C# Part

The code repository is available to view on my personal GitHub account: link.

If the hyperlink doesn't work, then the full link is:

`https://github.com/andreaswachs/BDSA2021-AS04v2`.

Software Engineering

Exercise 1

Encapsulation

Encapsulation is the practice of using access modifiers for class fields and methods.

This allows for customized control over if a certain class allows the outside world to utilize certain methods and changing fields.

Encapsulation is used to hide implementation details such that a given class only exposes certain methods and hides away how it works internally.

This is great for *Client-Server architecture*, where a class acting as a server will only allow clients to use the selected methods for it to operate in a stable and safe manner.

Imagine a *greeter* class, where a field contains the name of the person to greet. Here, the designers of the class might want to have stricter control of what clients are allowed to set this field to. It might make sense to disallow it to be set to an empty string or `null`.

Then, the designers would implement a setter method to set the field with appropriate error handling and to finish it a method that invokes the greeting.

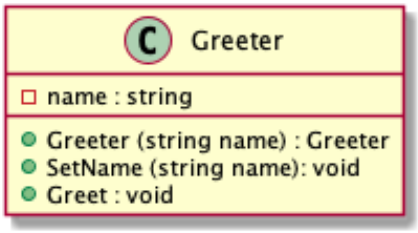


Figure 1: The `Greeter` class

Inheritance

Inheritance is the concept of having classes that inherits behavior from parent classes. The parent classes are called *super classes* and the child classes are called *sub classes*. When considering a full inheritance hierarchy, we call moving up the hierarchy for *generalizing* and moving down the hierarchy *specializing*.

For example, we can imagine different kinds of desserts. We can naïvely divide them into salty desserts and sweet desserts. Here we define more specialized cases of desserts. We can thus say that *"All salty desserts are desserts, but not all desserts are salty desserts"*.

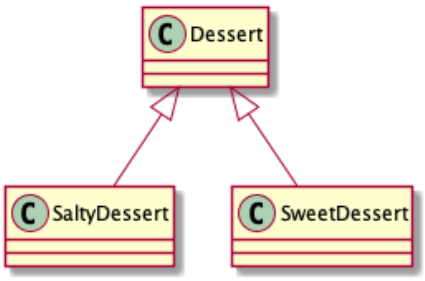


Figure 2: The class hierarchy for desserts

Polymorphism

Polymorphism is the concept of applying the same operations on different data types.

Polymorphism can come on several flavors, but commonly it is used where different classes implement the same interface, and they are used in a place where the interface is used amongst a collection of possibly different types.

Polymorphism is also used with generic types, where we can make data structures such as lists, that are able to work with any type, because it manages the concrete instances of the types, but doesn't really depend on what the data types "can do".

I've attempted to illustrate this in the image to the right. Here we see an interface defined to contain the method `Greet`. Anyone who implements this interface will promise to be able to make a greeting. This can be classes of disjoint inheritance hierarchies.

The `GreeterConsumer` is a third party entity that is able to consume these `Greetable` entities and invoke their independently implemented *Greet* methods.

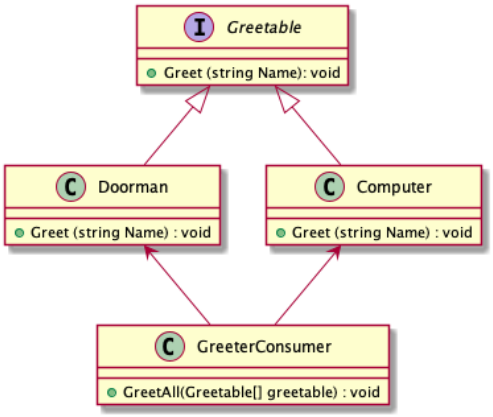


Figure 3: Polymorphism in action with a commonly implemented interface

C# Entities and their relationships

The relationship shown in figure 4 shows the multiplicity between `Users`, `Tasks` and `Tags`.

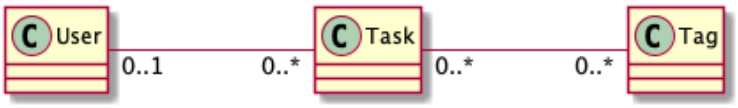


Figure 4: The relationship between Users, Tasks and Tags with a focus on their multiplicity

Task lifecycle and their State

The following state diagram shows how a given Task's State will change over time in response to different operations.

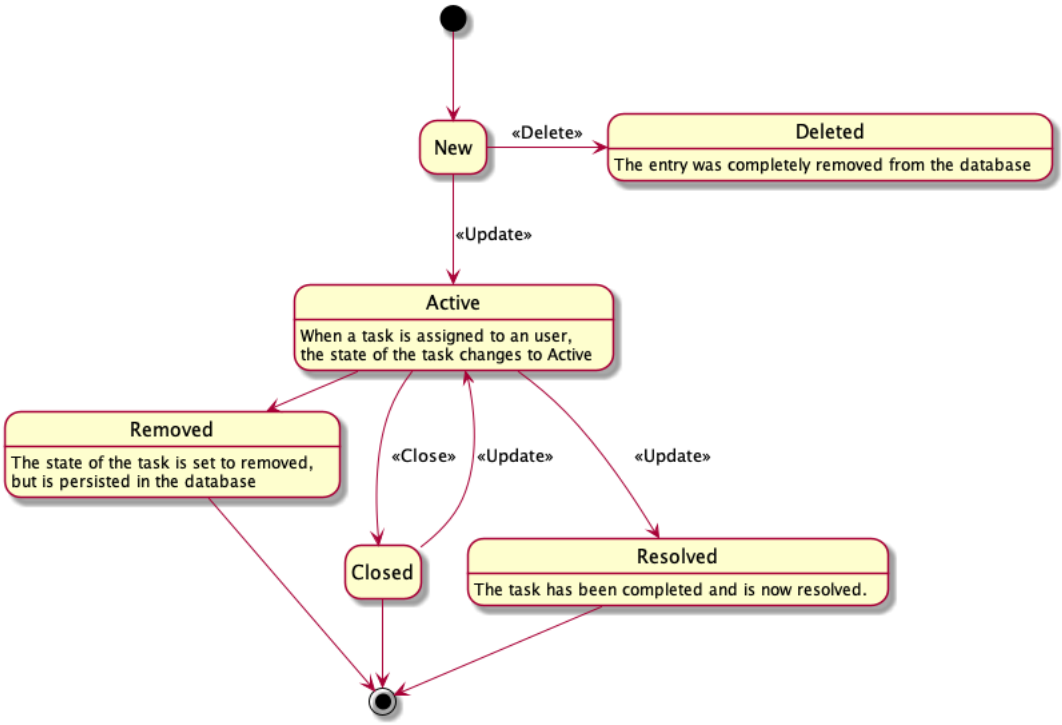


Figure 5: The lifecycle of a Task and its State

Examples of violations of the SOLID principles

Single Responsibility principle

Violating the single responsibility principle means that classes will hold more than one responsibility in terms of what they are used for. This is illustrated below:

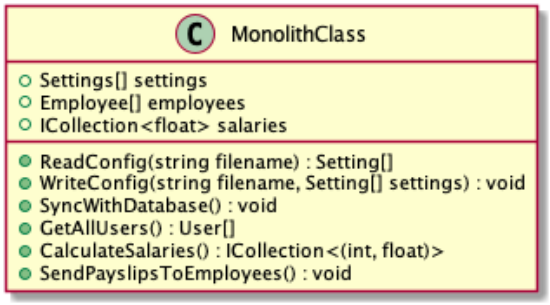


Figure 6: Violation of the Single Responsibility principle.

Open/closed principle

In object oriented programming, the principle about being allowed to extend existing functionality, but being prevented to modify it can be shown with a simple inheritance hierarchy.

To violate this principle, the designers of the general `Animal` class have not paid attention to sue of inappropriate access modifiers, such that sub classes of `Animal` can modify the previously defined methods and fields coming from the super class.

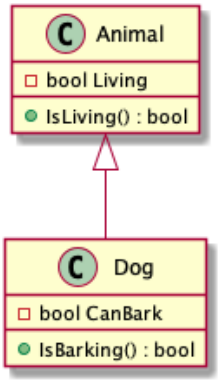


Figure 7:

Liskov's substitution principle

The following source code illustrates the missing use of Liskovs' substitution principle, where two classes of disjoint inheritance hierarchies implement the same function.

```
class Dog
{
    public void Report()
    {
        Console.WriteLine("I'm a dog!");
    }
}

class Cat
{
    public void Report()
    {
        Console.WriteLine("I'm a cat!);
    }
}

class AnimalFactory
{
    public void reportAnimal(Object animal)
    {
        if (animal is Dog)
        {
            Dog dog = (Dog) animal;
            dog.Report();
        } else if (animal is Cat)
        {
            Cat cat = (Cat) animal;
            cat.Report();
        } else {
            Console.WriteLine("What animal even is this??");
        }
    }
}
```

Interface segregation principle

Here is an interface with too many different responsibilies, all that ultimately will serve a client, but they are of different responsibilies.

```
interface IClientServable
{
  Report GetReport(int id);
  Report[] AllReports();
  int64 CalculateMonthlyPaycheckSum();
  int64 CalculateMonthlyCostsForHousing();
  Employee GetEmployee(int id);
  Employee[] AllEmployees();
  void SetDbContext(DBContext context);
  void SetLogger(Logger logger);
}
```

Dependency Inversion principle

Here is a break on the Dependency Inversion principle, where another class, the `Logger` class is being composed by the `Worker` class to enable use of logging.

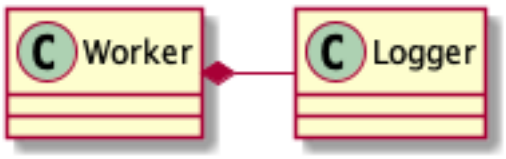
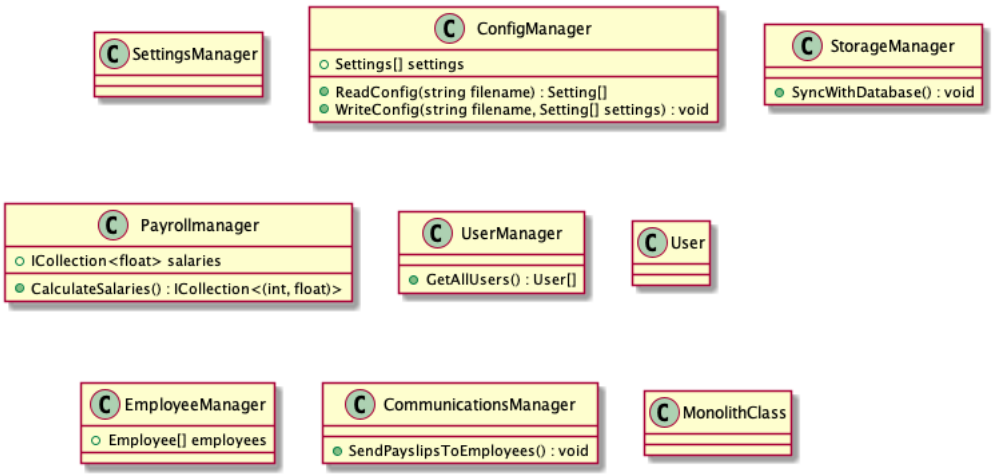


Figure 8: The worker uses the Logger class throughout its lifecycle

Solutions to the examples of broken SOLID principles

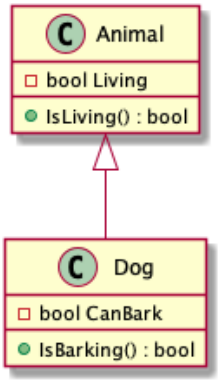
Single Responsibility principle

We split the monolith class into different classes based upon which methods share the same context of kind of responsibility.



Open/closed principle

We disallow the Dog sub class to be able to override the `IsLiving()` method and `Living : bool` field from the Animal super class.



Liskov's substitution principle

We fix the violation of the principle by using inheritance:

```
  abstract class Animal
  {
    public virtual void Report();
  }

  class Dog : Animal
  {
      public void Report()
      {
          Console.WriteLine("I'm a dog!");
      }
  }

  class Cat : Animal
  {
      public void Report()
      {
          Console.WriteLine("I'm a cat!);
      }
  }

  class AnimalFactory
  {
      public void reportAnimal(Animal animal)
      {
          Animal.Report();
      }
  }
```

Interface segregation principle

We fix the violation of the interface segregation principle by separating the giant monolithic interface smaller interfaces with higher cohesion:

```
interface IReportable
{
  Report GetReport(int id);
  Report[] AllReports();
}

interface ICalculateable
{
  int64 CalculateMonthlyPaycheckSum();
  int64 CalculateMonthlyCostsForHousing();
}

interface IEmployeeQueryable
{
  Employee GetEmployee(int id);
  Employee[] AllEmployees();
}

interface IDbConfigurable
{
  void SetDbContext(DBContext context);
}

interface ILoggerConfigurable
{
  void SetLogger(Logger logger);
}
```

Dependency Inversion principle

We employ constructer dependency injection to dynamically set the logger to use that implements the `ILoggerable` interface.