|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| The SOLID Principles |

|  |
| --- |
| By Per Laursen  18-03-2019 |

[Introduction 3](#_Toc3820704)

[Dependency inversion (a.k.a. Inversion of Control) 5](#_Toc3820705)

[IoC by use of Template Method pattern 7](#_Toc3820706)

[The Dependency Injection (DI) principle 9](#_Toc3820707)

[Dependency Injection – parameter level 10](#_Toc3820708)

[Dependency Injection – method level 13](#_Toc3820709)

[Dependency Injection – object level 15](#_Toc3820710)

[IoC by use of Dependency Injection 17](#_Toc3820711)

[The Single Responsibility Principle 18](#_Toc3820712)

[The Open/Closed Principle 22](#_Toc3820713)

[The Liskov Substitution Principle 27](#_Toc3820714)

[Definition 27](#_Toc3820715)

[Intentions and Contracts 28](#_Toc3820716)

[Preconditions 30](#_Toc3820717)

[What is inheritance actually? 31](#_Toc3820718)

[Dealing with non-compliance 32](#_Toc3820719)

[Postconditions 36](#_Toc3820720)

[Invariants 38](#_Toc3820721)

[Are we done? 40](#_Toc3820722)

[Final words, for now… 41](#_Toc3820723)

[The Interface Segregation principle 42](#_Toc3820724)

[Segregation of a CRUD interface 42](#_Toc3820725)

[Interface segregation and the Decorator pattern 45](#_Toc3820726)

[Inheritance vs Composition 46](#_Toc3820727)

[Exercises 49](#_Toc3820728)

[SOLID.1 49](#_Toc3820729)

[SOLID.2 51](#_Toc3820730)

[SOLID.3 52](#_Toc3820731)

[SOLID.4 53](#_Toc3820732)

# Introduction

We have come quite a long way since the beginning of these notes. We now have a rich palette of language constructions and programming techniques available, and can create rather sophisticated applications, both with regards to user interfaces and un­der­lying logic and data structures.

The next step is to consider how to put it all together. It may seem strange to start to discuss this topic at such a late stage; wouldn’t it make more sense to start with such a discussion, so we have a sense of direction for our software development from the outset? There is definitely merit for this point of view, but being able to have such a discussion does require some know­ledge about several programming techniques, so we are in a bit of a chicken-and-egg situation. Also, the scenarios for which we deve­lop software have to reach a certain level of complexity, before it really makes sense to embark on such high-level discussions.

In any case, we will in this chapter try to zoom out from all the specific details of how to develop software, and discuss some very general principles for development of high-quality software. Actually, using the term “high-quality” about software is an excellent starting point for such a discussion. What does “high-quality” even mean in the context of software development? The somewhat unsatisfying answer is proba­bly “it depends…”. It depends on the specific scenario, but also on the beliefs and attitudes of those developing the software. The question we are ultimately trying to answer is:

**What goals do we strive at, when we develop software?**

The answer “it depends…” will obviously also apply here. What could these goals more specifically be? If we are developing soft­ware for a nuclear power plant control system, we will probably “lack of errors” as a very important goal, compared e.g. to ease of use. Sure, the software should also be easy to use, but if we have to spend a bit more money on training the operators in how to use the software, as compared with the potential consequences of releasing the software with (too) many errors in it, we will probably rather spend resources on e.g. testing rather than polishing the GUI. Other scenarios may have very different primary goals; if some company needs to get their product to market as quickly as possible – perhaps because other compa­nies are also pushing a similar pro­duct – it might be much more important to get a MVP (Minimal Viable Product) on the market fast, rather than waiting for Quality Assu­rance to dissect the product for several months. Many of the globally largest soft­ware producers have – more or less explicitly – adopted an “always in beta” approach to software life­cycles, meaning that more traditional goals as “error-free” and “implements all requirements” are no longer at the top of the list.

If that is indeed the state of affairs, does it then even make sense to talk about “best practi­ces”? Yes, it still does. Even though the traditional view on how to manage the lifecycle of a software product may have changed, there are still some properties about the soft­ware which are desira­ble to strive for.

One such property is **reusability**. This has always been a “mantra” in Object­-Oriented software development, and for good reasons. If we can develop truly re­usable classes – or even systems of classes – then it will be easier to develop other systems needing such classes as well. Why does this make it “easier”? If the class is indeed reusable as-is, then we need not spend any resources on development and quality assurance. We get the entire functionality for free. This in turn will enable us to deliver a com­plete product both faster and cheaper, which are two measures of quality that often score quite high on a quality scale. The obvious question is then: How do we develop soft­ware such that it does become reusable? We will return to this very important question in a moment.

Another property is **extensibility**. It is great if we have a pool of reusable classes at our disposal, but even greater if these classes can be “extended” in certain ways, in a fashion that is unlikely to “break” the existing classes. This will enable us to use these classes in scenarios they were not explicitly developed for, which makes them very versatile. Again, we need to ask: How do we develop soft­ware such that it does beco­me extensible?

These questions are obviously questions that have been given a lot of thought in the software development community, and many attempts to answer them have been given over the years. One way of answering such questions is to try to formulate cer­tain principles for software development, that seem to have proven themselves over time. One set of such principles are the **SOLID[[1]](#footnote-1)** principles, which are the topic for this chapter. These principles have over time obtained a seminal status in the software development community, and should thus be part of the knowledge base of every aspiring software developer, but again; these principes should not be perceived as “commandments” that must be obeyed. Always take the specific circumstances of a software development scenario into consideration as well.

So, what are these **SOLID** principles then? The term “SOLID” is an acronym for five such principles:

* **S**ingle Responsibility
* **O**pen/Closed
* **L**iskov Substitution
* **I**nterface Segregation
* **D**ependency Injection/Inversion

The remainder of this chapter will detail each of these principles. We will, however, start from the **D** in **SOLID**, four a couple of reasons. The first reason is clari­fica­tion. As written above, the **D** seems to stand for two principles:

* Dependency Injection
* Dependency Inversion

The reason for presenting it in this way is because there seems to be some inconsis­tency in the community about the specific meaning of the **D**. Some texts describe the **D** as meaning “dependency injection”, while other texts use “dependency inversion”. Does it matter? Is it just two words for the same idea? No, the principle of “depen­den­cy inversion” is actually a broader principle than “dependency injection”, and you can even see dependency injection as one way of achieving dependency inversion. Therefore, we will in the rest of the chapter define the **D** as **dependency inversion**. Also, this principles is perhaps the most fundamental principle of all five principles, and we will therefore focus on this principle first.

# Dependency inversion (a.k.a. Inversion of Control)

We have seen numerous example of “dependency” during our journey in software development, since it is such a fundamental idea. Once our software grows beyond the most trivial examples, we split up our code indo methods, classes, etc., and these elements will depend on each other in order to implement the required functionality. The **Dependency Inversion** principle exposes two fundamentally different ways of letting such elements depend on each other, and recommends one approach over the other. Before proceeding, we also mention that the **Dependency Inversion** prin­ciple is also very often described as the **Inversion of Control** principle, or just the **IoC** principle. We will use the term **IoC** here as well, since it also makes it easier to distin­guish from **Dependency Injection**, which is often abbreviated **DI**.

Let us consider an example right away. Suppose we are developing a system to simu­late the behavior of certain animals. Before even starting to consider the actual soft­ware needed for this, we must of course establish a model for animal behavior. We come to the conclusion that for our simulation, we can model animal behavior in a versy simple manner:

1. If the preferred food for the animal is nearby in the environment, then the animal will obtain the food.
2. Otherwise, the animal will enter into its default idle behavior.

This is of course an extremely simplified model, but the validity of the model in itself is not the issue here. The issue here is: given this model, how do we then implement it in a suitable way? The big unknown here is of course the term “suitable”…

Since we are well trained in classic object-oriented software development, we start out by defining a general interface for animals:

**public interface IAnimal**

**{**

**void Act();**

**}**

As simple as it gets… The next natural step would then be to define a base class for animals, like this:

**public abstract class Animal : IAnimal**

**{**

**public abstract void Act();**

**public bool FoodAround(string food)**

**{**

**return …;**

**}**

**}**

We don’t have that much to say about the **Act** method itself at this level, so we just declare it as **abstract**, and thus defer the actual implementation to classes deriving from **Animal**, i.e. representing specific animals. This seems to make good sense, since it must be the animal-specific classes that implement animal-specific behaviors. The **Animal** class does however seem to be a good place to define more general-purpose methods like the **FoodAround** method, such that animal-specific classes can make use of such library-like methods when implementing behaviors.

With this in place, we can create classes for specific animals, like a **Cat** class:

**public class Cat : Animal**

**{**

**public override void Act()**

**{**

**if (FoodAround("Mouse"))**

**{**

**HuntMice();**

**}**

**else**

**{**

**Sleep();**

**}**

**}**

**private void HuntMice() { }**

**private void Sleep() { }**

**}**

This class defines a how a cat behaves, and the implementation seems to be aligned with our general model for animal behavior. All is good… or is it? Yes, this implemen­tation of cat behavior complies with our general model, but who is responsible for this to happen? Well, the **Cat** class is…and isn’t that how it’s supposed to be? Well, think again about our model. We claimed that all animals should follow this model w.r.t. behavior, but we have no control w.r.t. if animal-specific classes actually imple­ment this model. It is entirely up to the implementer of the **Cat** class to ensure that the implementation complies with the general model. In more general terms: it is the derived classes that are in control of implementing the model.

## IoC by use of Template Method pattern

Leaving a derived class in control of implementing a general algorithm is problematic, in the sense that since we have indeed defined a general model for animal beha­vior, the implementation of the model should be done in the base class instead, such that the base class has control over the algorithm for behavior. But how can we do that, if some of the steps in the behavior algorithm are indeed animal-specific? This can be done by using the **Template Method** design pattern. We have discussed this pattern elsewhere, but the essence of the pattern is to define a method which outlines a ge­ ne­ral algorithm, but leaves certain steps of the algorithm open for implementation in derived classes. In our case, a revised version of the **Animal** base class could look like this:

**public abstract class Animal : IAnimal**

**{**

**public void Act()**

**{**

**if (FoodAround(PreferredFood()))**

**{**

**GetFood();**

**}**

**else**

**{**

**Idle();**

**}**

**}**

**private bool FoodAround(string food) { return …; }**

**protected abstract string PreferredFood();**

**protected abstract void GetFood();**

**protected abstract void Idle();**

**}**

This is quite different from before: now the **Animal** class clearly defines the general algorithm for animal behavior, but does so by defining a set of **abstract** methods, which must then be implemented in the derived classes. In this way, it is no longer the derived classes that are in control; it is now the base class which has control, and control has thus been “inverted” as compared to before. You could argue that you are still giving some control to the derived classes, since they can still implement the abstract methods in any way they want. That is as such true, but the derived classes can no longer redefine the general algorithm! The abstract methods are very limited in scope, and are designed to fill in very specific “gaps” in the general algorithm. A revised version of the **Cat** class could now look like this:

**public class Cat : Animal**

**{**

**protected override string PreferredFood() { return "Mouse";}**

**protected override void GetFood() { HuntMice(); }**

**protected override void Idle() { Sleep(); }**

**private void HuntMice() { … }**

**private void Sleep() { … }**

**}**

There is no longer any sign of the general algorithm in the **Cat** class, it’s only job now is to plug in the missing parts in the general algorithm. Also, the **Cat** class no longer calls any methods in the **Animal** base class. The **IoC** principle does not as such forbid that derived classes make calls to base class methods, as long as such class are to “library-like” methods. The defining idea is the transference (or “inversion”) of con­trol over the central algorithm from the derived class to the base class.

## The Dependency Injection (DI) principle

We stated above that the principle known as **Dependency Injection** (often just **DI**) is one way of implementing the **IoC** principle, and that is indeed true. Formu­la­ting it this way may sound like we are dismissing **DI** as a small detail, which only pur­pose is to serve as an implementation strategy for **IoC**. This is far from true! **DI** is in its own right a very important principle, and we will therefore give it a rather thorough treat­ment, before using it in an **IoC** context. We will even discuss **DI** in a broader per­spec­tive than what is most commonly done (many discussions only perceive **DI** as a prin­ciple for dependencies between classes and objects), to illustrate that this prin­ciple comples into play at almost all level of complexity. In a sense, we have been using this principle for quite some time ☺.

The term **Dependency Injection** seems to suggest that **someone** will **inject** **some­thing** into **something else**, as illustrated below:



The term also suggests that this **something else** did not contain this **something** to begin with. If it did, it would then be inherently “locked” in a tightly coupled relation­ship with this **something**, which is what we want to avoid. Replacing the terms used above with the terms commonly used when discussing Dependency Injection, we get this illustration:



So, an **Injector** will inject a **Service** into a **Client**, and the client can then make use of this service. But how is this different than just having a hard-coded relation between the service and the client? Once we have performed the injection, the dependency will be the same anyway, yes? Well, not quite. The difference lies in the knowledge the cli­ent will have about the specific nature of the service.

### Dependency Injection – parameter level

Let’s illustrate this with an example. As mentioned previously, this discussion will perceive **DI** in a broader context than usual, so the definition of the term “service” will be stretched a bit. Please play along…

Consider the below – quite clumsy – attempt to define a method for calculating the square of 4:

**public int SquareOf4()**

**{**

**return 4 \* 4;**

**}**

The method does what it advertises; it calculates the square of the number 4. If some­­one later on asks us for a method which can calculate the square of 6, we can quickly deliver:

**public int SquareOf6()**

**{**

**return 6 \* 6;**

**}**

At this point in your software development training, you should hopefully be able to come up with a much better solution to this problem:

**public int Square(int n)**

**{**

**return n \* n;**

**}**

Of course! This is much better than just mindlessly copying the original code, and re­placing 4 with whatever new value someone requests. This solves the problem once and for all. But what did we more specifically do? Something like this:



In the “Before” scenario, we had hard-coded a tight relationship between the met­hod **Square** (the Client) and the number **4** (the Service – yes, a big word for a num­ber…). In the “After” scenario, we removed the explicit dependency between **Square** and **4**, and turned the number into something the Injector – i.e. the caller of **Square** – must “inject” into the method, by providing it as a parameter to **Square**. Why was it possible to do this? The only operation happening inside **Square** is multiplication, and – and this is a very important point – multiplication works in the same way, no matter the specific number we supply! We can therefore loosen up the requirements to that “thing” we do multiplication on. Can it be anything at all, then? No, since multiplica­tion only makes sense on numerical values. It will not make sense to perform multi­pli­cation on a **string** value. So, any numerical value, then? In principle, yes. However, the return type of **Square** has been defined to be **int**, so requiring that the parameter must also be of type **int** seems to be just the right amount of restriction we need to put on the parameter. In this case, the analysis was very simple. Still, it is a valid illu­stration of perceiving parameterisation as a variant of the **DI** principle.

Let’s see a slightly less obvious example. The below code defines a class **Die**, which is intended to model a standard 6-sided die:

**public class Die**

**{**

**private int \_faceValue;**

**private static Random \_random = new Random();**

**public Die()**

**{**

**Roll();**

**}**

**public int FaceValue**

**{**

**get { return \_faceValue; }**

**}**

**public void Roll()**

**{**

**\_faceValue = \_random.Next(0,6) + 1;**

**}**

**}**

Rolling the die – by calling the **Roll** method – sets the face value to a random num­ber between 1 and 6 (the call of **Next**(0,6) will generate a number between 0 and 5, so we need the +1 after the call). So, what exactly makes this a model of a 6-sided die? That is of course the specific number 6, used when calling **\_random.Next**. If we now wanted a class for modeling a 10-sided die, we could just copy-paste the entire class, and replace 6 with 10, yes? No, let’s not go down that road again… Instead, we will solve the problem by applying the **DI** principle again. But where? Should we turn the hard-coded number into a parameter to **Roll**, like this?:

**public void Roll(int noOfSides)**

**{**

**\_faceValue = \_random.Next(0, noOfSides) + 1;**

**}**

It’s a possible solution, but sort of breaks the idea that a **Die** object models a real-world die. A real die doesn’t change its number of sides between calls, so what then? A better solution would be to specify the number of sides when a **Die** object is con­struc­­ted, by adding a parameter to the **Die** class constructor, plus an instance field to the class:

**public class Die**

**{**

**private int \_faceValue;**

**private int \_noOfSides;**

**private static Random \_random = new Random();**

**public Die(int noOfSides)**

**{**

**\_noOfSides = noOfSides;**

**Roll();**

**}**

**public int FaceValue**

**{**

**get { return \_faceValue; }**

**}**

**public void Roll()**

**{**

**\_faceValue = \_random.Next(0, \_noOfSides) + 1;**

**}**

**}**

The revised class can now be a model for a die with any number of sides; the specific number of sides for a specific object is specified at object creation time, by the object creator, i.e. the Injector. Again, this rests on the observation that die operations are so general that they do not depend on the specific number of sides, as long as that number is an integer number (how does a die with 3.7 sides behave…?). A more com­plete implementation should probably also check that **noOfSides** is at least 2, and per­­haps throw an exception if this is not the case.

### Dependency Injection – method level

We have now (hopefully) gained an understanding of the **DI** prin­ciple, so let’s see it applied at a method level. Applying the principle here is conside­rably harder, and requires use of some of the more advanced language constructions in C#. Consider the problem of filtering out certain values from a list of given values, accor­ding to some condition. An example is given below:

**public List<int> FilterValues(List<int> values)**

**{**

**List<int> filteredValues = new List<int>();**

**foreach (var value in values)**

**{**

**if (value > 10)**

**{**

**filteredValues.Add(value);**

**}**

**}**

**return filteredValues;**

**}**

In this example, we filter out those numbers which are larger than 10, and add them to the resulting list **filteredValues**, which is returned to the caller. What if we wish to filter according to a different condition? The method for doing this would actually be identical to the example, except for the condition specified in the **if**-statement (as high­­lighted in the code). The condition itself is then a perfect candidate for being turned into a parameter. But how? Think about what characterises the condition. It can be per­ceived as a function, taking one integer value as input, and returning a boolean value. We have learned earlier that such a function characterisation can be expressed as a C# type, in this case the function type **Func<int,bool>**. That type iden­ti­fies exactly those func­tions we just described. We can then rewrite the above to:

**public List<int> FilterValues(List<int> values, Func<int,bool> condition)**

**{**

**List<int> filteredValues = new List<int>();**

**foreach (var value in values)**

**{**

**if (condition(value))**

**{**

**filteredValues.Add(value);**

**}**

**}**

**return filteredValues;**

**}**

A caller can then call **filterValues** like this:

**List<int> filteredValues = FilterValues(someValues, v => v > 10);**

The function parameter is here expressed as a lambda expression, but it could also be a named function, as long at it conforms to the type specification. This is a more elaborate example of **DI** through parameterisation, but the rea­soning behind it is actually quite similar to the next-to-trivial examples we saw ear­lier. We recognised that part of the logic inside the method did not need to be tightly coupled with a specific “value” (here a selection condition, which can be expres­­sed as a function of a specific type), so we parameterised that value, thereby making it possible for the caller to inject a specific value – here a specific selection criterion – when the method is called.

### Dependency Injection – object level

In order to see an example of **DI** at the object level, we could elaborate our previous example of modeling animal behavior. The animals we are modeling should have some sort of “world” to exist in. Again, we can go ahead and create a interface and perhaps a couple of implementations:

**public interface IWorld**

**{**

**bool IsAnimalClose(string animalDesc);**

**}**

**public class WorldFewAnimals : IWorld**

**{**

**public bool IsAnimalClose(string animalDesc)**

**{**

**// Low probability**

**return …;**

**}**

**}**

**public class WorldManyAnimals : IWorld**

**{**

**public bool IsAnimalClose(string animalDesc)**

**{**

**// High probability**

**return …;**

**}**

**}**

The implementation details of these **World…** classes are not important here, but it is important how the dependency between animal objects and world objects is establi­shed. One strategy could be to provide a parameter to the **Animal** constructor, which sould indicate if the animal will live in a world with many or few animals (only parts of the class is shown below):

**public abstract class Animal : IAnimal**

**{**

**private IWorld TheWorld { get; }**

**protected Animal(bool manyOrFew)**

**{**

**if (manyOrFew)**

**{**

**TheWorld = new WorldManyAnimals();**

**}**

**else**

**{**

**TheWorld = new WorldFewAnimals();**

**}**

**}**

**}**

It may look reasonable at first, but this approach establishes a tight coupling between the **Animal** class and specific implementations of the **IWorld** interface. What if we at some point create a revised implementation of one of these classes, say **WorldMany­AnimalsV2**? We will then have to open up the **Animal** class and change its implemen­tation, with all the effort needed for retesting, release, etc. being imposed on us. The solution is simply to apply the same principles as before, i.,e. make it possible for the cre­ator of an **Animal** object to specify which specific **IWorld** implementation it should depend on:

**public abstract class Animal : IAnimal**

**{**

**private IWorld TheWorld { get; }**

**protected Animal(IWorld theWorld)**

**{**

**TheWorld = theWorld;**

**}**

**}**

This is an archetypical example of **DI**: rely on interfaces instead of specific imple­men­tations, and allow an external agent to inject dependencies:

**IWorld aWorld = new WorldManyAnimals();**

**Cat aCat = new Cat(aWorld);**

Sure, if we need to use a new implementation of **WorldManyAnimals**, we need to change this code. But maybe this code is part of a class with the specific responsibility of setting up such dependencies, meaning we have a much better division of respon­sibilities between classes.

Fur­thermore, this approach makes testing of classes much more flexible. Whenever an ob­ject only depends on other objects known by interface type – and these objects are inject­ed into the object – it will always be possible to use “surrogate” objects, as long as these objects also implement the interface in question. Instead of having to use an implementation which e.g. connects to a remote database, a surrogate object can be used instead, which is likely to make the test both faster and more reliable.

## IoC by use of Dependency Injection

We have now hopefully established clearly what **DI** is all about. In fact, our example above is an excellent example of how to use **DI** as a way of achieving **IoC** at object level. How exactly has “control” been “inverted” here? Consider again the first imple­mentation of the **Animal** class, where the **Animal** class explicitly created new objects of **IWorld** implementations (we show only the constructor here):

**protected Animal(bool manyOrFew)**

**{**

**if (manyOrFew)**

**{**

**TheWorld = new WorldManyAnimals();**

**}**

**else**

**{**

**TheWorld = new WorldFewAnimals();**

**}**

**}**

The **Animal** class is here in “control”, in the sense that it alone decides which specific objects it wants to depend on! The outside world can influece which of the two speci­fic implementations it chooses, but it cannot force a dependency of any other imple­mentation. The second implementation turns this around:

**protected Animal(IWorld theWorld)**

**{**

**TheWorld = theWorld;**

**}**

Now the **Animal** class has no control over which object it will depend on; this is now up to the outside world, i.e. the agent which creates an **Animal** object, and injects a specific **IWorld** implementation into it. Between these two parties, control has now been inverted.

# The Single Responsibility Principle

After this rather significant effort spent on discussing just one of the five principles, you may wonder if all five principles require just as much effort to explain and under­stand. Fortunately not ☺. After having started from the back-end of **SOLID**, we will now have a look at the remaining four pinciples in order, starting with the **Single Responsibility** principle.

This principle builds on a very simple and pragmatic idea related to organisation of code in general:

* Code that changes for the same reasons should be grouped together
* Code that changes for different reasons should be separated

We have used this principle many times already, for instance in the argumentation for the MVVM archetecture. Code that e.g. defines business rules should somehow be separated from code that defines how something looks in the UI. The arguments are also fairly well-known; if we bunch all this code into e.g. a single class, that class will have to be changed very often. If the business rules change, the class must be changed. If the UI logic changes, the class also has to be changed! So what…? Why is it a problem to change a class frequently? Often a change in a class will trigger a lot of additional activities, such a testing, integration, release, etc.. These activities use up resources, i.e. cost money! Also, if a lot of (logically) unrelated code is bunched up in the same class, the risk of unintentionally introducing errors also increases, simply because it is harder to maintain an overview of the internal dependencies.

Consider for instance our animal behavior example from earlier. We said at some point that the **Animal** base class should contain the general algorithm for animal behavior, but also that it could make sense to include more library-like methods in the base class as well. Let’s pursue that idea a bit: in the **Animal** class, we had an **Act** method which defined the general algorithm for animal behavior. It also contained a number of abstract methods, used when implementing the **Act** method. But it also cointained an innocently looking method **FoodAround**:

**private bool FoodAround(string food)**

**{**

**return …;**

**}**

How would such a method actually be implemented? An animal object was also sup­posed to have a reference to some sort of “world” object, more specifically an object implementing the interface **IWorld**. So maybe the implementation would look some­thing like this:

**private bool FoodAround(string food)**

**{**

**return TheWorld.IsAnimalClose(food);**

**}**

So we’re implementing this method by calling a method in the **IWorld** interface. Is that a problem? Well, suppose now that somebody figures out that **IsAnimalClose** should take an additional **distance** parameter, which defines what we more specifi­cally mean by “close”. We may then need to change the **FoodAround** method:

**private bool FoodAround(string food)**

**{**

**return TheWorld.IsAnimalClose(food, 200);**

**}**

A very simple change, but a change nonetheless, and we may now need to test the **Animal** class, release it, etc.. But if this is really a change that is needed, then surely we cannot avoid this? True, but if we take a moment to reconsider what the respon­si­bilities for the **Animal** class really are, we can maybe reduce the impact of such a change. What are the responsibilities for the **Animal** class now? The class now:

* Defines the general algorithm for animal behavior (the **Act** method)
* Has a library of useful methods, which can be called by the **Animal** class itself, but perhaps also by the derived classes

So, two responsibilities… Could it make sense to separate them? The change we just described above has nothing to do w.r.t. the first responsibility. Sure, it may change the behavior of the code in the **Act** method, but does not require any change to the code as such in the **Act** method. Still, all systems that depend on the **Animal** class may be affected by the need to change the class, even if the change concerns func­tionality that is not being used by the system!

The solution? Split up the **Animal** class into two new classes with more focused responsibilities: An **AnimalBehavior** class (and probably also an **IAnimalBehavior** interface), and likewise an **AnimalLibrary** class (and interface).

**public interface IAnimalBehavior**

**{**

**void Act();**

**}**

**public interface IAnimalLibrary**

**{**

**bool FoodAround(string food);**

**void Sleep();**

**}**

**public abstract class AnimalBehavior : IAnimalBehavior**

**{**

**protected IAnimalLibrary AnimalLib { get; }**

**protected AnimalBehavior(IAnimalLibrary anAnimalLib)**

**{**

**AnimalLib = anAnimalLib;**

**}**

**public void Act()**

**{**

**if (AnimalLib.FoodAround(PreferredFood()))**

**{**

**GetFood();**

**}**

**else**

**{**

**Idle();**

**}**

**}**

**// Abstract methods omitted**

**}**

**public class AnimalLibrary : IAnimalLibrary**

**{**

**public bool FoodAround(string food)**

**{**

**// General code for FoodAround...**

**}**

**public void Sleep()**

**{**

**// General code for sleeping...**

**}**

**}**

With all this in place, we can also revise our implementation of the **Cat** class.

**public class Cat : AnimalBehavior**

**{**

**public Cat(IAnimalLibrary anAnimalLib) : base(anAnimalLib)**

**{**

**}**

**protected override string PreferredFood()**

**{**

**return "Mouse";**

**}**

**protected override void GetFood()**

**{**

**HuntMice();**

**}**

**protected override void Idle()**

**{**

**AnimalLib.Sleep();**

**}**

**private void HuntMice() { … }**

**}**

With these changes, who will now be affected by a change in **AnimalBehavior**? Only those classes with a direct dependency to this class. Any class which only makes use of the library part will now be completely unaffected. The **Cat** class will still cleary be affected by a change to **AnimalBehavior**, since it inherits from this class – there is no way around this, as long as we rely on inheritance. But what about a change to **Ani­malLibrary**? This will not affect the **Cat** class, since it does not make direct use of this class. The **Cat** class – and also the **AnimalBehavior** base class – are only depend­ent on library implementations through the **IAnimalLibrary** interface! And how is that dependency established? By **Dependency Injection**!

The idea of decomposing large classes into smaller classes, and letting said smaller classes establish dependencies through interfaces and **DI**, is the very essence of the **Single Responsibility** principles. A rather tough question remains, though: Is there any lower limit for how small classes ought to be? When should we stop decompo­sing? Setting a general rule for this is impossible… However, one approach could be to investigate exactly how the clients to a particular class use the functionality of the class. If two methods have a different usage pattern, they should probably go into separate interfaces (and probably separate classes as well). Iterating over this pro­cess will at some point result in a set of interfaces where all methods follow the same usage pattern, which should indicate that you can now stop decomposing.

# The Open/Closed Principle

The Open/Closed principle comes in several variations with regards to formulation, but the essense of the principle is:

Software entities should be **open** for extension, but **closed** for modification.

The first time you see this principle, it seems to be almost paradoxical. How can something be open for something, and also closed for something that sounds like almost the same thing? This requires a bit of clarification.

The first clarification concerns the term “software entities”. What is that exactly? In the realm of Object-Oriented software development, we can usually substitute this with “a set of classes and interfaces, designed for one common purpose”. With that out of the way, we can address the “closed” part. This sounds very rigid. Must we then deve­lop perfect software in the first try? That is of course unrealistic. A less draconic version of the “closed” part could be “closed for modification that requires clients of the code to change”, clients being defined as any other part of the software making use of the class in question.

Let’s illustrate these concepts with an example from a different realm than animal behavior simulation: Suppose that a class **Client** makes use of a class **CalculatorV10**, which in turn uses a class **Data** for providing data for some sort of calculation. Code for the **Client** class could look like this:

**public class Client**

**{**

**public CalculatorV10 \_calculator;**

**public Client()**

**{**

**\_calculator = new CalculatorV10();**

**}**

**public void ProcessData(Data d)**

**{**

**int result = \_calculator.Calculate(d);**

**//...**

**}**

**}**

Suppose now that the developers of **CalculatorV10** find a critical error in the class, to an extent where a new version of the class must be released. How would such a bug-fix version be released? As a new release of a class called **CalculatorV10**? Or maybe as an entirely new class called **CalculatorV11**? In the first case, the new release will not require the client code to change, even though it might be required to rebuild the enti­re system, but that is more of a technicality. In the second – and probably more likely – scenario, the client code will indeed need to be updated. Is that a problem? Many companies have a policy of always testing updated versions of their code, so if a part of the client code has been updated, it might be necessary to spend resources on re-testing the system. This comes on top of the cost of actually updating the code. In short, this code is not really designed with the open/closed principle in mind.

How can we change this? In the above example, we have not stated anything about the relation between the classes **CalculatorV10** and **CalculatorV11**. They both offer some sort of (probably identical) calculation functionality, but how are they related at a class level? The worst scenario would be that they have no relation. They might be implementing almost the same functionality, and may even have a method with the same name and the same parameter list, but if they have no relation in terms of inhe­ritance or interfaces, it becomes impossible to migrate from one vesion to the other, with­out having to explicitly modify the client code.

A better – but not perfect – strategy is to relate the classes by inheritance. Suppose the **CalculatorV10** class is implemented like this:

**public class CalculatorV10**

**{**

**public virtual int Calculate(Data d)**

**{**

**int result = 0;**

**// Code for calculating a result…**

**return result;**

**}**

**}**

Note that the method **Calculate** has been defined as being **virtual**, indicating that the method can be overrided in a derived class. The class **CalculatorV11** could then be im­plemented in this way:

**public class CalculatorV11 : CalculatorV10**

**{**

**public override int Calculate(Data d)**

**{**

**int result = 0;**

**// Bug-fixed code...**

**return result;**

**}**

**}**

Does this help us with regards to keeping the the client code unchanged? To some extent, but not completely. Let’s look at the code again:

**public class Client**

**{**

**public CalculatorV10 \_calculator;**

**public Client()**

**{**

**\_calculator = new CalculatorV10();**

**}**

**public void ProcessData(Data d)**

**{**

**int result = \_calculator.Calculate(d);**

**//...**

**}**

**}**

Two places in the code are particularly interesting (highlighted above). The first is the declaration of the instance field **\_calculator**. What objects can this instance field refer to? Obviously to objects of type **CalculatorV10**… but also objects of type **Calcula­tor­V11**, due to the inheritane relationship. This is clearly an improvement. However, the second piece of highlighted code is still problematic, since we here explicitly cre­ate an ob­ject of type **CalculatorV10**. This is an extremely central problem in software design; whenever we need to create new objects, we must state their specific type explicitly. The entire category of design patterns known as **creational patterns** are in fact varia­tions over how to “encapsulate” the object creation process.

Are we then stuck here? If we insist on creating a calculation object explicitly in the **Client** class, we cannot really do much more. However, if we assume that a calcula­tion object can be created outside the **Client** class, and somehow be made available to the methods in the **Client** class, we suddenly have more options. Consider the below ver­sion of the **Client** class:

**public class Client**

**{**

**public CalculatorV10 \_calculator;**

**public Client(CalculatorV10 calculator)**

**{**

**\_calculator = calculator;**

**}**

**public void ProcessData(Data d)**

**{**

**int result = \_calculator.Calculate(d);**

**}**

**}**

The **Client** class will now no longer create a calculation object, but rather receive a calculation object through the constructor parameter (doesn’t that sound very fami­liar, considering the previous chapter on **IoC** and **DI**…?). What type can this object have? Again, both **CalculatorV10** and **Calcula­torV11**, due to inheritance. In this way, the **Client** class is unaware of the specific type of calculation object; it just knows that it is – or inherits from – **CalculatorV10**. We could then release further versions of the calculator class, and make use of them in the **Client** class without any code changes, as long as they inherit from **CalculatorV10**. This idea can be taken a bit further by bringing **interfaces** into play. Suppose an inter­face for calculation has been defined:

**public interface ICalculator**

**{**

**int Calculate(Data d);**

**}**

We now assume that all calculation classes will implement this interface. We can then update the **Client** class further:

**public class Client**

**{**

**public ICalculator \_calculator;**

**public Client(ICalculator calculator)**

**{**

**\_calculator = calculator;**

**}**

**public void ProcessData(Data d)**

**{**

**int result = \_calculator.Calculate(d);**

**}**

**}**

As long as all future calculator classes implement this interface, we can make use of them in the **Client** class, without the need for updating the code. This is a quite typi­cal example of how to design with the open/closed principle in mind; the **Client** class is “open” to make use of future calculation classes – without detailed knowledge on how they specifically work – but also “closed” in the sense that there is no need to chan­ge it, just in order to use it with new calculation classes.

One thing has been swept a bit under the rug, though. Somewhere in the client code – but outside the **Client** class – somebody needs to create a calculation object, and pro­vide it to a **Client** object, like this:

**ICalculator aCalculator = new CalculatorV11();**

**Client aClient = new Client(aCalculator);**

We are thus not completely out of the woods yet. Somebody does have to choose a specific implementation of **ICalculate**, and create an object of that type. However, mo­ving that responsibility out of the **Client** class opens up for a lot more flexibility with regards to where to create that object. As we have discussed a couple of times already, we could place this responsibility in a class which has such “injections” of dependencies at its primary area of responsibility.

# The Liskov Substitution Principle

The discussions on **Single Responsibility** and **Open/Close** Principles have hopefully demonstrated that the **SOLID** principles are not a set of very academic and cryptic principles; they are actually pretty straightforward and even intuitive. The **Liskov Substitution** principle, however, has a slightly different nature. The general principle is not as such very difficult to understand – even though it involves a bit more forma­lism than the other principles – but the consequences of the principles can seem sur­prising, and may even make you question if inheritance is useful at all! Don’t worry, inheritance is still useful, but the principle shows that we need to think care­fully about how we use inheritance.

## Definition

The principle is named after computer scientist Barbara Liskov, and it goes like this:

*If the class* ***S*** *is a subtype of the class* ***T****, then objects of type* ***T*** *may be replaced with objects of type* ***S****, without breaking the program.*

So, the principle is definitely about inheritance. **S** inherits from **T**, and objects of the classes **S** and **T** are then used for something. More specifically, we could imagine a **Client** class that uses objects of type **T** for something:

**public class Client**

**{**

**public void DoSomething(T obj)**

**{**

**// Code that uses obj of type T**

**}**

**}**

Suppose now that the class **T** implements a very simple method **SayHello**:

**public class T**

**{**

**public virtual void SayHello(string name)**

**{**

**Console.WriteLine($"Hello {name}");**

**}**

**}**

We can now make a more useful implementation of the **DoSomething** method in the **Client** class:

**public class Client**

**{**

**public void DoSomething(T obj)**

**{**

**obj.SayHello("Alex");**

**obj.SayHello("Betty");**

**}**

**}**

If we invoke **DoSomething** with an object of type **T**, it will simply print out a couple of greetings on the screen, just as we intended. We can then proceed to implement a class **S** which inherits from **T**:

**public class S : T**

**{**

**public override void SayHello(string name)**

**{**

**Console.WriteLine($"Hola {name}");**

**}**

**}**

If we now invoke **DoSomething** with an object of type **S** – which is indeed possible since **S** inherits from **T** – it prints out a spanish-flavored greeting. This is actually very useful, since we have now made **DoSomething** do something new, even though we did not change the code of **DoSomething** itself. This is the **Open/Close** principle in action! Very nice, but… maybe the code does something now that “breaks” the client code, in the sense that it behaves in a way that the client did not intend. But surely it doesn’t “break” the client code that we now print a spanish-flavored greeting… or does it? What are the precise expectations that the client has to the incoming object?

## Intentions and Contracts

In order to discuss this rather tough question, let’s introduce an interface first. This is just the **DoSomething** method lifted into an interface **IT**:

**public interface IT**

**{**

**void SayHello(string name);**

**}**

If you just look at this interface *as-is*, what would be your expectations about what any implementation of the interface will do? We would probably read some meaning into the fact that the method is named **SayHello**, and also the fact that it takes one parameter named **name** of type **string**, and therefore probably think that invoking the method will print out some form of greeting on the screen, which will somehow involve the value of the argument provided to the method. That is, however, a very loose formulation, involving phrases like “some form of” and “somehow”. Would it be acceptable if the method just printed *“Hello, you…”* on the screen, completely ignoring the argument? Would it be acceptable if the greeting is in a different langu­age than english? The point is; just creating a interface is usually not enough to esta­blish the intention of the interface. In order to establish such an intention, we need a more thorough description of the intention. Such a description is often – in the con­text of software development – referred to as a **contract**.

The term “contract” can have a more formal meaning when defining interfaces, but we will use it in a more informal way here, simply as a term for all that extra informa­tion we need to provide when creating an interface. A first attempt at establishing such a contract could simply be to include comments in the interface code:

**public interface IT**

**{**

**/// <summary>**

**/// Contract: Invoking this method should print a message on the screen.**

**/// The message should**

**/// 1) Have a polite greeting nature.**

**/// 2) Use the name provided in the argument.**

**/// 3) Be in English.**

**/// No side effect should occur by calling this method.**

**/// </summary>**

**void SayHello(string name);**

**}**

We are not in any way claiming that this is an objective and bullet-proof contract, and it also uses subjective phrases such as “polite”. Still, it could be useful simply by clarifying – to some extent, at least – the intention of the interface to anyone who needs to implement the interface.

Detecting if a contract like the above is being broken will however be quite difficult, and will probably need to involve human judgement. Suppose now we make a more “relaxed” version of the contract:

**public interface IT**

**{**

**/// <summary>**

**/// Contract: Invoking this method should print a message on the screen,**

**/// no matter the value of name.**

**/// </summary>**

**void SayHello(string name);**

**}**

If this is the contract, we are definitely on the safe side with our spanish-flavored implementation. This contract will allow for some more or less pointless implemen­tations as well, but it will at least be much easier to verify if an implementation obeys the contract or not.

How can we now break the contract? Consider the below implementation:

**public class CheckedGreeting : IT**

**{**

**public override void SayHello(string name)**

**{**

**if (name.Length < 3)**

**{**

**throw new ArgumentException("Name too short!");**

**}**

**base.SayHello(name);**

**}**

**}**

This is definitely a contract-breaking implementation, since we can now risk that the client code will throw an exception some place in the code that previously worked just fine! If we wish to obey the Liskov Substitution principle, we cannot accept this implementation. More specifically, we are breaking one of three very fundamental rules in the Liskov Substitution principle, relating to subtypes.

## Preconditions

According to the Liskov Substitution principle, it must always hold for subtypes **S** to a type **T** that:

* **Preconditions** in **T** are never strengthened by **S**.
* **Postconditions** in **T** are never weakened by **S**.
* **Invariants** in **T** must be preserved by **S**.

What is a “precondition”? In our example, a precondition could be that certain con­di­tions must apply to the **name** parameter, in order to consider it to have a valid value. In the base class **T**, we did not have any such precondition – or the precondition was just that the provided argument must be of type **string**, which is ensured by the type-checking system – since we just print whatever value the caller gives us. In the sub­class **CheckedGreeting**, we now strengthen that precondi­tion, which violates the rule on preconditions.

We will talk a bit further on postconditions and invariants as well, but this is a good place to stop and consider our position. With this small example, we may begin to question our perception of what inheritance really means.

## What is inheritance actually?

First of all; is the example above a realistic example? I think it is. It seems like a rea­sonable line of reasoning to think that *“I need a general base class for producing greetings, and then I can later on implement more specialised versions of greeting-producers, which may e.g. impose certain restrictions on the provided* ***name*** *argu­ment”*. Also, we are probably doing this reasoning because we a have a real need on our hands. We probably do need to implement this “checked” version of a greeting-producer, to meet some requirement for our application. The problem lies in this particular step of the reasoning:

1. If class **Y** implements a more restricted version of the functionality in class **X**…
2. …then class **Y** must be a subtype of class **X**

Why is this wrong? Consider what is means to be a “subtype”. If we think in terms of contracts and interfaces, we could start our by stating the following:

* All implementations of an interface must obey the interface contract.

In our specific example, we have defined a contract for the interface **IT**, and have also created an implementation **T** which obeys that contract. That natural consequence of this is that:

* All objects of type **T** obey the contract for interface **IT**

We have also created two classes **S** and **CheckedGreeting**, which both seem like they are candidates for being subtypes to **T**. But again; what does it really mean to be a “subtype”? We usually say that if **Y** inherits from **X**, then **Y** *is-a* **X**, i.e. the classes have an *is-a* relationship. But when is that true? We could also look at the relation from a set-oriented perspective: if **X** is the set of implementations that obey the contract for an interface, then it follows that if **Y** *is-a* **X**, then all implementations of **Y** must also obey the interface contract, since all these implementation will also be part of the set of implementations for **X**. This is precisely the definition of being a “subtype”!

Relating this to our example, we have:

1. All classes inheriting from class **T** must obey the contract for interface **IT**
2. Class **S** implements a specialised version of the functionality – it returns a spanish-flavored greeting – but it does obey the contract, since it prints a message on the screen, no matter the value of **name**.
3. Class **CheckedGreeting** implements a specialised version of the functionality – it returns a greeting for names longer than two characters – but it does not obey the con­tract, since it may throw an exception in some cases.

We are now at the very essence of the principle; a derived class **S** may definitely implement a more specialised version of the functionality in the base class **T**, but it must also obey all contracts that **T** is obeying! This is exactly what separates the two subclasses in our example:

|  |  |  |
| --- | --- | --- |
|  | *Specialises functionality* | *Obeys interface contract* |
| **S** | Yes | Yes |
| **CheckedGreeting** | Yes | No |

In other words: be careful what you use inheritance for…

## Dealing with non-compliance

From the above, we can finally come to the conclusion that we cannot let **Checked­Greeting** inherit from **T**, if we want to comply with the Liskov Substitution principle. But then what? If we really need the functionality implemented in **Checked­Greeting**, how should we then organise our code? We have two ways forward, which are not mutually exclusive:

* Change the contracts.
* Change the implementations.

The first approach may seem like the easy way out, and you shoud of course not just change a contract just to “fix” a problem with inheritance. Still, if we again consider our example, there must be a reason that we suddenly need this “checked” version of the functionality. Maybe our very “relaxed” version of the contract was not quite good enough. Consider this revised version of the contract:

**public interface IT**

**{**

**/// <summary>**

**/// Contract: Invoking this method with a name longer than**

**/// two characters should print a message on the screen.**

**/// </summary>**

**void SayHello(string name);**

**}**

With this contract, we could actually switch the roles of **CheckedGreeting** and **T**, such that **CheckedGreeting** is the base class and **T** is the subclass. How so? The implemen­tation in **CheckedGreeting** now obeys the contract, in the sense that it does produce a greeting for names longer than two characters. Sure, it still throws an exception for short names, but that’s not a functionality covered by the contract! We have a sort of “anything goes” situation w.r.t. how to act for short names; the base class throws an exception, but a derived class could change this behavior, e.g. by also being able to print a greeting for short names as well! But is that not breaking the contract? Well, the contract does not say anything about how an implementation should act in case of short names… This could also indicate that the contract itself is flawed, but as the contract stands, anything goes.

What about changing the implementation? If we revert to the original contract, there is not any way in which we can “bend” **CheckedGreeting** into being part of the class hierarchy defined by **IT** and **T**. One way forward could be to create a new interface **ICheckedGreeting** which models the idea of a “checked” greeting generator:

**public interface ICheckedGreeting**

**{**

**/// <summary>**

**/// Contract: Invoking this method with names longer than two**

**/// characters should print a message on the screen.**

**/// In case of being invoked with a name at most two characters**

**/// long, the method may throw an ArgumentException**

**/// </summary>**

**void SayHello(string name);**

**}**

If the **CheckedGreeting** class now implements this interface, it will obey its contract. The consequence is that we can no longer use **CheckedGreeting** as an argument for any methods having a parameter of type **IT**:

**public class Client**

**{**

**public void Run()**

**{**

**DoSomething(new S()); // OK**

**DoSomething(new CheckedGreeting()); // ERROR**

**}**

**public void DoSomething(IT obj)**

**{**

**// All these cases should work…**

**obj.SayHello("Alex");**

**obj.SayHello("Betty");**

**obj.SayHello("Bo");**

**}**

**}**

This might be surprising, since the interfaces **IT** and **ICheckedGreeting** look identical; they both contain a single method **SayHello**, with exactly the same signature. How­ever, that does not make them identical in the eyes of the type checking system. This is a good thing, since they are indeed different with regards to intent. The contracts for the interfaces are different, and they should therefore be treated as two logically different entities. With this change, we can never be in a situation where the client makes the call **SayHello(“Bo”)** and expects this to print out a message, but instead encounters an exception.

Another consequence of introducing **ICheckedGreeting** will probably be that some­where in our application, we need to change some code that now uses the **IT** inter­face to use **ICheckedGreeting** instead, i.e. the place where we do wish to have some sort of check when producing greetings. But doing this will just bring our client code in better compliance with the requirements. After all, the contracts must be rooted in requirements to the application, so we should ensure that contracts and actual implementations match as closely as possible, e.g. by using the appropriate interfa­ces as types for method parameters.

Suppose, however, that we do insist on having a single inheritance hierarcy for our greeting produ­cers. That would probably involve reconsidering both the contracts and the implementations. With regards to contracts, we might come to the conclu­sion that the “stronger” version of the contract is needed, i.e. that we may throw an exception for short names.

With regards to implementation, we could come up with a base class like this:

**public abstract class GreetingBase : ICheckedGreeting**

**{**

**public void SayHello(string name)**

**{**

**if (name.Length < 3)**

**{**

**HandleShortName(name);**

**}**

**else**

**{**

**SayHelloUnconditional(name);**

**}**

**}**

**protected virtual void SayHelloUnconditional(string name)**

**{**

**Console.WriteLine($"Hello {name}");**

**}**

**protected abstract void HandleShortName(string name);**

**}**

We are using the **Template Method** pattern to implement **SayHello**, and are thereby deferring the decision on how to handle short names to the derived classes. This is in compliance with the contract, since the contract only states that **SayHello** may throw an exception in case of short names. We can now create two implementations which derive from **GreetingBase**:

**public class GreetingStrict : GreetingBase**

**{**

**protected override void HandleShortName(string name)**

**{**

**throw new ArgumentException("Name too short!");**

**}**

**}**

**public class GreetingRelaxed : GreetingBase**

**{**

**protected override void HandleShortName(string name)**

**{**

**SayHelloUnconditional(name);**

**}**

**}**

Both of these implementations are obeying the interface contract, and since they now implement the same interface, they can be used interchangeably. However, the client now needs to be prepared for handling a possible exception:

**public class Client**

**{**

**public void Run()**

**{**

**DoSomething(new GreetingRelaxed()); // OK**

**DoSomething(new GreetingStrict()); // OK**

**}**

**public void DoSomething(ICheckedGreeting obj)**

**{**

**List<string> names = new List<string> {"Alex, ", "Betty", "Bo"};**

**foreach (string name in names)**

**{**

**try**

**{**

**obj.SayHello(name);**

**}**

**catch (Exception e)**

**{**

**Console.WriteLine($"Exception when " +**

**$"calling SayHello: {e.Message}");**

**}**

**}**

**}**

**}**

It is thus the client which “pays the price” for this approach to the implementation, in the sense that the client code becomes more complex. Is this an indication that we might be on the wrong path here? Not really, since it is a natural consequence of the contracts we have chosen to set up. Judging whether these contracts are correct is another matter, which ultimately leads us back to the question of what the specific requirement for the applications are.

## Postconditions

We can perceive preconditions as certain criteria that need to be fulfilled in order to invoke a specific functionality, as illustrated by ensuring that arguments to a method comply with certain conditions. **Postconditions** can similarly be perceived as certain criteria that must be fulfilled after a specific functionality has been executed. If we are e.g. calling a method on an object, a postcondition could specify conditions on the state of the object after the method call has executed, but also conditions on the return values from methods. Suppose we have defined a small interface **IEmployee**:

**public interface IEmployee**

**{**

**/// <summary>**

**/// Contract: the yearly salary returned must**

**/// be a value between 10,000 and 1,000,000**

**/// </summary>**

**int GetYearlySalary();**

**}**

A contract-compliant – but maybe a bit useless – implementation of this interface could be this:

**public class Employee : IEmployee**

**{**

**private const int lowerLimit = 10000;**

**private const int upperLimit = 1000000;**

**private static Random \_random = new Random();**

**// Not sure I want to work in this place...**

**public virtual int GetYearlySalary()**

**{**

**return \_random.Next(lowerLimit, upperLimit);**

**}**

**}**

Finally, a client may need to process a list of employee objects, by placing each object into a salary “bracket”, like this:

**public class Client**

**{**

**private List<List<IEmployee>> \_employeesBySalaryBracket;**

**public Client()**

**{**

**\_employeesBySalaryBracket = new List<List<IEmployee>>();**

**// …initialise the list properly**

**}**

**public void PutEmployeesIntoBrackets(List<IEmployee> employees)**

**{**

**foreach (IEmployee emp in employees)**

**{**

**int bracketIndex = (emp.GetYearlySalary() - 10000) / 1000;**

**\_employeesBySalaryBracket[bracketIndex].Add(emp);**

**}**

**}**

**}**

The effect of calling **PutEmployeesIntoBrackets** should thus be to place each object in a salary “bracket”, the specific bracket being calculated from the yearly salary. All this should work nicely, but it does hinge on the assumption that the salary can never be lower than 10,000. If the salary is always in the interval 10,000 to 1,000,000, we will have 1,000 brackets being created, with indices from 0 to 999. Now a new CEO – who is already extremely wealthy – enters the company. As an altruistic gesture to the company and its employees, he forfeits any salary for at least the first year. In order to represent this sort of employee, we create a new class **AltruisticCEO** which derives from **Employee**:

**public class AltruisticCEO : Employee**

**{**

**public override int GetYearlySalary()**

**{**

**return 0;**

**}**

**}**

If we now try to execute **PutEmployeesIntoBrackets** on a list of employee objects in-cluding an **AltruisticCEO** object, we will – not surprisingly – experience an **Argu­ment­OutOfRange** exception, since the code will try to place the CEO in a salary bracket with index -10…

What have we done here? We have weakened a postcondition, which was one of the “don’ts” for subclasses wishing to comply with Liskov Substitution. We have widened the range of possible return values from calling **PutEmployeesIntoBrackets**, and the­reby jeo­pardized any code relying on the original range! Just as in the previous exam­ple, the reasoning behind implementing the **AltruisticCEO** class seems sound, but we have actually violated the contract on the **IEmployee** interface…

## Invariants

Having defined postconditions as conditions on the state of an object after calling a method, it might be a bit hard to see the difference between postconditions and invariants. An **invariant** should be perceived as a condition that remains true for the entire lifetime of an object.

Let’s create a variation over the previous **IEmployee** interface. We have limited the contract to concerning the value of the property **TaxPercentage**:

**public interface IEmployee**

**{**

**int GetYearlySalary();**

**int GetYearlySalaryAfterTax();**

**/// <summary>**

**/// Contract: the returned value must be the same throughout**

**/// the lifetime of any object implementing this interface.**

**/// </summary>**

**int TaxPercentage { get; }**

**}**

A first implementation of this interface could be the below class (again, we are only interested in the implementation of the **TaxPercentage** property):

**public class TaxedEmployee : IEmployee**

**{**

**public TaxedEmployee(int taxPercentage)**

**{**

**TaxPercentage = taxPercentage;**

**}**

**public int GetYearlySalary() {…}**

**public int GetYearlySalaryAfterTax() {…}**

**public int TaxPercentage { get; set; }**

**}**

This is a valid – i.e. compilable – implementation of **IEmployee**, but somewhat fragile, since we are exposing the **set**-part of **TaxPercentage** by marking it as **public**. Any client of **Taxed­Employee** objects may thus change the value of **TaxPercentage**, so we have to be absolutely sure that clients only ever see such an object through a refe­ren­ce of type **IEmployee**. Even tho­ugh we have only included the **get**-part of **TaxPer­centage** in the interface, it is perfectly valid to implement both the **get**- and **set**-part as **public** in a class implementing the interface.

The obvious solution to this problem is to change the accessibility for the **set**-part of **TaxPercentage**. Changing the accessibility to **protected** seems to solve the problem. However, we are then still at the mercy of any class deriving from **TaxedEmployee**, since they are then free to change the value of **TaxPercentage**. The conclusion is then that the **set**-part needs to be set to **private**. Not a very complex solution, but it again just shows that it is very easy to open the door for contract violations, if you do not carefully consider the consequences of allowing clients (including derived classes) access to the inner workings of a class.

## Are we done?

We have spent quite some time on discussing the implications of the Liskov Substi­tu­tion principle, and the conclusions may seem dire: unless you are very careful when using inheritance, you may easily end up with classes that are not compliant with the principle, and may thus break client code simply by introducing new classes, which in turn means you are also violating the Open/Close principle…

Still, not everything is lost. The example of implementing greeting generators hope­fully shows that you can – and should – definitely still use inheritance as a tool for implementing variations over behavior. If such variations do not affect the general conditions relating to input arguments and return values, you are usually good to go. If an interface only makes you obligated by contract to e.g. return a string, you can still use inheritance to implement and make available variations over this obligation. The potential problems start if you begin to tamper with these general conditions.

Such “tampering” could in a sense also be perceived as actually changing the type of input arguments and/or return value. Consider once again our example for an inter­face for generating greetings:

**public interface IGreeting**

**{**

**void SayHello(string name);**

**}**

Suppose we simply define it to be “illegal” to call **SayHello** with a string shor­ter than three characters. That is, calling it with a shorter string should always result in some kind of exception. What if we could make it impossible to call **SayHello** with an illegal string? The problem is that **string** is not really the perfect type for our needs. All legal names are strings, but not all strings are valid names. Suppose we create a new class **Name**, solely with the purpose of representing legal names:

**public class Name**

**{**

**public string Value { get; }**

**public Name(string value)**

**{**

**if (value.Length < 3) { throw new …}**

**Value = value;**

**}**

**}**

A bit over the top? Maybe, but what if we need to ensure that we use valid names in many places in the application? Note how an implementation of **IGreeting** could look now:

**public class Greeting : IGreeting**

**{**

**public void SayHello(Name name)**

**{**

**Console.WriteLine($"Hello {name.Value}");**

**}**

**}**

The “if-name-is-too-short-then-throw-exception” logic is now once and for all isola­ted inside the **Name** class, and all clients relying on receiving a valid name can now confidently omit any sort of validation of the incoming value.

The example of postcondition violation can be seen in the same light. Suppose that C# contained a “ranged integer” type, perhaps with a syntax like **[10…100]** (some lang­uages actually contain such types). We could then express the return type of **GetYearlySalary** as being **[10000…1000000]** instead of just **int**. If a new class then tries to return **0** from **GetYearlySalary**, it would simply be a type violation, which the compiler would detect and report. Even though C# does not have such types, we can still just use the idea from above and implement a dedicated class **ValidSalary**, which would contain the specific restrictions on salaries.

## Final words, for now…

Among the five **SOLID** principles, the Liskov Substitution principle does have a – maybe well-deser­ved – reputation of being the most cryptic to understand and chal­lenging to apply. Even after this rather tho­rough discussion, there are several finer details to dive into, for those interested in fully understanding the principle. As al­ways, there are many materials available for further study. Among those, I would like to mention the book **Adaptive Code via C#[[2]](#footnote-2)**, which itself is a book entirely dedicated to discussing the **SOLID** principles. Not a book for those completely new to software development, but with quite a lot of in-depth treatment of all principles.

Finally, I really don’t hope that this discussion discourages you from using inheritance ever again. Inheritance is still a strong and relevant tool, but as is the case whenever you are wielding a powerful weapon: *with great powers come great responsibility…*

# The Interface Segregation principle

In the discussion of the **Single Responsibility** principle, the main conclusion was to separate functionality into highly cohesive classes, i.e. classes that are focused on implementing one specific responsibility. Also, the resulting classes should strive at establishing dependencies through **interfaces**. The **Interface Segregation** principle can be perceived as a rather natural consequence of this idea; if we separate func­tionality into small, cohesive classes, and also define corresponding interfaces for these classes, these interfaces must then also be small and focused.

The above rationale is valid, but we can take the idea of segregation even fur­ther w.r.t. interfaces. For classes, we suggested the rule-of-thumb that the process of creating smaller and smaller classes should end when the usage pattern for the methods in a class is the same for all methods. For interfaces, however, there are sound reasons to take the process even further.

## Segregation of a CRUD interface

Let’s see an example right away: We have several times created interfaces and classes for implementing the concept of a “catalog”, which is essentially a data container with an interface containing methods for the classic **CRUD** (**C**reate, **R**ead, **U**pdate and **D**elete) functionalities. Typically, we define an interface like this:

**public interface ICreateReadUpdateDelete<T>**

**{**

**void Create(int key, T obj);**

**T Read(int key);**

**void Update(int key, T obj);**

**void Delete(int key);**

**}**

We have intentionally chosen the rather verbose name **ICreateReadUpdateDelete** for the interface, but we will also refer to this interface as **ICRUD**, for brevity. Given this interface, we can then create a class (say, **CatalogV1**) which implements this interface. The implementation itself is not of interest here, but the main point for now is that **CatalogV1** implements the **ICRUD** interface.

Suppose now that some other part of the code needs to use the functionality defined in the **ICRUD** interface, or perhaps just a part of the functionality. The class **ClientV1** below is an example of this:

**public class ClientV1<T>**

**{**

**private ICreateReadUpdateDelete<T> \_catalog;**

**public ClientV1(ICreateReadUpdateDelete<T> catalog)**

**{**

**\_catalog = catalog;**

**}**

**public void DeleteMany(List<int> keys)**

**{**

**keys.ForEach(\_catalog.Delete);**

**}**

**}**

Two features of **ClientV1** are of interest here:

* It needs a reference of type **ICRUD**
* It only uses delete-related functionality

Is there a problem lurking here? We can at least recognise that we are in principle giv­ing **ClientV1** more than it needs in terms of knowledge and functionality. By giving it a reference of type **ICRUD**, we are telling **ClientV1** that the object at the end of the reference implements the full **ICRUD** interface. **ClientV1** doesn’t need to know this in order to get its job done; it only needs to know that there is a **Delete** method avail­a­ble at the end of the reference, that it may call. Also – and maybe more problematic – we are granting **ClientV1** full access to all of the **CRUD** functionality. Maybe the en­tire system relies on an assumption that **ClientV1** will not e.g. insert new objects into the catalog, and by providing it a reference on which this is possible, somebody may at some point break this assumption, simply due to insufficient knowledge about this assumption.

There are thus definitely valid arguments for changing this structure. Let’s do this by breaking the **ICRUD** interface into four separate – and extremely small – interfaces:

**public interface ICreate<T>**

**{**

**void Create(int key, T obj);**

**}**

**public interface IRead<T>**

**{**

**T Read(int key);**

**}**

**public interface IUpdate<in T>**

**{**

**void Update(int key, T obj);**

**}**

**public interface IDelete<T>**

**{**

**void Delete(int key);**

**}**

This may seem like going to the extreme, but as we will see later, it actually makes good sense. With these interfaces in place, we can create a new version of the cata­log implementation, e.g. called **CatalogV2**, which will implement these four inter­faces. Note that the implementation itself will be identical to the implementation of **CatalogV1**. We will also create a new version of the client class, named **ClientV2**:

**public class ClientV2<T>**

**{**

**private IDelete<T> \_deleteImpl;**

**public ClientV2(IDelete<T> deleteImpl)**

**{**

**\_deleteImpl = deleteImpl;**

**}**

**public void DeleteMany(List<int> keys)**

**{**

**keys.ForEach(\_deleteImpl.Delete);**

**}**

**}**

This class is indeed different than **ClientV1**, since it only receives a reference of type **IDelete**. This is just enough to get the job done – but not do anything else – which is precisely what we wanted.

How do we then establish an actual dependency between a client and a catalog? Some­where in the code, some entity – e.g. an “injector” class – will have to create specific client and catalog objects, and create the dependency:

**CatalogV2<T> catalogV2 = new CatalogV2<T>();**

**ClientV2<T> clientV2catV2 = new ClientV2<T>(catalogV2);**

So, even though **CatalogV2** implements all four **CRUD** interfaces, the client only sees the delete-part of the implementation, which is just what we want.

## Interface segregation and the Decorator pattern

This ability to “fine-tune” the level of knowledge objects have about each other is in itself a very useful consequence of interface segregation. Also, it provides some addi­tional degress of freedom w.r.t. configuration. By “configuration”, we mean that we can easily create modified – but still useful – versions of the original functionality, with­out affecting the client at all. A very common pattern for implementation of such modifi­cations is the **Decorator**[[3]](#footnote-3) pattern.

The essence of the **Decorator** pattern is that a “de­co­rator” class will add extra func­tion­ality to a given interface, but will do so by implemen­ting the same interface itself, such that clients of the interface cannot tell whether they are using a reference to an original implementation or a decorated implementation.

In our example, we could imagine that we sometimes require that delete operations must be confirmed in some way, before they are executed. Below is an implementa­tion of this requirement, using the **Decorator** pattern:

**public abstract class DeleteWithConfirm<T> : IDelete<T>**

**{**

**private IDelete<T> \_deleteImpl;**

**protected DeleteWithConfirm(IDelete<T> deleteImpl)**

**{**

**\_deleteImpl = deleteImpl;**

**}**

**public void Delete(int key)**

**{**

**if (ConfirmDelete(key))**

**{**

**\_deleteImpl.Delete(key);**

**}**

**}**

**protected abstract bool ConfirmDelete(int key);**

**}**

Note that the **DeleteWithConfirm** class:

* Has a reference of type **IDelete**
* Itself implements **IDelete**
* Wraps the call of **delete** into a condition
* Uses an abstract method **ConfirmDelete** in the condition
* Is itself an abstract class

The **DeleteWithConfirm** class thus makes use of the **Template Method** pattern, in the sense that **Delete** now defines the general algorithm for the decorated version of the original **Delete**, but defers one step (the condition) to a derived class. We can then create a derived class, say **DeleteWithConfirmImpl**, which implements a specific strategy for obtaining a confirmation.

The whole point of this is to enable the following:

**CatalogV2<T> catV2 = new CatalogV2<T>();**

**DeleteWithConfirmImpl<T> delDeco = new DeleteWithConfirmImpl<T>(catV2);**

**ClientV2<T> cliV2A = new ClientV2<T>(catV2); // OK**

**ClientV2<T> cliV2B = new ClientV2<T>(delDeco); // Also OK**

We can now use **ClientV2** with an original implementation of **IDelete**, but also with a decorated version of **IDelete**, without affecting the client at all. Doesn’t that sound a lot like the **Open/Close** principle as well ☺?

## Inheritance vs Composition

This is all fine, and hopefully provides enough evidence for the merits of Interface Segregation. We have, however, left the original **ClientV1** class a bit out to dry… If the **ClientV1** class had indeed been rewritten into the **ClientV2** class, we would not have any issues left, but it could easily be imagined that both the **ClientV2** class and the “legacy” **ClientV1** class would have to co-exist for some time. If we also assume – just to make matters complicated for ourselves – that it has been decided to retire the original **CatalogV1** class, we would need to somehow make **ClientV1** – which requires a reference of type **ICRUD** – and **CatalogV2** compatible.

A first attempt to achieve this could be to use the **Adapter** pattern by inheritance. The **Adapter** pattern is targeted at implementing a given interface by means of another “adapted” interface, and this is indeed the situation at hand. We can in fact implement this very easily:

**public class AdaptedCatalog<T> : CatalogV2<T>, ICreateReadUpdateDelete<T>**

**{**

**}**

That’s it! We can get away with this trivial implementation because the “sum” of the four segregated interfaces is exactly the same as **ICRUD**. There are, however, some limitations in how we can use this adapted catalog:

**AdaptedCatalog<T> adapCat = new AdaptedCatalog<T>();**

**ClientV1<T> clientV1adapCat = new ClientV1<T>(adapCat);**

Sure, **ClientV1** is now able to use **CatalogV2** – since **AdaptedCatalog** inherits from **CatalogV2** – but there is no easy way to introduce “decoration” in the manner we saw earlier. We could perhaps create a class which derives from **AdaptedCatalog** (say, **AdaptedCatalogConfirmDelete**), but that is a bumpy road… What if we later on want to implement a different sort of decoration for delete? We can create yet ano­ther derived class…but what if we also want to be able to decorate some of the three remaining interface? Before we know it, we will have a jungle of derived classes on our hands, one for each combination of decorations. Not where we want to go…

Instead, let’s try to use **composition** instead of inheritance. We create a new class **DecoratedCatalog**, which must also implement **ICRUD** (to make it compatible with **ClientV1**), but instead of inheritance, we let the class contain four references to imple­mentation of each of the four segregated interfaces:

**public class DecoratedCatalog<T> : ICreateReadUpdateDelete<T>**

**{**

**private ICreate<T> \_create;**

**private IRead<T> \_read;**

**private IUpdate<T> \_update;**

**private IDelete<T> \_delete;**

**public DecoratedCatalog(**

**ICreate<T> create,**

**IRead<T> read,**

**IUpdate<T> update,**

**IDelete<T> delete)**

**{**

**\_create = create;**

**\_read = read;**

**\_update = update;**

**\_delete = delete;**

**}**

**public void Create(int key, T obj)**

**{**

**\_create.Create(key, obj);**

**}**

**// Likewise for Read, Update and Delete**

**}**

How can we then use this new class. In a much more flexible manner:

**CatalogV2<T> catV2 = new CatalogV2<T>();**

**DeleteWithConfirmImpl<T> delDeco = new DeleteWithConfirmImpl<T>(catV2);**

**// OK: All references are to original CatalogV2**

**DecoratedCatalog<T> decoCatA = new DecoratedCatalog<T>(**

**catV2, catV2, catV2, catV2);**

**// OK: Three references are to original CatalogV2,**

**// one reference is to decorated interface.**

**DecoratedCatalog<T> decoCatB = new DecoratedCatalog<T>(**

**catV2, catV2, catV2, delDeco);**

**// OK**

**ClientV1<T> clientV1decoA = new ClientV1<T>(decoCatA);**

**// Also OK**

**ClientV1<T> clientV1decoA = new ClientV1<T>(decoCatB);**

We now have full fredom w.r.t. how to configure the “catalog” – or more precisely; the object(s) which implement the four segregated interfaces – but retain the com­patibility with **ClientV1**. Interface segregation was a key factor in achieving this, but it is also an illustration of the principle of **preferring composition over inheritance[[4]](#footnote-4)**.

Composition often liberates us from drowning in a plethora of derived classes, each implementing some specific combination of functional variants. As we also saw when discussing the **Liskov Substitution** principle, excessive use of inheritance can often make it very delicate to add subclasses to a system. This problem is also alleviated when favoring composition over inheritance.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | SOLID.1 |
| **Solution** | WildLife |
| **Purpose** | Try to rewrite code according to the **Inversion of Control** (IoC) prin­ciple. |
| **Description** | The project contains a model for animal behavior. The model contains four kinds of animals: **Mouse**, **Rabbit**, **Fox** and **Tiger**. An animal is modeled with a num­ber of properties (**Kind**, **Gender**, **Age**, **Dead**, **Hungry**, **Sleepy**) and methods (**Sleep**, **Hunt**, **Scavenge**, **Eat**, **Mate**, **Flee** and **Idle**). Each method models a very basic behavior.  All animals live in a **World**, where they can encounter other animals. These ani­mals can be of any of the four kinds, and may be male or female. The behavior of an animal will be influenced by its own “state” (i.e. the properties above), and the animals encountered in the world.  The behavior of a specific animal is modeled by combining the methods from above. Even though the behavior is individual for each kind of animal, the beha­viors follow a similar pattern:  **if** (condition1) { // perform 1.priority behavior}  **else if** (condition2) { // perform 2.priority behavior}  **else if** (condition3) { // perform 3.priority behavior}  **else if** (condition4) { // perform 4.priority behavior}  **else** { // perform idle behavior}  The initial version of the project contains two folders   * **Common**: Contains base and support classes for animal behavior (in particular the **AnimalBase** class), and for very simplistic world modeling. * **Original**: Contain classes modeling behaviors for the four kinds of animals. It also contain the base class **AnimalBaseOriginal** – which itself inherits from AnimalBase – but this class does not really contain any functionality. |
| **Steps** | 1. Take a good look at the **AnimalBase** class in the **Common** folder. In parti­cu­lar, make sure you understand the role of the **Act** method (which is de­fined as being **abstract**). Where do you expect **Act** to be implemented? 2. Take a look at the four animal-specific classes in the **Original** folder. Note how **Act** is implemented. How does the implementation match the model described above? Is there anything preventing us from implementing a behavior in a completely different way? 3. Create a new folder (e.g. called **IoC**), and try to re-implement the model for behavior with the **IoC** principle in mind, placing the new classes in the new folder. This new folder will thus also end up containing (at least) a base class and four animal-specific classes. In order to “comply” with the **IoC** principle, it should be a base class that controls the general structure of the **Act** method, while the animal-specific classes should only fill in the animal-specific parts that vary. 4. The exercise is not as such about actually testing the code, but you can run the project as-is, which will execute a small test of the given code. If you wish to compare the given implementation to your own implemen­ta­tion, you can modify or extend the test code in the **Tester** class. 5. [Extra] The behavior model specified above is a bit limited, in the sense that all animals must have a specific number of behaviors (four, plus the idle behavior). See if you can create an implementation where a variable number of condition/behavior pairs can be specified in order to model the behavior of an animal. It should still be the base class that is responsible for execution of the **Act** method. |

|  |  |
| --- | --- |
| **Exercise** | SOLID.2 |
| **Solution** | NaiveRPG |
| **Purpose** | Try to find opportunities to use **Inversion of Control** (**IoC**) and **Depen­dency Injection** (**DI**) to improve the structure of an application. |
| **Description** | The **NaiveRPG** project contains seven classes, which fall into three categories:   * Game participants: **Character**, **Bear** and **Troll**. * Game items: **Boots**, **Sword** and **Shield**. * Game simulator: **Game**   The application does implement an extremely simple Role-Playing Game (RPG), but it has a very inflexible structure. |
| **Steps** | 1. Investigate the classes mentioned above, until you have a reasonable understanding of all the classes. Most classes are quite simple. 2. Now focus on the **Run** method in the **Game** class. This method imple­ments the “engine” for the game, i.e. it is intended to manage the general progression of the game. In its current form, the method is however very inflexible, and running the game will produce the same result over and over, since the setup (participants and items) is always the same, and cannot be changed unless the method itself is updated. 3. Use the **IoC** and **DI** principles, your knowledge about Object-Oriented Pro­gram­ming and your common sense to improve the structure of the imple­mentation. The goal should be to make the **Run** method as flexible as possible, i.e. it should be as independent as possible with regards to spe­cific participants, items, etc.. You are free to change the structure in any way you want; this could e.g. be by using inheritance, defining interfaces, adding properties to classes, adding parameters to methods, etc.. 4. Once you feel that the structure has been sufficiently improved, feel free to add additional elements to the game, e.g. additional game items and participants, or perhaps completely new game elements like e.g. weapon enhancements or more advanced combat mechanics. |

|  |  |
| --- | --- |
| **Exercise** | SOLID.3 |
| **Solution** | ShapesCompare |
| **Purpose** | See a simple example of violation of the **Liskov Substitution** principles, and try to come up with a couple of ways to resolve the violations. |
| **Description** | The project contains the folder **Original**, which contains the interface **IShape**, a couple of classes which implement the interface, and a class **Tester** which per­forms a small test. More specifically, the test tries to compare the ratio between the areas of various shapes. |
| **Steps** | 1. Get an overview of the classes in the **Original** folder, until you understand the purpose of each class (all classes are fairly simple). 2. Take a closer look at the method **CompareShapes** in **Tester**. Try to figure out what could cause an **ArgumentException** to be thrown. 3. Run the application *as-is*. It will execute the test defined in the **Run** method in **Tester**. The test prints out the ratio of areas of various pairs of shapes. 4. In the **Run** method, try to uncomment the line which adds a **Point** object to the list of shapes. Run the application; you should now see an exception. 5. The problem seems to be that the **Point** class implements the **IShape** inter­face – since **Point** inherits from **ShapeBase** – but also returns an area of 0 (zero). This violates the contract specified for the **IShape** interface. 6. Figure out what to do about this problem. Your solution should make it impossible to use **Point** objects in the test code. The solution is probably simpler than you think… (hint: what does the contract say about what it means to be a “shape”…?) 7. The code contains another similar problem, which is related to the **Circle** and **Square** classes. Try to see if you can spot the problem, just by looking at the class definitions. If you need a hint, go to the **Run** method in **Tester**, and uncomment the line of code which creates a **Square** object like this: **new Square(7, 5, 0)**, and run the application. What happens? 8. The problem seems to be that we can create **Circle** and **Square** objects with non-positive values for radius and side length, respectively. Clearly, a circle with a radius of e.g. -3 does not make sense. Figure out how we can pre­vent this problem. You could e.g. implement various checks of the values used in the constructors for **Circle** and **Square**, but you could also consider if we should introduce a brand new type for representing positive decimal values… |

|  |  |
| --- | --- |
| **Exercise** | SOLID.4 |
| **Solution** | FOOrrestGump |
| **Purpose** | Go beserk in an acute case of **SOLID** fever! That is, rewrite the given code with generalisation in mind, using the **SOLID** principles where-ever you see an opportunity for it. |
| **Description** | The project contains a very specific implementation of a imaginary scenario, where the fictional character *Forrest Gump* eats a box of chocolates. The folder **FGScenario** contains the code which implements this scenario. |
| **Steps** | 1. Take a look at the classes in **FGScenario**, and the implementation of the scenario in **Main**. How reusable and configurable are the classes used for the implemen­tation? 2. Before embarking on your re-implementation, try to think about a more gene­ral model for the type of scenario we are dealing with. You could e.g. try to think in terms of “consumer”, “container”, “consumable”, “factory”, “scenario”, etc.. 3. Go ahead and create a much more general framework for modeling scena­rios. Use as many **SOLID** principles, interfaces, abstractions etc. as you wish; it is okay to go a bit “over the top” in this exercise ☺. 4. Once you have implemented such a framework, make sure that you can actually implement the original scenario with your new framework. If you want an extra challenge, you could set a limitation saying that you may not change any of the existing classes (Hint: adapters could perhaps be useful here…). 5. Try to use the framework for implementing a different scenario. That could e.g. be *Jenny* working her way through a bag of peanuts (or alter­na­tively a big can of worms, which she deserves for being such an ungrate­ful b!tch…). |

1. https://en.wikipedia.org/wiki/SOLID [↑](#footnote-ref-1)
2. Adaptive Code via C#, by Gary McLean Hall, ISBN: 978-0-7356-8320-4 [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/Decorator\_pattern [↑](#footnote-ref-3)
4. https://en.wikipedia.org/wiki/Composition\_over\_inheritance [↑](#footnote-ref-4)