CS3213 Project – Week 6

Unit Testing | 16-02-2022

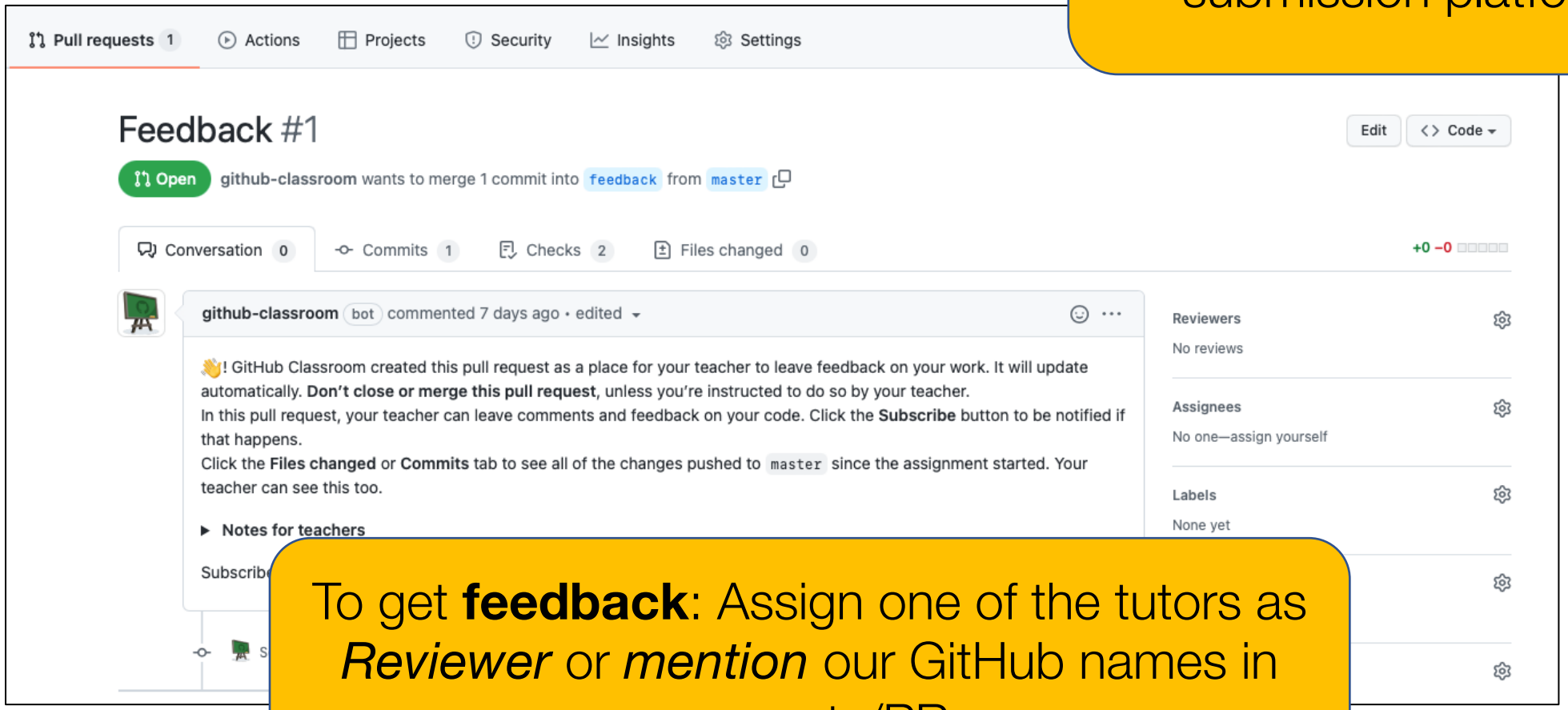
- ❏ Unit Testing (Basics)
- ❏ Assignment 6 - ”Intermediate Deliverable”

Important Upcoming Deadlines

- ❑ Intermediate Deliverable: Tuesday, 01/03/2022, 10 pm (Week 7)
- ❑ Final Code Submission: Tuesday, 12/04/2022, 10 pm (Week 13)
- ❑ Presentations: Wednesday, 13/04/2022 (Week 13)
- ❑ Final Report Submission: Wednesday, 20/04/2022, 10 pm (Reading Week)

GitHub Classroom

Use your GitHub repository for your coding! Not just as submission platform.



To get **feedback**: Assign one of the tutors as *Reviewer* or *mention* our GitHub names in comments/PRs.

Software (Unit) Testing

Goals of this lecture:

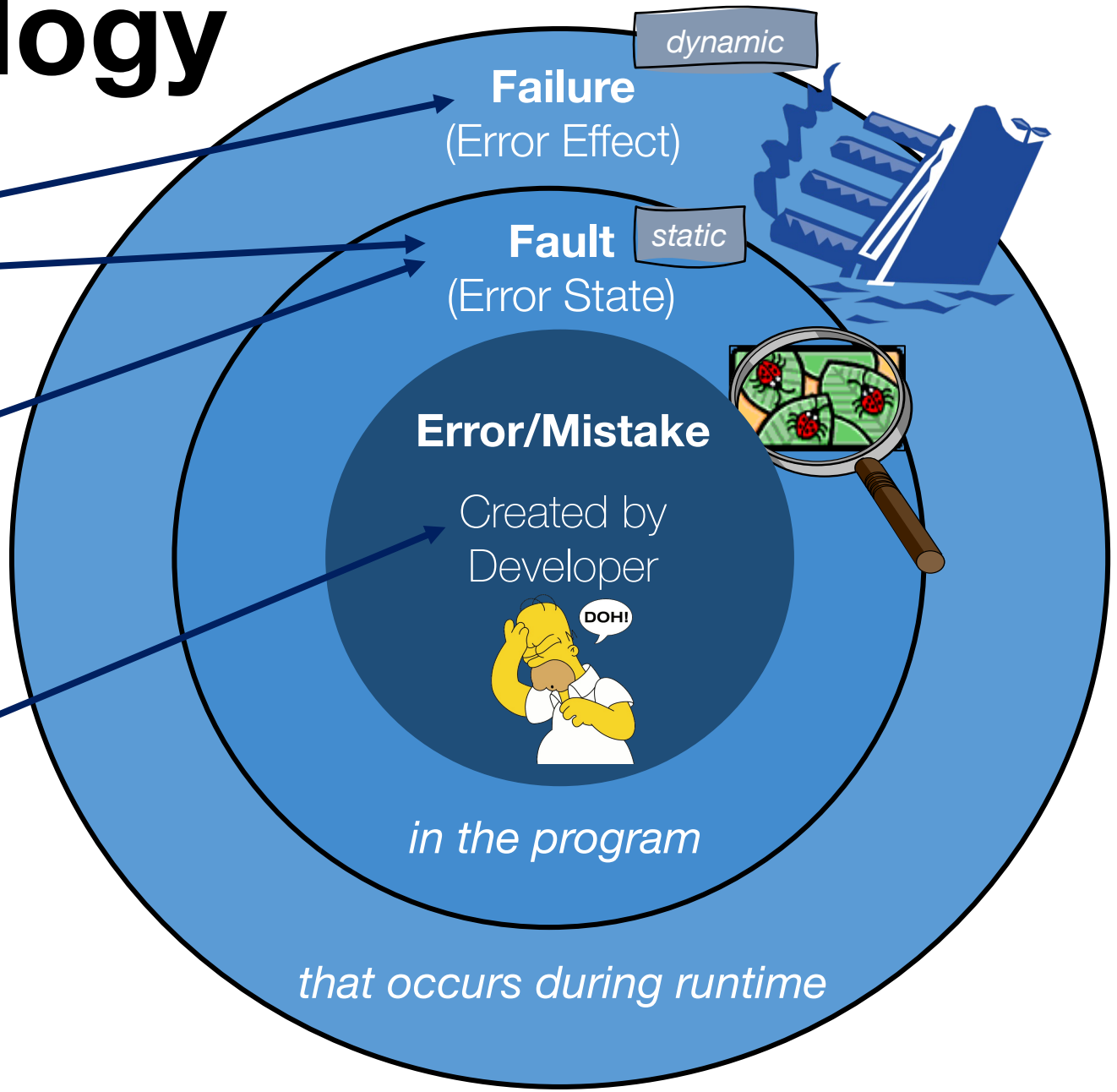
- ❑ Testing Terminology & Motivation
- ❑ Basic Testing Process
- ❑ Functional Unit Testing: Equivalence Class Analysis
- ❑ JUnit Testing

Testing Terminology

Verification + Testing

Debugging

Standards, Norms, Experience



Chain of Error

- All error conditions and defects occur during completion:
  - The error effect (failure) occurs during the execution of the program.
  - The error effect is generated by an error state (fault) in the software.
  - The error condition is triggered by the error action (mistake) of a human (programmer).
- Fault masking: “An occurrence in which one defect prevents the detection of another” [IEEE 610]
- To detect a fault state, the normal situation must be defined.

(Dynamic) Testing

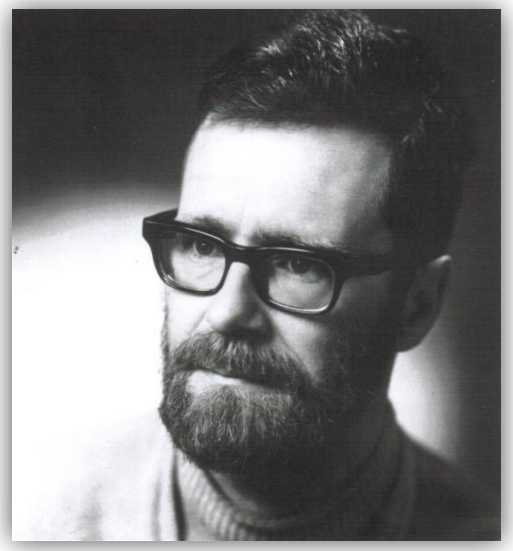
- Advantages
  - Testing is a *natural* validation procedure!
  - The test is (possibly) **reproducible** and therefore **objective**.
  - Test, once well organized, can be repeated very cheaply.
  - Target environment (translator, OS, etc.) is also checked.
  - System behavior is made **visible**.
- Disadvantages
  - Expressiveness of tests is overestimated, does not show correctness, because even the state spaces of small programs are huge.
  - Test says nothing about most software features.
  - It is not possible to replicate all application situations.
  - The test does not show the **cause** of the error.

Foundations of Testing

- Some fundamentals have been established over the last 50 years.

»Program testing can be used to show the presence of bugs, but never to show their absence!«

Edsger W. Dijkstra, 1970

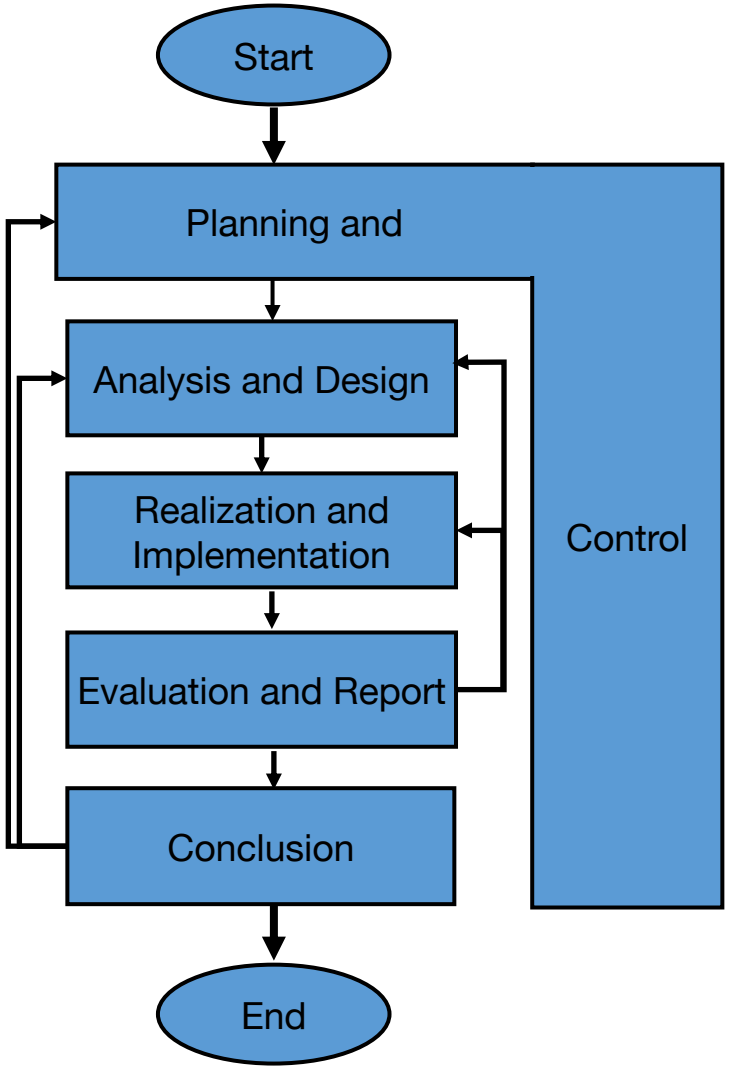


»Complete testing is not possible«
»Start as early as possible with testing«

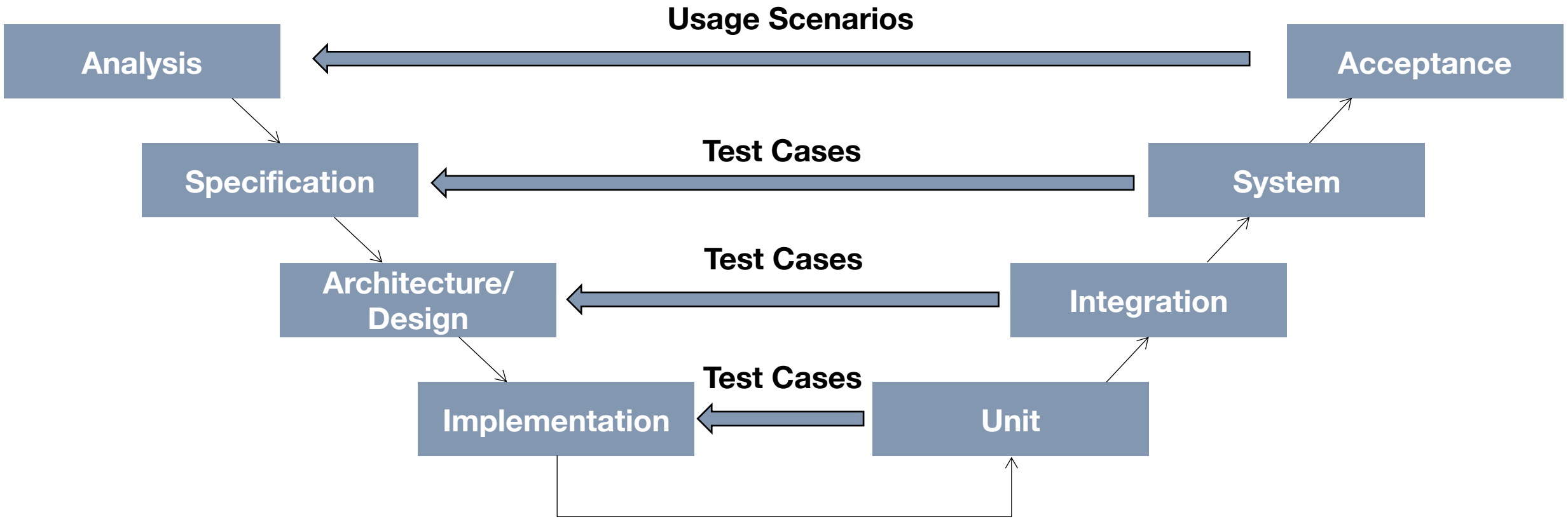
- Testing is not a late phase of the development process, but should be included as early as possible. The sooner errors are found, the lower the costs.

Testing: Basic Rules

- Testing (like all validation effort) must be **planned**.
- Testing requires **independence**.
- Testing is a **creative** and **challenging** activity.
- Testing is **destructive**.
- Every test case has an **expected outcome**.
- Complete testing is **impossible**.



Testing Levels



Rules for Test Execution

- ❑ **no abort** of the test after the first deviation
- ❑ **no special test variant** of the program
- ❑ **no switching** between test and debugging

Advantages:

- ❑ effort can be estimated
- ❑ identification of basic defects
- ❑ no cumulative corrections
- ❑ efficient use of the test harness
- ❑ no damage to the test item due to forgotten additives
- ❑ results concern the product

Testing Advice

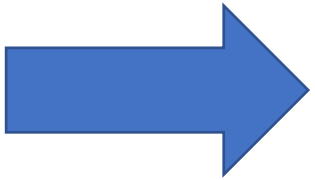
Regardless of the software development model chosen, the following aspects have proven useful for testing:

- For every development activity there is a corresponding test activity.
- Testing activities should start **early** in the development cycle. Test analysis and test design should begin in parallel with the corresponding development stage.
- Involve testers **early** in the review process of development documents.
- Software development models should not be used "out of the box". They must be adapted to project and product characteristics (e.g., number of test stages to be applied, number and length of iterations, etc. must be adapted per project context).

Goal of (Dynamic) Testing

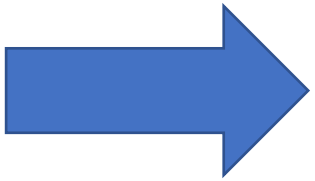
The goal of dynamic testing procedures is to generate test cases of a sample of the possible inputs that are:

- ❑ representative,
- ❑ error-sensitive,
- ❑ low redundancy and
- ❑ economical.



Black-Box Test
Functional Test

Test data selection according to the (targeted) characteristics of the program to be tested (function, response time), i.e., according to the **specification**.



White-Box Test

Test data selection under the influence of the internal structure of the program.

Equivalence Class Analysis (1/2)

- Creation of equivalence classes of input values based on the functional properties of the program or better its functional specification.
- Values from an equivalence class
  - cause identical functional behavior and
  - test an identical specified program function.
- Creation equivalence classes based on the specification ensures that all specified program functions are tested with values from their assigned equivalence class.
- Equivalence classes can also be created from the output value ranges.

Equivalence Class Analysis (2/2)

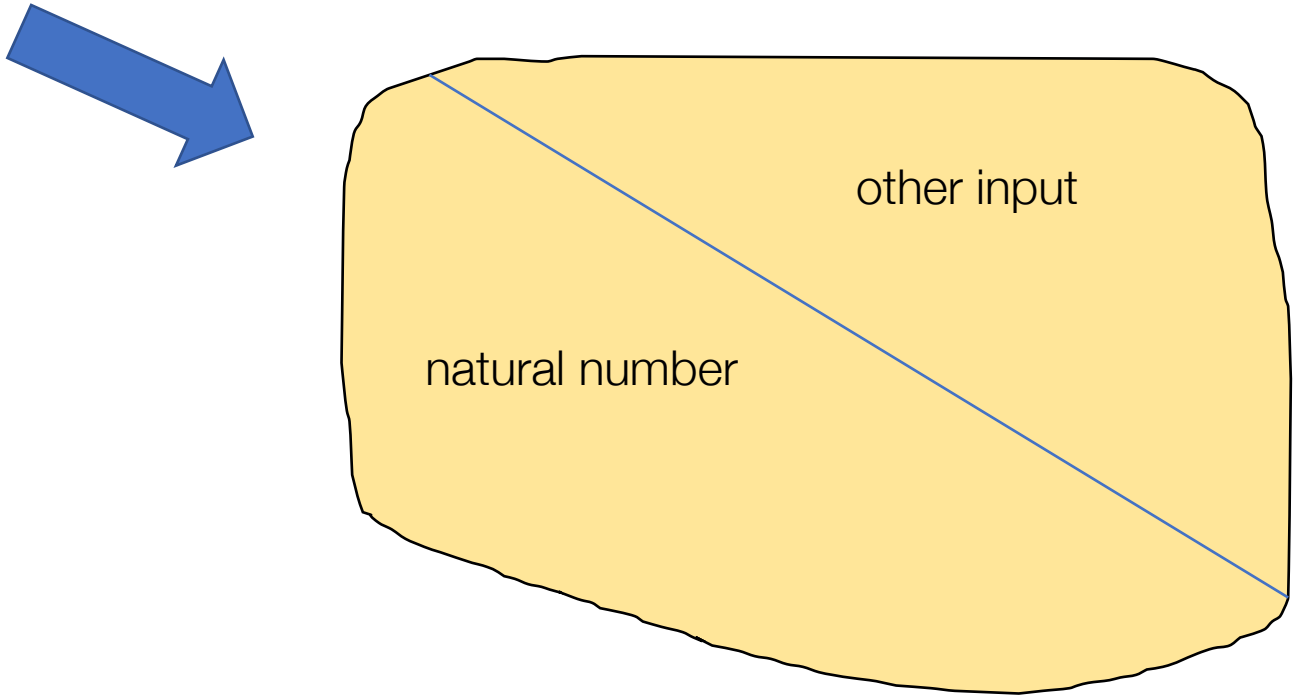
- Procedure (attributed to [Myers, 1979]): Equivalence Class Partitioning
  1. Identify: input variables, equivalence classes for valid and invalid inputs
  2. Divide classes further intuitively, if necessary
  3. Select input data for each class, determine expected outcomes
- The equivalence classes 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 1/4)

- A program to calculate the **factorial** of *n*.

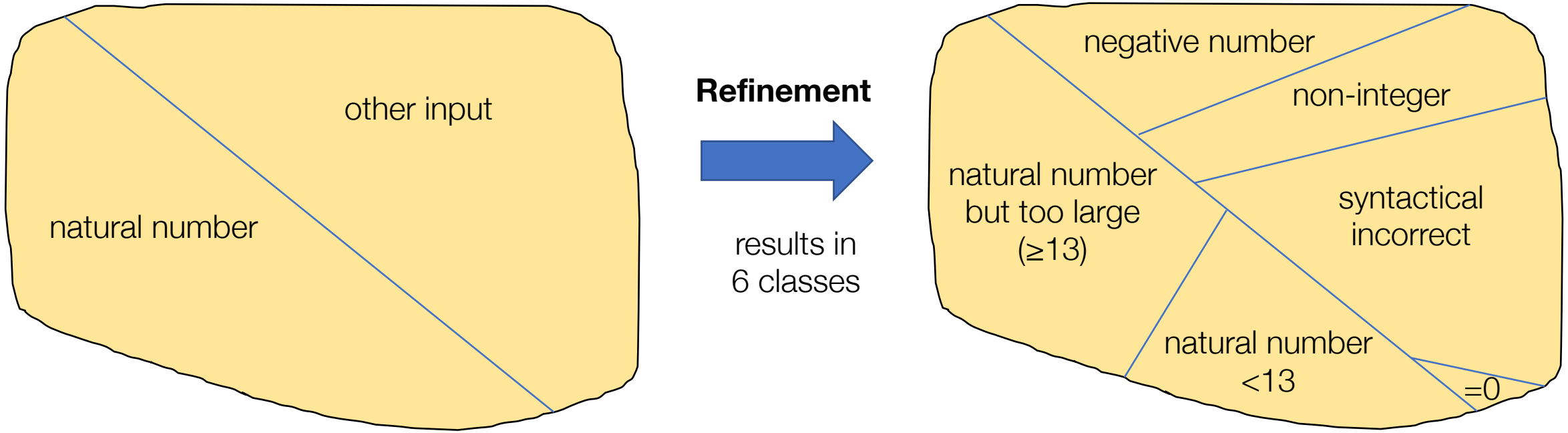
*"In mathematics, the factorial of a non-negative integer **n**, denoted by **n!**, is the product of all positive integers less than or equal to **n**."*

https://en.wikipedia.org/wiki/Factorial



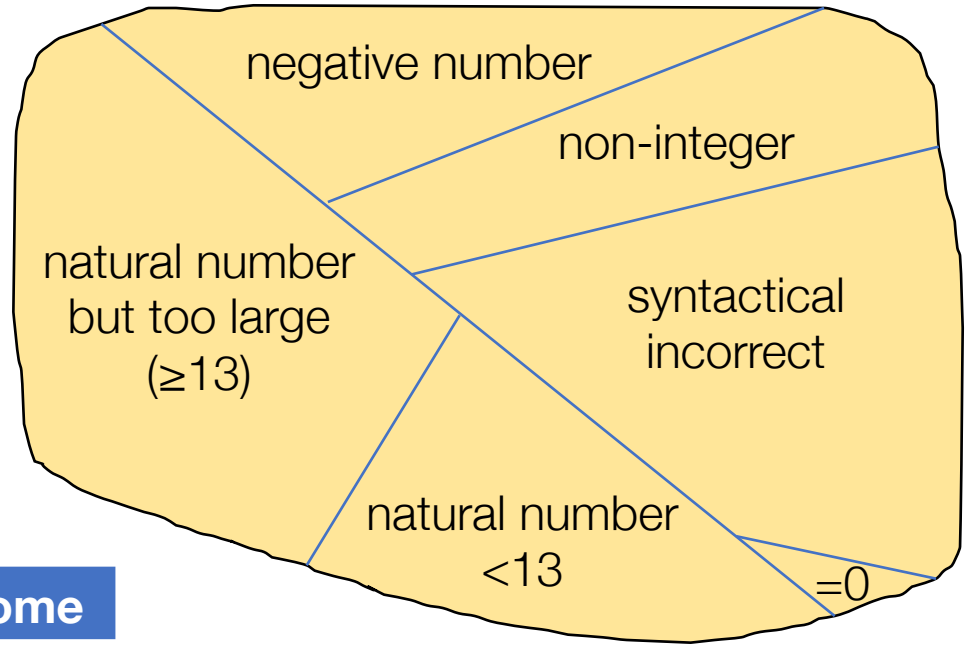
Equivalence Class Partitioning
(Example 2/4)

- 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*≥13), and (4) syntactically incorrect inputs. Special case: 0!



Test Case Selection
(Example 3/4)

- Choose representatives for each class and determine the expected outcome.

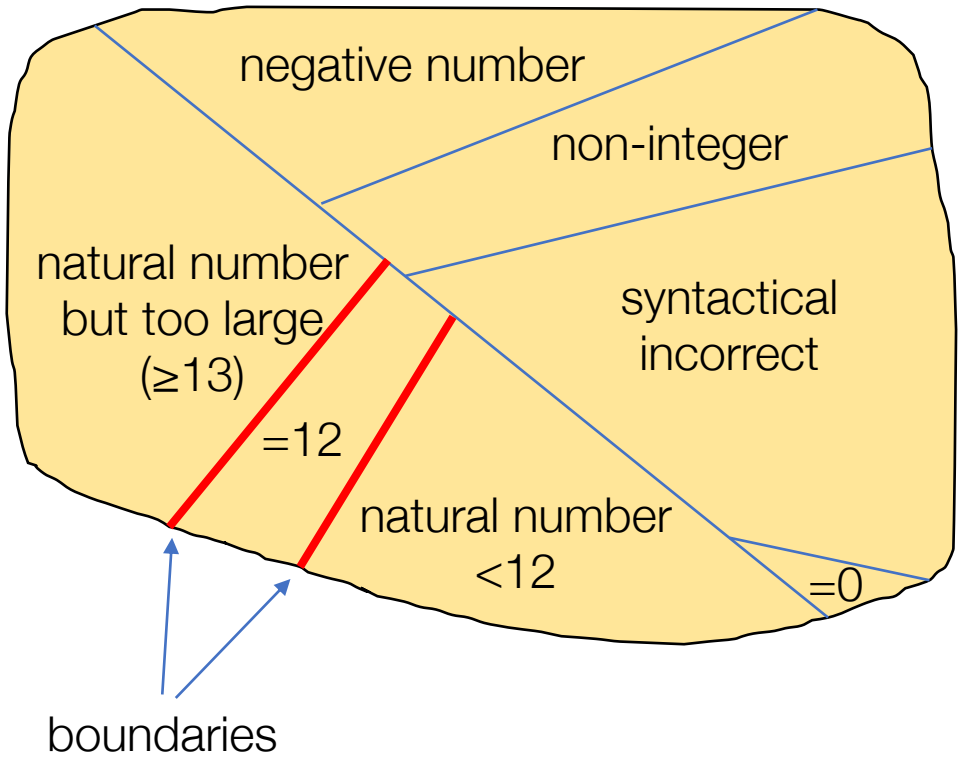


Class	Input	Expected Outcome
Negative number	-5	Error message
Non-integer	3.14	Error message
Too large number	100	Error message
Syntactical Incorrect input	“ABC”	Error message
Normal/expected input	7	5040
Zero	0	1

Boundary Value Analysis
(Example 4/5)

- For even better validation, in addition to a representative representative of an equivalence class, its limits should also be tested by limit values.
- Usually one uses a value below, on and above the limit.
- In the example we have the limit case 12, so besides 12 we should also test 11 and 13. The reason for the effectiveness of boundary tests is that in programming one often does not implement boundary cases correctly, for example by using ">" instead of "≥".

Boundary Value Analysis
(Example 5/5)



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

Any remaining question about Equivalence Class Testing?

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

(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.

The solution will be discussed in Week 7.

Unit Testing

"In this test, individual, manageable program program units are tested, depending on the programming language, e.g., functions, subroutines or classes."

Ludewig/Lichter, 2007

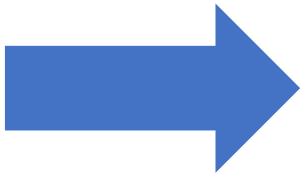
- Each component is tested individually, in isolation.
- Implemented software units are tested systematically.
- Error conditions can be clearly traced back to the source.
- Components can be interconnected, this is not considered in unit testing and only the component in itself is tested.
- Unit tests are based on the component specification, the code and all related documents.

Unit Testing in Java with JUnit

- JUnit is a unit testing framework for Java.
- It is now considered the standard for unit testing in Java.
- Originally developed by Kent Beck and Erich Gamma.
- Current version JUnit 5: http://junit.org/

Motto: „Keep the bar green to keep the code clean!“

Visualization by means of colored bar: if the test finds no errors, the bar turns **green**; a **red** bar indicates errors.

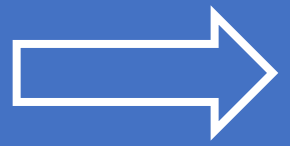


Examples and project-related workflow will be shown in the Lab!

Characteristics of JUnit

- Separation of application and test code.
- Test cases often structured in a separate class hierarchy.
- Individual test cases independent of each other (but can also be combined).
- Display of result immediately after execution by colored bar.
- Integration into many IDEs (e.g. Eclipse, IntelliJ, Netbeans, ...).

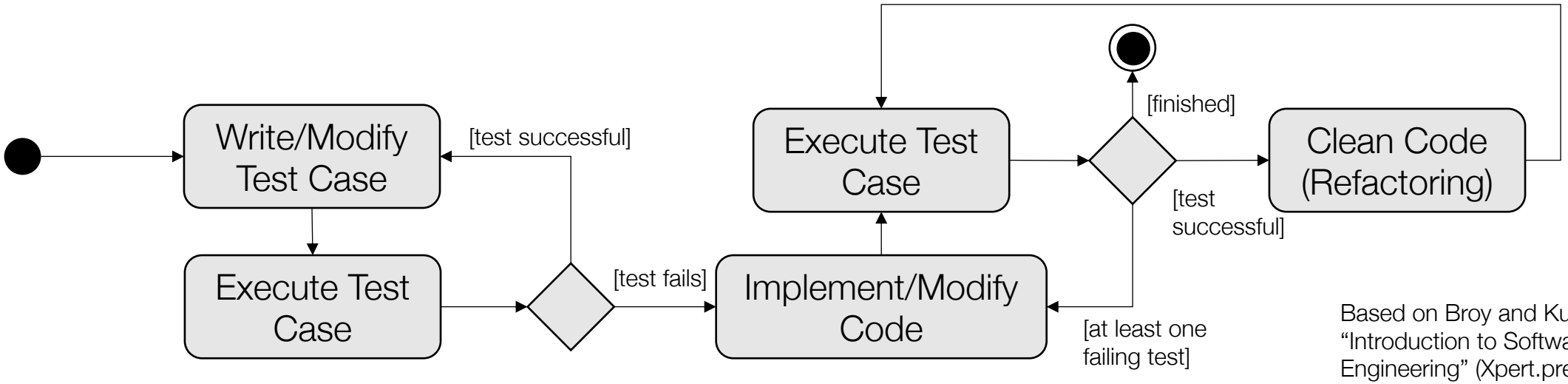
What are the attributes of a “good” Test Case?



The solution will be discussed in Week 7.

Test Driven Development (TDD)

- 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!

Conclusion

- ❏ Use Parser API to prepare test inputs.
- ❏ Next step: exploring the **solution space → start implementation**

Next Week: **Recess Week (no lecture/labs)**

In Two Weeks (Project-Part) – Week 7:
Advanced Unit Testing

- ➢ Discussion Testing Best Practices and Coverage Metrics
- ➢ Assignment 7: Unit Testing