Weekly Homework 9

Yevgeniy Terentyev, Miriam Gaetano
Softwaretechnik

June 13, 2025

Task 9-1: Terms

a) What is meant by the term “information hiding” in the context of software development? Explain the principle and explain how it relates to the “need to know” principle.

Information hiding is a key design principle that helps create robust, maintainable, and secure software systems by encapsulating the internal workings of components and exposing only what is necessary for interaction.

Information hiding and the ”need to know” principle work together to promote modularity, security, and maintainability in software development. By ensuring that components only expose what is necessary, both concepts help create systems that are easier to manage and less prone to errors or security vulnerabilities.

b) Why is the principle of information hiding useful? Explain this using an example.

By hiding implementation details, developers can reduce the complexity of the system. Users of a module do not need to know how it operates internally, which simplifies their interactions with it.

When implementation details are hidden, changes to a module can be made without affecting other parts of the system. This makes the software easier to maintain and evolve over time.

c) What is meant by the “design by contract” principle? In what context is it used and why?

It is a principle in software development methodology that establishes a formal agreement between software components, specifying the obligations and guarantees of each component. Contracts: In DbC, a contract consists of:

- Preconditions: Conditions that must be true before a method or function is executed. These are the responsibilities of the caller.

- Postconditions: Conditions that must be true after a method or function has executed. These are the responsibilities of the called method.
- Invariants: Conditions that must always hold true for a class or module throughout its lifetime.

DbC is often used in object-oriented programming languages, where classes and methods can have well-defined contracts and when designing application programming interfaces (APIs), DbC can help clarify how different components should interact. By specifying clear contracts, developers can ensure that components behave as expected, they also make it easier for other developers to understand how to use them. When changes are made to a component, the contracts help ensure that the changes do not violate the expected behavior. Contracts also allow multiple developers or teams to work on different components with a shared understanding of how those components should interact.

d) Explain the relationship between the terms OCL, invariant, design by contract, precondition, constraints and postcondition.

- Preconditions are a fundamental part of the contracts defined in DbC, ensuring that the method is invoked in a valid state.
- Like preconditions, postconditions are part of the contracts in DbC, ensuring that the method fulfills its obligations after execution.
- Invariants complement preconditions and postconditions by ensuring that the overall integrity of the class is maintained throughout its lifecycle.
- OCL can be used to express precondition, postconditions and invariants for classes in UML models.
- Constraints can be seen as a broader category that includes invariants, preconditions, and postconditions. They define the rules that must be adhered to within the system, ensuring that the software behaves as intended.

Task 9-2: Reading and writing OCL

a) Based on the diagram, express the OCL constraints c1 to c3 in natural language, i.e. in a form that you would use in a conversation with a person without a computer science background.

```
context Task inv c1: score > 0
```

Every task's score must be grater then 0.

```
context Student inv c2: solutions->size() = exam.tasks->size()
```

For every student the following must be always true: the number of solutions submitted must be equal to the number of tasks for a given exam. This means that a student must submit a solution for every task included in the exam, ensuring that no tasks are left unanswered.

```
context exam inv c3:
    participant->forAll (t |
        t.passed implies t.solutions->exists (l |
            l.points > 0 and l.task.exam = self
        )
    )
```

For each exam the following must be true: for all participants to the exam, if the participant has passed the exam it must be true also that he has submitted at least one solution, the points for that solution must be grater then 0 and the solution to the task must be associated with this exam.

In a bit simpler form: This means that if a participant is marked as having passed the exam, they must have submitted at least one solution that received a positive score for that specific exam.

Explanation regarding the implication: In the context of the OCL expression, the keyword `implies` represents a logical implication. The expression $A \implies B$ can be understood as follows: if A is true, then B must also be true. However, if A is false, the entire expression $A \implies B$ evaluates to true, regardless of the truth value of B. In this specific case, $t.passed$ is A, and the condition $t.solutions \to \text{exists}(l \mid l.points > 0 \wedge l.task.exam = \text{self})$ is B. Therefore, if $t.passed$ is false, the implication evaluates to true, meaning that the condition does not impose any requirement on the solutions submitted by the participant. The implication only requires that if a participant has passed, then they must have at least one solution with more than 0 points for the exam.

b) Specify OCL constraints that formalize the following facts. Make sure to formulate syntactically correct OCL expressions. This also includes observing the exact notation of classes, attributes, operations and associations from the UML class diagram.

1. in every exam there is at least one task with exactly one point.

```
context Exam inv: self.tasks->exists(t | t.points = 1)
```

2. a post-examination cannot have a post-examination.

```
context Exam inv: self.postExamination->isEmpty() or
self.postExamination.postExamination.isEmpty()
```