**
Discussion: Week 10

- You must strictly comply with the noted deadline. No late submissions!
- This is a **group** assignment, i.e., you need to solve and submit this assignment in the assigned/-formed groups via LumiNUS. Acts of plagiarism are subjected to disciplinary action by the university. Please refer to <https://www.nus.edu.sg/celc/programmes/plagiarism.html> for details on plagiarism and its associated penalties.
- Please use appropriate tools to create your solutions (e.g., LibreOffice/Word/LaTeX for textual submissions, or draw.io for graphical solutions). Handwritten solutions are accepted only in exceptional cases and if they are very legible.
- Please create a **PDF document** from the solution including a **title sheet** with the exercise sheet number, group number and the names/matriculation numbers of the students in the group.
- Please use this scheme as the file name for the PDF document: **assignment_X.group.YY.pdf**, where X is the exercise number and YY is the group number.
- Please submit this PDF document via LumiNUS. In case of any discrepancies regarding the submission date, the date given in LumiNUS will count.
- There are **2 marks** to be scored for this assignment sheet. The worst score for any assignment sheet is 0 marks.

Overview

Testing is not just a stage at the end of software development before delivering the product to the customer. On contrary, it can be (and should be) a continuous action throughout the complete software development process. You already have spent some weeks implementing your project, and the latest now is the time to rethink your testing approach and make sure that your development efforts are going in the correct direction. Therefore, we ask you in this assignment to design test cases to validate your component. Note that you do not have to wait until week 10 to work on this assignment but better start early with designing proper test cases for your project.

General comment for research-related projects (i.e., 3.3, 4.3, 5.1, 5.2): While your testing approach might have different goals than those in the other projects, you still can provide test cases that support your evaluation plan. Therefore, the assignment applies to your project similarly, with a partial focus on what you have decided in your evaluation plan in Assignment 4b.

Due to: Tuesday,
22/03/2022 (**Week 10**)

Available Marks: **2**

Assignment 7 – Task: Unit Test Suite (1/2)

- ☐ Testing is not just a phase at the end of the project!
- ☐ Design custom **unit test cases** for your project and the corresponding system component.
- ☐ Create **new test classes** in the existing test folder. Your goal should be to cover all relevant requirements (incl. corner/special cases) for your component.
- ☐ You will need to **extend the test data/resources**.

Assignment 7 – Task: Unit Test Suite (2/2)

❑ Submit a PDF report (\leq **1 page**, excluding the title page) with the following structure:

- 1. Implementation Snapshot:** Specify here the link (i.e., GitHub Commit Hash) to your code submission that includes your test cases.
- 2. Test Results:** Show the current state of your implementation by executing all test cases in your project. Submit the overall test report (passing and failing test cases).
- 3. Test Coverage Reports:** Show the code coverage results achieved by your test.

Assignment 7 – Grading Comments

- ❑ **Report Structure:** your PDF report needs to follow the described structure
- ❑ **Test Case Structure/Quality** [1 mark]: Test cases should be **well structured** and clearly separate test input, program execution, and expected test output. They should follow the **best practices** covered in the lecture (Week 7).
- ❑ **Test Coverage** [1 mark]: Your test cases should cover *all* your implementation *classes*. You should aim at a *line coverage* of *70% or more*. Furthermore, you need to cover the main aspects relevant for your project:
 - ❑ *Parser/Interpreter*: see expected language features in the LumiNUS forum
 - ❑ *Alignment*: you need to explore the various cases of structural alignment and variable alignment: different structures and their combinations, cases with multiple possible variable mappings, etc.
 - ❑ *Error Localizer*: your validation is meant to investigate the differences between the implemented algorithms, hence, you should focus mostly on that part, but also cover correctness testing
 - ❑ *Feedback*: your validation depends on your feedback strategies; explore various combinations of input and make sure that all your feedback aspects (mentioned in Assignment 4b) are covered



**Any remaining question about
Assignment 7?**

Debugging – History

9.9.1945
15:45

92

9/9

0800 Antan started
1000 " stopped - antan ✓

1300 (032) MP - MC { 1.2700 9.037 847 025
2.130476415 } 9.037 846 995 correct
(033) PRO 2 2.130476415
2.130676415
correct

Relays 6-2 in 033 failed special speed test
in relay " 11.000 test "

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545

Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antan started.
1700 closed down.

Relay 2145
Relay 33

- ❑ A moth in the Mark II computer causes errors in Relay No. 70, Panel F.
- ❑ Mrs. Grace Murray Hopper removes the error and documents it in the log book: "First actual case of bug being found."
- ❑ "open"-visible error!
→ Elimination is easy!

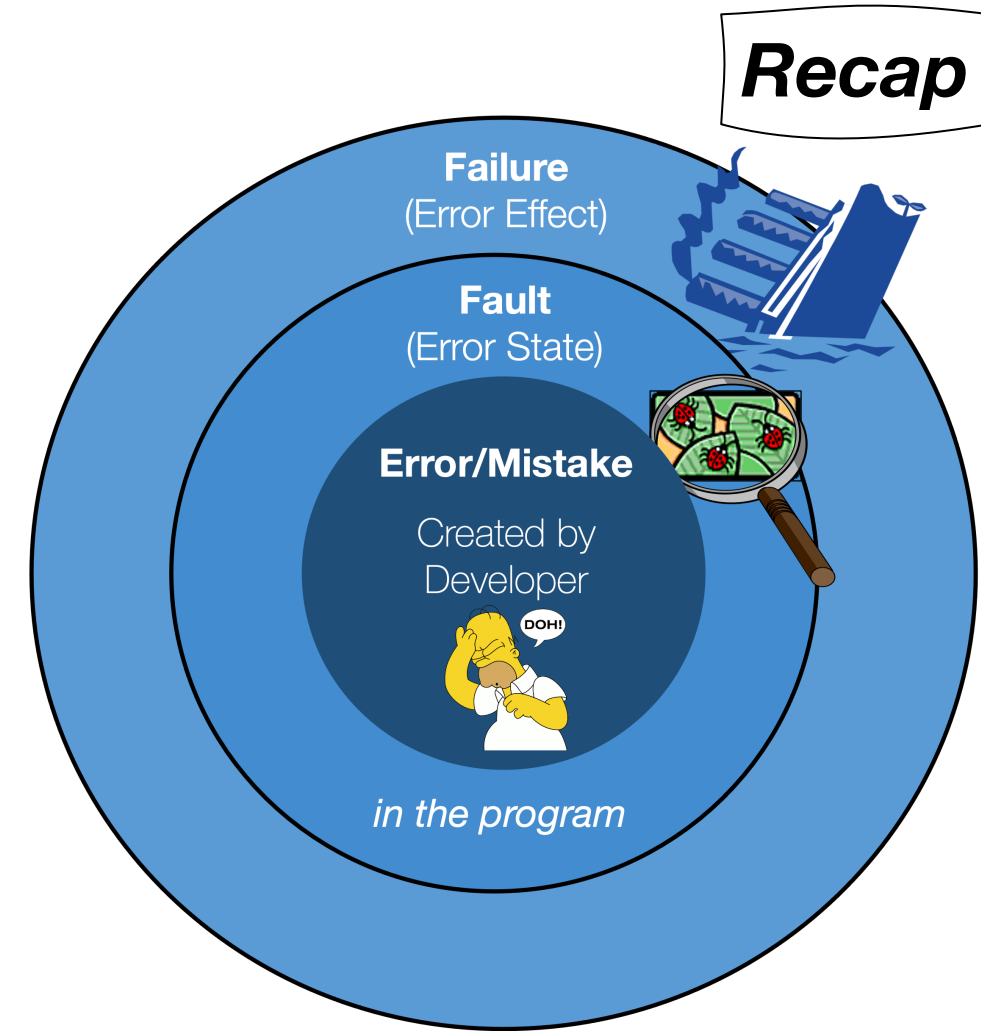
Debugging – From Error to Failures

The issue of *debugging* is to

- ❑ *relate* an observed failure to a fault/defect and
- ❑ to *remove* the defect such that the failure no longer occurs.

Debugging Steps:

- ❑ Execution of tests!
- ❑ ***Fault Localization!***
- ❑ Identify possible fixes.
- ❑ Choose the best fix.
- ❑ Implement the best fix!



Debugging – Scientific Method

1. **Observe** the misbehavior.
2. Create a **hypothesis** about the cause of the misbehavior.
3. Use the hypothesis for predictions.
4. Test the hypothesis with **experiments** or observations and **refine** the hypothesis based on the results.
5. Repeat steps 3 and 4 until the **cause** is found.

Debugging – Difficulties

- ❑ Symptom and failure cause can be far apart.
- ❑ Symptoms of one error may be hidden by other errors.
 - ❑ Fault masking: “An occurrence in which one defect prevents the detection of another [IEEE 610]
- ❑ Symptoms of one error may disappear or change due to correction of another error.

„Debugging is one of the more frustrating parts of programming. It has elements of brain teasers, coupled with the annoying recognition that you have made a mistake.“

B. Shneiderman: Software Psychology. Winthorp Publishers, 1980.

Debugging – Techniques

❑ **Brute Force**

Collect all data about the program execution. Try to find the error in it.

❑ **Cause Elimination**

The test case is reduced in size until only a small part of the program is executed. The error "must" be located in this part.

❑ **Backtracking**

The possible program execution paths are traced back from the occurrence of the symptoms to the program start. The error "must" be on one of the paths.

Tools can be used to collect the needed data (executed program parts, program paths, program state, ...): Coverage analysis, Interactive Debugger, Trace Generator

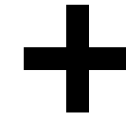
→ *see Lab sessions in Week 8!*

Conclusion

- ❑ Testing is **no late** phase in the development process!
- ❑ **Debugging** is time-consuming, but can be aided by **tools** and **techniques**. Next time we will explore this more deeply.

Next Week (Project-Part) – Week 8: **Debugging**

- More Debugging (TRAFFIC, Slicing, Delta Debugging)
- Assignment 8: Final Code Submission + Presentations



Midterms