IT-Universitetet i København

Analysis Design and Software Architecture

Assignment 04

Group 39
Joachim de Fries Willumsen jwil@itu.dk
Jonas Lindvig jonli@itu.dk
Rasmus Balthazar Rødgaard Andreasen rban@itu.dk

October 14, 2021

Repository link:
`https://github.com/Rasmus-Balthazar/BDSA_Assignment04/tree/main`

1 Exercise 1

Study the meaning of encapsulation, inheritance, and polymorphism in OO languages. Provide a description of these concepts including UML diagram showcasing them Encapsulation is the action of minimizing of decoupling the system from the encapsulated component, thus minimizing the impact of changes to the system as the components are then less reliable on each other. An example could be the use of the Factory design pattern - a certain input is given and must result in a corresponding output, however, instead of the input being passed directly to the factory class(es) it is received by a preliminary class, which reads it and calls the relevant method. Changes in the input parameters would then no longer affect every factory, but only the receiver class.

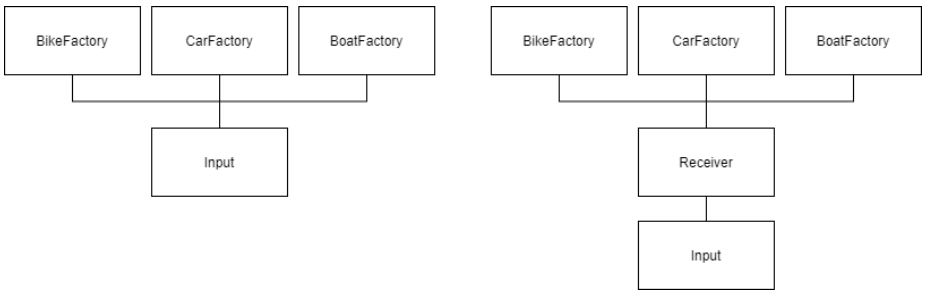

Figure 1: Encapsulation example

We use inheritance to classify objects into Super and subclasses to organize objects into an understandable hierarchy and to reduce redundancy and enhance extensibility of our code. An Example could be an organisation of cars. Instead of repeating given fields such as engine, wheel, door etc. we can simply create a superclass with such standard fields, and then set all models to be subclasses. We then only have to write/override fields which are different from the superclass

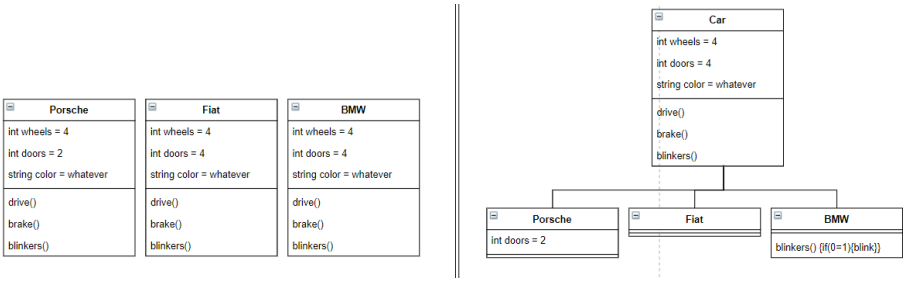

Figure 2: Inheritance example

Polymorphism is a concept in object oriented programming that lets us use the super class to refer to different sub-classes. This means we can have different things happen for each sub-class, through virtual dispatching, as they can have their own individual implementation of the super-classes.

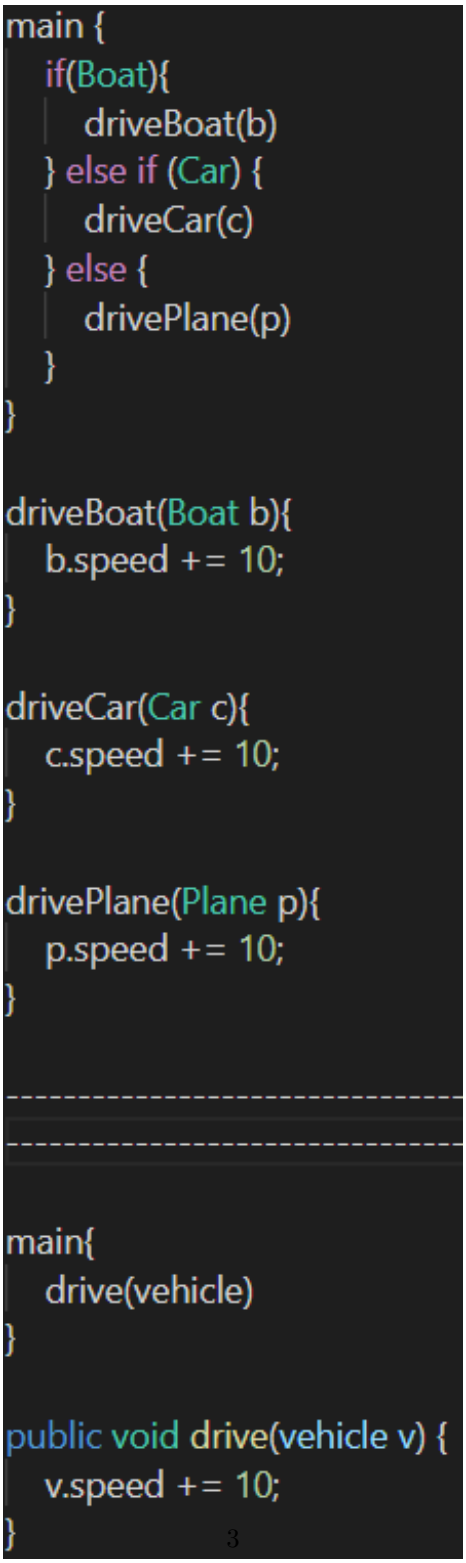

Figure 3: Polymorphism example

2 Exercise 2

As a maintainance activity (i.e., after having written the code): draw a class diagram representing your implementation of the entities for the C part. The purpose of the diagram should be to document the main relationships between the entities and their multiplicity

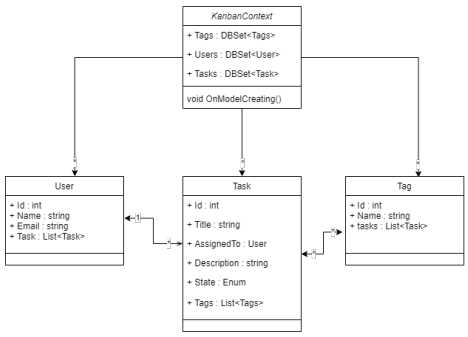

Figure 4: Kanban Class Diagram

3 Exercise 3

As a maintainance activity (i.e., after having written the code): draw a state machine diagram representing your implementation of the task entity. The purpose of the diagram should be to document the diffent states the entity can go through and the transitions that trigger the changes.

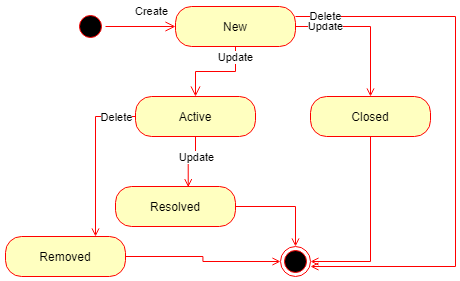

Figure 5: Task State Diagram

4 Exercise 4+5

For each of the SOLID principles, provide an example through either a UML diagram or a code listing that showcases the violation of the specific principle and provide a refactored solution as either a UML diagram or a code listing that showcases a possible solution respecting the principle violated. Note: the

examples do not need to be sophisticated.

Single Responsibility Principle: each class only does one thing and every class or module only has responsibility for one part of the software's functionality. More simply, each class should solve only one problem:

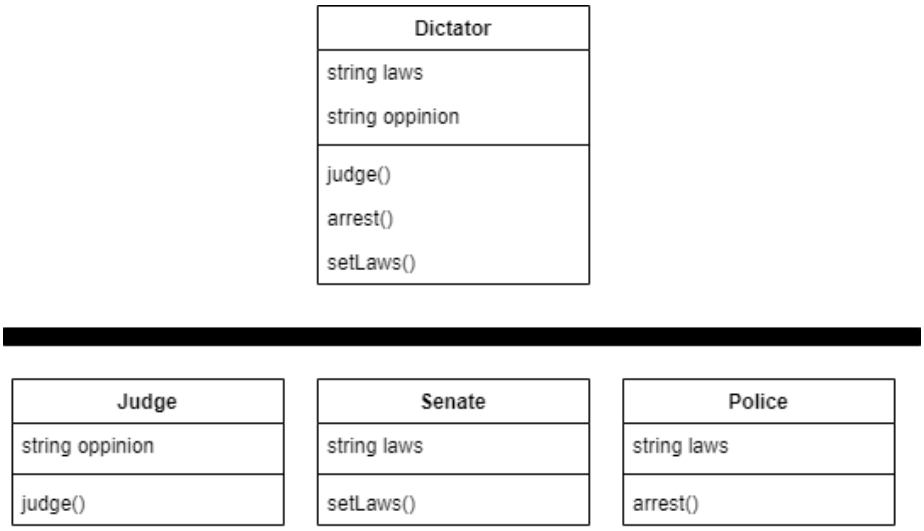

Figure 6: Single Responsibility Principle Example: Wrong in upper part, and corrected below

Open-Closed Principle: You should be able to extend a class's behavior without modifying it. - Your class complies with this principle if it is: Open for extension, meaning that the class's behavior can be extended; and Closed for modification, meaning that the source code is set and cannot be changed.

The way to comply with these principles and to make sure that your class is easily extendable without having to modify the code is through the use of abstractions. Using inheritance or interfaces that allow polymorphic substitutions is a common way to comply with this principle

OPEN CLOSED EXAMPLE

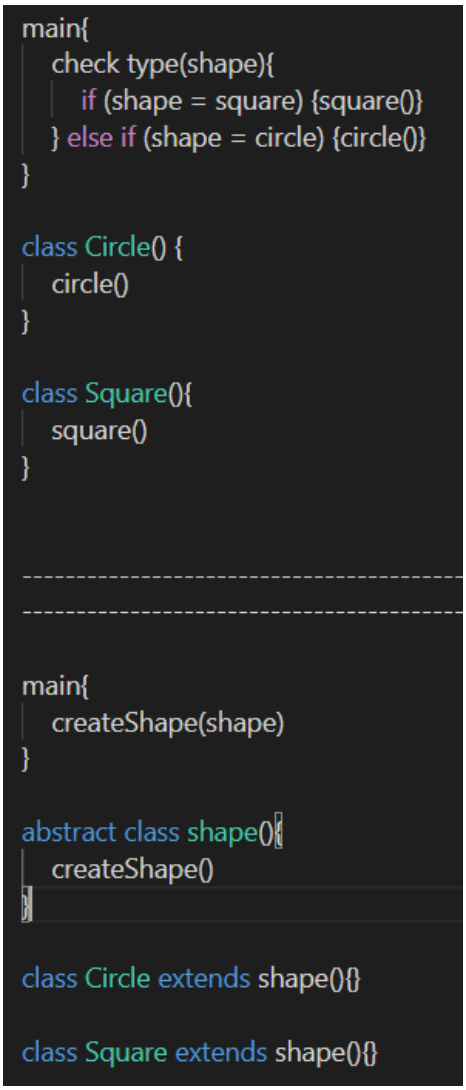

Figure 7: Open-Closed Principle Example: Wrong in upper part, and corrected below

Liskov Substitution Principle: every derived class should be substitutable for its parent class.

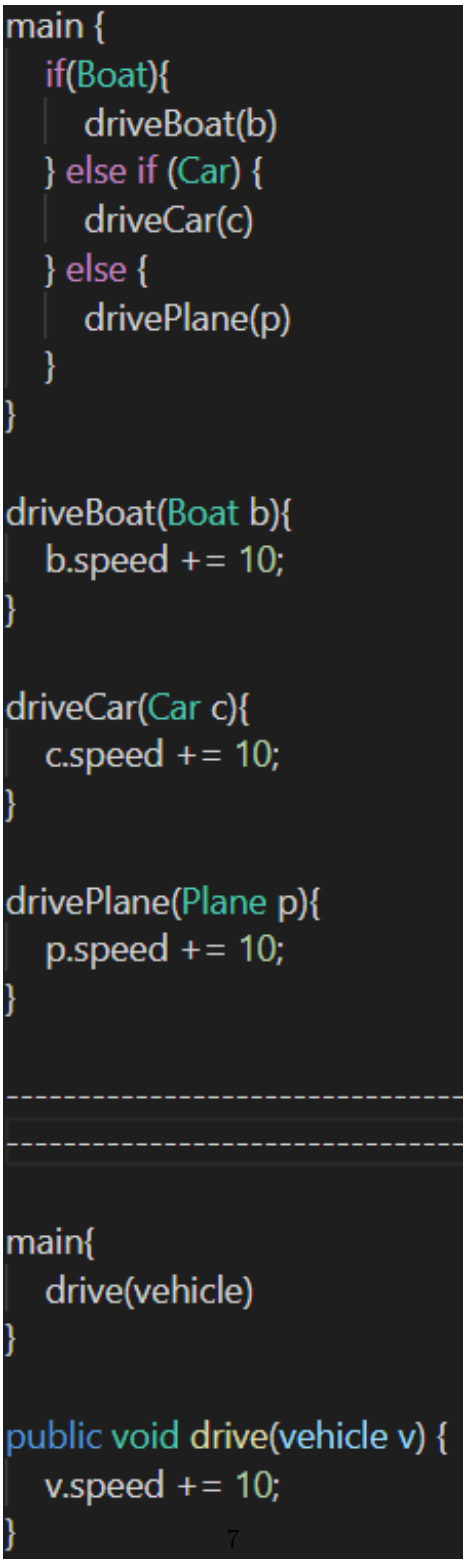

Figure 8: Liskov substitution Principle Example: Wrong in upper part, and corrected below

Interface Segregation Principle: it's better to have a lot of smaller interfaces than a few bigger ones.According to this principle, engineers should work to have many client-specific interfaces, avoiding the temptation of having one big, general-purpose interface.
INTERFACE SEGREGATION EXAMPLE

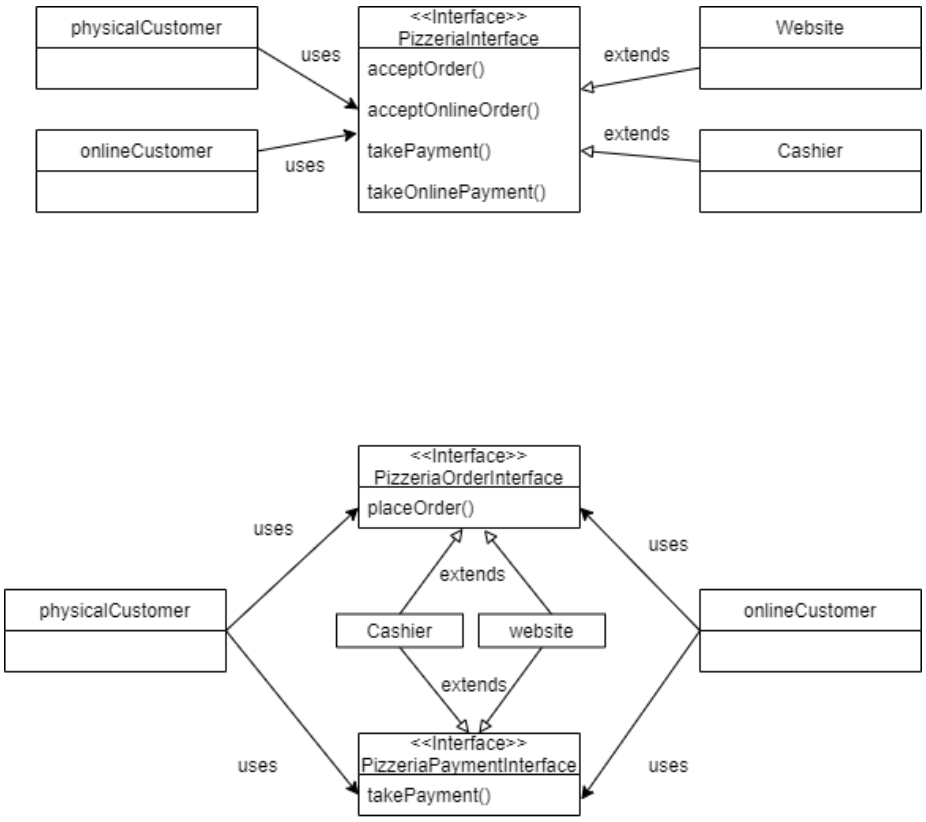

Figure 9: Interface segregation Principle Example: Wrong in upper part, and corrected below

Dependency Inversion Principle: depend on abstractions, not on concretions. high level modules should not depend upon low level modules. Both should depend on abstractions." Further, "abstractions should not depend on details. Details should depend upon abstractions

Dependency Inversion example

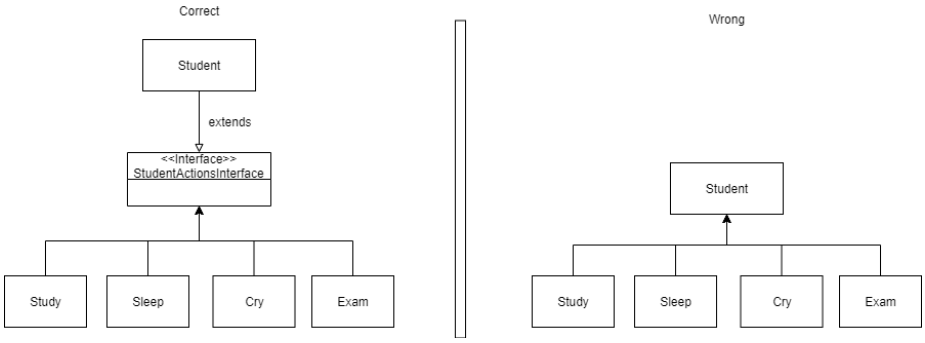

Figure 10: Principle Example: Wrong in right part, and corrected to the left

Source: https://www.bmc.com/blogs/solid-design-principles/