Assignment4

mlth, oljh

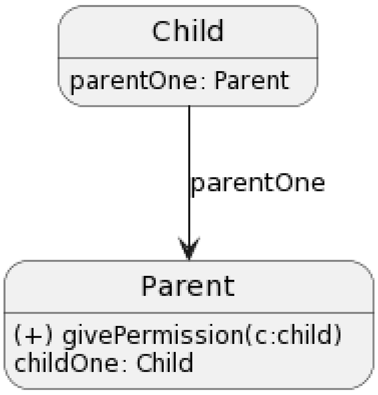
October 2022

1 C#

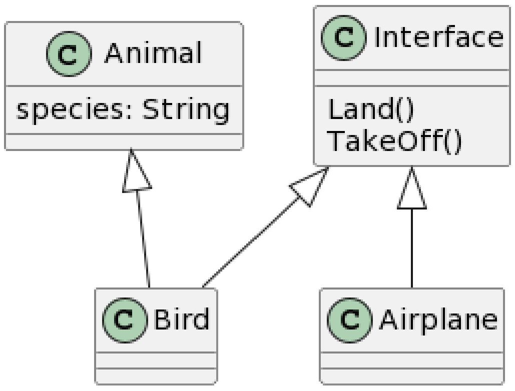
2 Software Engineering

2.1 Exercise 1

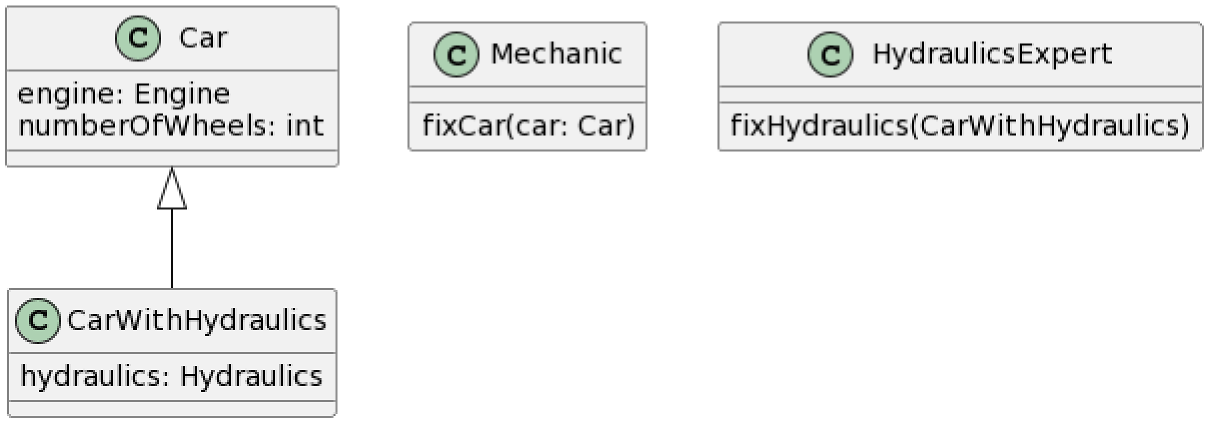
Encapsulation: Encapsulation refers to the idea of bundling data and methods together in different modules, like for example classes. This can create boundaries around objects, so that other classes cannot interfere with the data inside the object. If for example we have a Parent class and a Child class, and the Parent class has a givePermission method, that takes a Child object and gives that child permission to do some action. The Child object should not be able to call the givePermission method, since it would then not need the parents permission. Here is a class diagram showing the scenario where the child has a parent, on which it can call the givePermission method to give itself permission. To prevent this, the method could be made private, so it would only be the parent that could give permission to their children.



Inheritance: Inheritance refers to the idea of modules lending some structure from other modules. That can for example be a subclass that have some similarities with a superclass, but adds some extra features to the so called parent-class. This can prevent a lot of code duplication. In C# you can also use interfaces to further implement inheritance. Interfaces have the advantage that a class can implement more than one interface, but can only extend one other class. If for example we have a superclass called Animal, there are lots of other classes that could extend this one, but only some of them are able to fly. So we could make a bird class that Extended the Animal class with land() and takeOff() methods. However, other objects than just animals can fly. If we wanted to extend our program to contain airplanes, we would also want that class to contain the methods. So not we want to move the land() and take() methods into a class of its own, lets call it Flyable, so that the Bird class can extend both the Animal and Flyable class. However, since a class can only extend one class, we change the Flyable class to be an interface. This can be seen on the following class diagram:



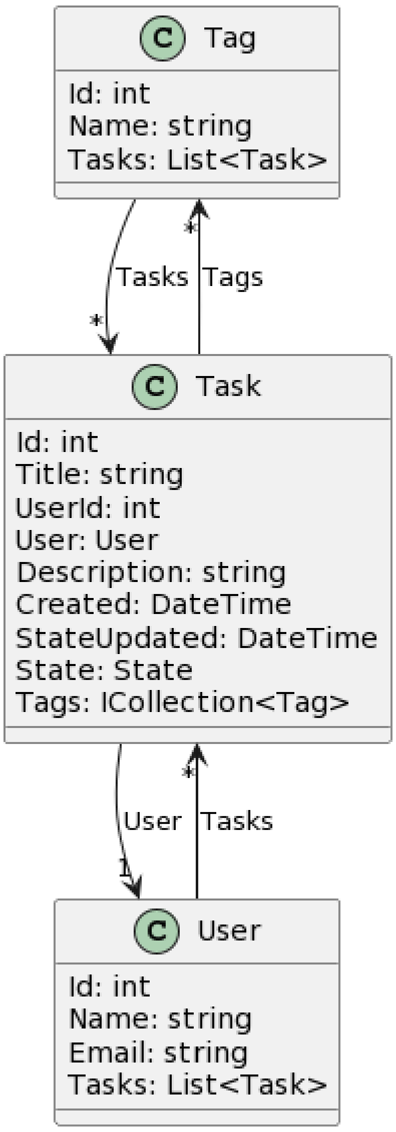
Polymorphism: Polymorphism refers to the idea that all places where a class is needed, a class that extends this class, can be inserted instead. Subclasses inheret all the features of their parent, and therefore it makes sense that they can do the same tasks as their parent. This does not go the other way. A superclass cannot replace a subclass, since the subclass might have extended functionality that is not in the superclass. If for example we have a Car class that is extended by a CarWithHydraulics class, and two additional classes: A Mechanic class and a HydraulicsExpert class. The mechanic can fix both cars, since he knows everything about cars, including hydraulics. The HydraulicsExpert can only fix the CarWithHydraulics. This scenario is shown in the following class diagram:



So, the Mechanic's fix method can take both a Car and a CarWithHydraulics object, but if a Car was passed to the HydraulicsExpert's fixHydraulics method, an error would occur, since the Car class does not contain hydraulics.

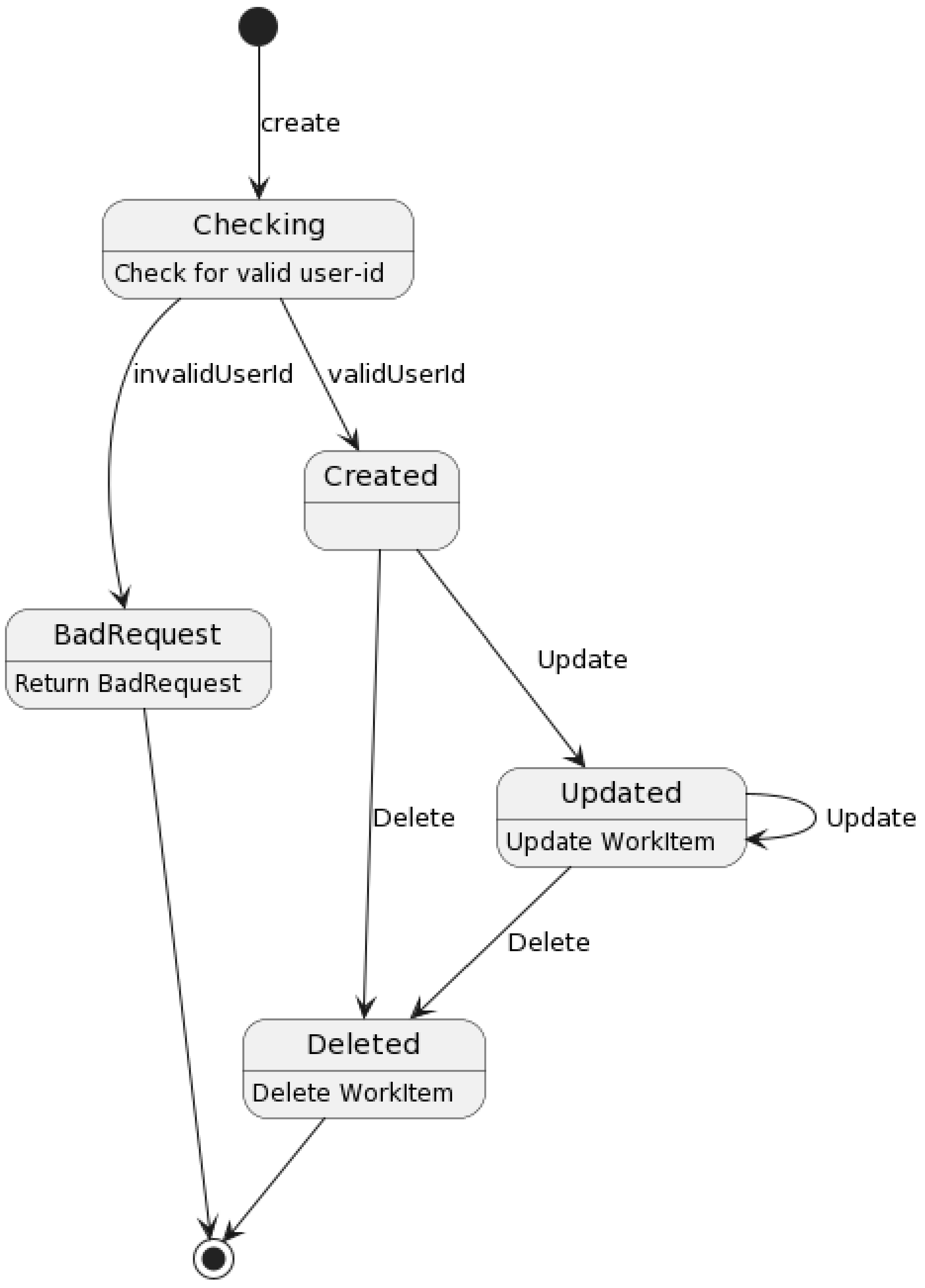
2.2 Exercise 2

Link for diagram: //www.plantuml.com/plantuml/png/XL6zQiD03Dxr5C9Z8KDN4K8WwH0eNUeym64B8t9t2ITrAdttdcnmRt398idwVgoFEwg5ObyfmX23AlPX-WQyZ8II3V0Z92PKAv8dgA3UbF1TrFO-7-07LftUMc4l5XlbLRcic7_GN07Fh4EHXyaz_x5FXODnfPzhwoKnuAVLyVeOD_UqDF3hbV1okiV8WtlkwwQvql_vVxot52G-ILlzSBSxOFVIpGHQQ0ZU6cGYBzgiNxisk9Sj8u0Zvx6-noy0



2.3 Exercise 3

Link for diagram:// Single Responsibility Principle //www.plantuml.com/plantuml/png/XOwn 3i8m34HtVuLdfFv01rG1IzTA5GDYY6e3KTCKd8JlfoI5b2vCkVYTVLSvhyI7mS3-nbslxRN4g F0o2ZwLqOJ1iHIQOAUeuKTWvqjisOUnc4TmOCEUgSGay3XANtiUeBsJYY293E2qEMDHR357TyA JvTibKU6q7zkqZgMcx8YMPJDjbxvCthuf4EPpSM_L-OUjQaDw_hFrpVcFd_oABON1l000



2.4 Exercise 4

2.4.1 Single Responsibility Principle

```
public static int CalculateSquarePlusHalf(int x) => x * x + x / 2;
```

This method has two responsibilities. It both Calculates the square of the number and adds half of the original number. This confusing, since the method signature is somewhat vague about what "half" actually means. Furthermore it violates the Single Responsibility Principle. Note, however, that in practice, this may be fine. As Martin states in APPP: "An axis of change is an axis of change, if the changes occur."

If we want to add the double amount, or we want to cube the number first, the class will have more than one reason to change.

2.4.2 Open/Closed Principle

Consider the method below:

```
public static List<Ingredient> GetIngredients(Drink drink)
{
    switch(drink)
    {
        case RumNCoke:
            return new List<Ingredient>
            {
                Ingredient.Rum,
                Ingredient.Coke,
                Ingredient.Ice
            };
        case GinNTonic:
            return new List<Ingredient>
            {
                Ingredient.Gin,
                Ingredient.Tonic,
                Ingredient.Ice
            };
        case Margarita:
            return new List<Ingredient>
            {
                Ingredient.Tequila,
                Ingredient.TripleSec,
                Ingredient.Ice
            };
        default:
            return new List<Ingredient> {};
    }
}
```

Drink and Ingredients are enumerators, containing all possible drinks and ingredients. By itself, this is fine, however, if we want to add another drink, for example a RumNTonic, we would have to add another case two the switch statement. Likewise, if we changed the recipe of RumNTonic to not include ice (Ice is expensive!), then we would need to modify the switch statement. The switch statement resides in the GetIngredients() method, so changing a recipe has repercussions in a different module, than you might expect. This is a violation of OCP.

2.4.3 Liskov Substitution Principle

Look at the listing below:

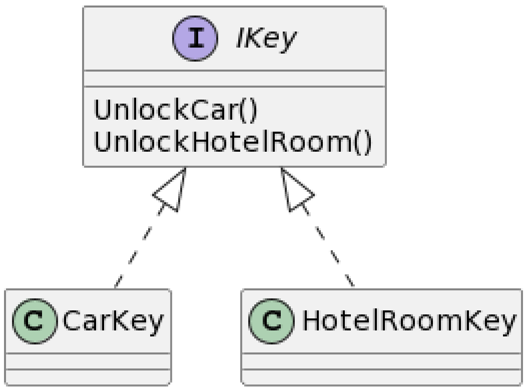
```
public class LSP {
    // Liskov Substitution Principle
    public static void MakeAnimalSpeak(Animal animal)
    {
        if(animal.GetType() == typeof(Bird))
            (animal as Bird).Chirp();
        else if(animal.GetType() == typeof(Dog))
```

```
            (animal as Dog).Bark();
    }
}
public abstract class Animal
{
}
public class Bird : Animal
{
    public void Chirp() => Console.WriteLine("Chirp chirp");
}
public class Dog : Animal
{
    public void Bark() => Console.WriteLine("Bark Bark");
}
```

When we call the method MakeAnimalSpeak(Animal animal), the method checks whether the instance of Animal is actually an instance of dog or bird. If it is a dog, it calls the Bark() method on the dog. If it is a bird, it calls the Chirp() method on the bird.

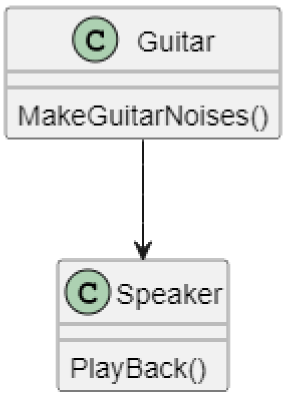
This method violates not only LSP, since the Animal is not actually substitutable for its subclasses, it also violates OCP and SRP. OCP, because we have to change the method when we add a new animal, and SRP because MakeAnimalSpeak(Animal animal) also checks the type of the animal.

2.4.4 Interface Segregation Principle



The IKey interface requires the classes with implement it to include both UnlockCar() and UnlockHotel-Room(). This violates the interface segregation principle, since a CarKey should not be able to open a Hotel Room.

2.4.5 Dependency Inversion Principle



In this example, a Speaker listens to the guitar and plays back it sounds when it is played. This may seem like a good solution, however, since the guitar is connected to the speaker, any changes made to how the speaker works, may affect the way the guitar works. This is a violation of DIP.

2.5 Exercise 5

2.5.1 Single Responsibility Principle

```
public static int CalculateSquare(int x) => x * x;
public static int CalculateHalf(int x) => x / 2;
```

By splitting this up into two functions, we have not only made the method signatures easier to read, since they do exactly what they say, they also only do one thing each, meaning the Single Responsibility Principle is enforced.

2.5.2 Open/Closed Principle

```
public static IList<Ingredient> GetIngredientsRefactor(Drink drink)
{
    return DRINKS[drink].Ingredients;
}

public interface IDrink
{
    IList<Ingredient> Ingredients { get; }
}

public class RumNCoke : IDrink
{
    public IList<Ingredient> Ingredients => new List<Ingredient>
    {
        Ingredient.Rum,
        Ingredient.Coke,
        Ingredient.Ice
    };
}

public class Margarita : IDrink
{
    public IList<Ingredient> Ingredients => new List<Ingredient>
    {
        Ingredient.Tequila,
        Ingredient.TripleSec,
        Ingredient.Ice
    };
}

public class GinNTonic : IDrink
{
    public IList<Ingredient> Ingredients => new List<Ingredient>
    {
        Ingredient.Gin,
        Ingredient.Tonic,
        Ingredient.Ice
    };
}


private static readonly Dictionary<Drink, IDrink> DRINKS =
    new Dictionary<Drink, IDrink>()
{
    [Drink.RumNCoke] = new RumNCoke(),
    [Drink.GinNTonic] = new GinNTonic(),
    [Drink.Margarita] = new Margarita()
};
```

In the Listing above, we implement the interface IDrink in three new classes RumNCoke, Magarita and GinNTonic. The IDrink interface only knows that it has a property Ingredients, containing a list of ingredients.

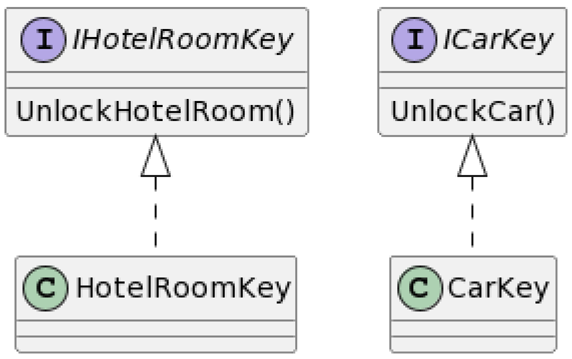
A Dictionary, where the keys are the enumerator Drink and the values are the new classes. Now we can translate the Drink enumerator to a class implementing IDrink. Our actual GetIngredients() method is now called GetIngredientsRefactor(). It is much smaller and completely closed. It could be argued, though, that new modules have been opened. You can never completely close a system, only decide which modules you wish to have open.

2.5.3 Liskov Substitution Principle

```
public class LSPRefactored {
    // Liskov Substitution Principle - refactored
    public static void MakeAnimalSpeakRefactor(AnimalRefactor animal)
    {
        animal.Speak();
    }
}
public abstract class AnimalRefactor
{
    public abstract void Speak();
}
public class BirdRefactor : AnimalRefactor
{
    public override void Speak() => Console.WriteLine("Chirp chirp");
}
public class DogRefactor : AnimalRefactor
{
    public override void Speak() => Console.WriteLine("Bark Bark");
}
```

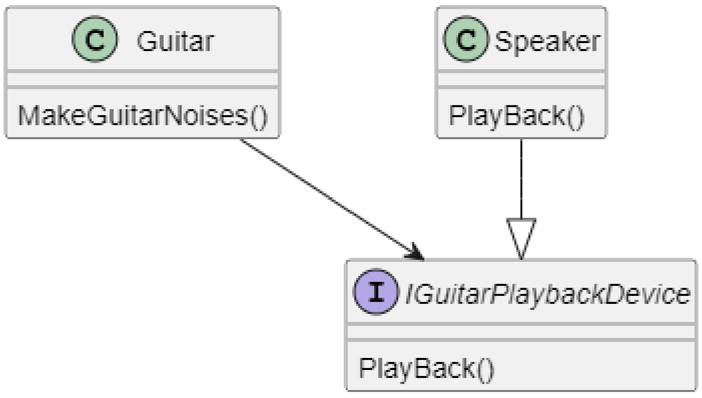
The abstract Animal class now has an abstract method Speak(). This method replaces the Bark() and Chirp() methods in the subclasses Dog and Bird. Now our MakeAnimalSpeak method is much shorter and only has one responsibility. If it another animal should be added, no modification to MakeAnimalSpeak is needed. Therfore it now enforces the OCP, SRP and LSP principles.

2.5.4 Interface Segregation Principle



By ”splitting” the interface into two, a CarKey no long has the UnlockHotelRoom() method, meaning ISP is enforced.

2.5.5 Dependency Inversion Principle



Instead of the guitar directly having a speaker, it instead has an IGuitarPlaybackDevice. The speaker then implements this interface. This way, it does not matter whether the speakers implementation changes, as long as it has that method.