Object-Oriented Programming with C#

Object-Oriented Programming, Part III

By Per Laursen
03-08-2025

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.

The next step is to consider how to put it all together. We have to some extent done this already, since we have learned about various ways to relate classes to each other, by means of interfaces, inheritance, composition, etc.. Still, the scenarios we have investigated have usually been rather simple, and perhaps also without a well-defined purpose. A seasoned software developer should also know about techniques for how to use all these tools for various real-life purposes. Such purposes – along with general techniques for achieving them – can be formulated as so-called **Design Patterns**.

A **Design Pattern** is – as the term suggests – a design (in terms of classes and interfaces) for how to solve a commonly occurring problem in software development. Design patterns are not theoretical constructions, but rather "best practices" which have been recognised over time as sound solutions to common problems. Quite a lot of material is available on design patterns, the most well-known probably being the book Design Patterns[1], which in the mid-90s named and described 23 specific design patterns, divided into three main categories: Creational, Structural and Behavioral. The names assigned to the individual design patterns are still used today, and thus provides a common "vocabulary" for software developers, enabling developers to refer to non-trivial designs by a single phrase. We will investigate a few of these design patterns in more detail in this chapter.

Before diving into specific design patterns, we will try to put them into perspective. As stated above, design patterns are not theoretical constructions thought up by academics in an ivory tower, but rather observed "best practices" which have been condensed into their very essence, both with regards to formulation and solution. But when is something considered a "best practice"? If something is indeed a "best practice", then it must be better than other practices, yes? This in turn requires us to think about how we measure the "value" of a practice, which ultimately brings us to a core question

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

[1] Design Patterns, by Erich Gamma, Richard Helm, Ralph Johnsen and John Vlissides, ISBN 0-201-63361-2

This is a very tricky question, since it will be highly dependent on the specific circumstances for a specific software development activity. A somewhat subjective answer to the question could be: to develop software of the highest possible quality. But that just begs another question:

How do we measure quality?

If we are developing software for a nuclear power plant control system, we will probably rate "lack of errors" very high on a quality scale, 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 quality measures; if a 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 "reasonable" version of the 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 a more traditional perspective on quality as meaning "error-free" does not apply any longer.

If that is indeed the case, 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 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?

This text does not aspire to be a complete treatment of all the useful properties a piece of software should posses. Other desirable properties are e.g. testability, reliability, performance, instrumentation, etc.. Discussions about such properties must be found elsewhere; we will primarily focus on **reusability** and **extensibility**.

Having convinced ourselves that reusability and extensibility are indeed positive properties worth striving for, can we now jump into specific design patterns? Not quite yet. First, we will have a look at a couple of more general software design **principles**, which come into play again and again when discussing specfic design patterns. More specifically, we will look at two such principles:

- The **Open/Closed** principle
- The **Dependency Injection** principle

These two principles form the **O** and **D** in a larger set of such principles know as the **SOLID** principles, which is also a well-established term in software development. The complete list of SOLID principles is:

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

As for Design Patters, a lot of material detailing the SOLID principles can be found elsewhere[2].

[2] A very recommendable book on SOLID is: "Adaptive Code via C# (Agile coding with design patterns and SOLID principles", by Gary M. Hall, ISBN 978-0-7356-8320-4

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? 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 classes in question.

Let’s illustrate these concepts with an example: Suppose a class **Client** makes use of a class **CalculatorV10** (V10: version 1.0) 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 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 itself, but rather receive a calculation object through the constructor parameter. 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 – existing and future – 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 about 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. We will see examples of how to address this problem later in this chapter.

A common theme in strategies for implementing the open/closed principle is to remove knowledge about specific types from the client code, and replace it with e.g. knowledge about interfaces only. A third party must then somehow supply specific implementations of these interface to the client code, e.g. as constructor parameters. This way of establishing relations between otherwise loosely coupled classes is in fact what the **Dependency Injection** principle is all about.

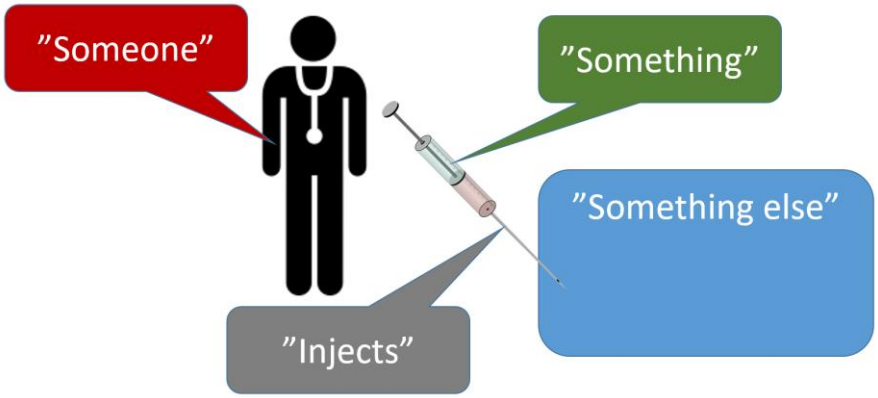
The Dependency Injection principle

As discussed in the previous chapter, we often strive at creating "reusable" classes, which makes it possible to use such classes together with entirely new classes, without having to modify said reusable classes, or – in the worst of scenarios – having to write an entirely new class ourselves ☺. If we can pick a well-tested, well-documented *off-the-shelf* class and easily reuse it in our own project, we can probably develop better, cheaper and less error-prone applications, as compared to having to develop the entire application from scratch.

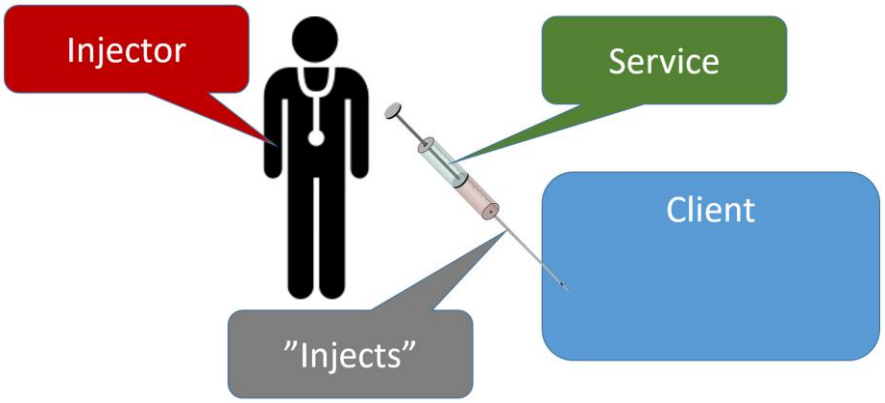
The principle known as **Dependency Injection** is generally considered a very important principle with regards to developing reusable classes, so we will also investigate

this principle a bit, before seeing it in action in some specific Design Patterns. We will even look a Dependency Injection in a somewhat broader perspective than just when applied to relations between classes, since it makes sense to discuss it at lower levels as well. In a sense, we have been using this principle for quite some time already…

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, no. 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 Dependency Injection 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;
}
```

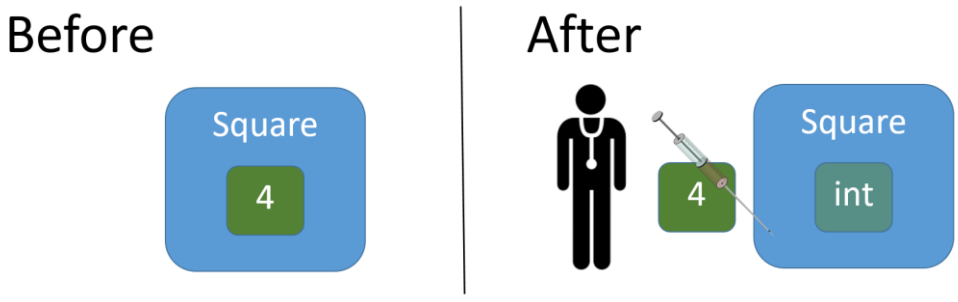
The method does what it advertises; it calculates the square of the number 4. If someone later on asks us for a method which can calculate the square of 6, we can quickly deliver:

```
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 re-placing 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 thus 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 Dependency Injection 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 – will set 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, hence the +1 after the call). So, what exactly makes this a model of a 6-sided die? That is of course the number 6, used in the call of **_random.Next**. If we 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 Dependency Injection 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 the object is constructed, by adding a parameter to the 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 Dependency Injection 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 Dependency Injection 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, and let the caller inject a specific value – here a specific selection criterion – when the method is called.

Dependency Injection – class/interface level

In order to see an example of Dependency Injection at the class/interface level, we need not look any further than one of the examples used when discussing the Open/Closed principle. Our starting point was a **Client** class as given below:

```
public class Client
{
	public CalculatorV10 _calculator;

	public Client()
	{
		_calculator = new CalculatorV10();
	}

	public void ProcessData(Data d)
	{
		int result = _calculator.Calculate(d);
		//...
	}
}
```

The main problem with this class is the tight coupling to the **CalculatorV10** class. If we at some point need to upgrade to a newer version of the calculation – perhaps in the form of a class **CalculatorV11** – we need to change the implementation of **Client**. We can remedy this to some extent by adding a parameter of type **CalculatorV10** to the **Client** constructor, which will relieve the client of having to explicitly create a new object. Furthermore, this makes the dependency between **Client** and **CalculatorV10** more visible to the outside world. In the above code, a **CalculatorV10** object is created inside the **Client** constructor, but the only way to become aware of this fact is to inspect the actual source code, which may not always be possible (the class

could e.g. be a third-party class). This can make it harder to test the class properly. Adding a parameter of type **CalculatorV10** to the **Client** constructor does however not remove the tight coupling between **CalculatorV10** and **Client**; it just exposes it. If we now introduce a general calculation interface **ICalculator**, and use this interface as the type for the **Client** constructor parameter, the coupling is lifted:

```
public class Client
{
	public ICalculator _calculator;

	public Client(ICalculator calculator)
	{
		_calculator = calculator;
	}

	public void ProcessData(Data d)
	{
		int result = _calculator.Calculate(d);
		//...
	}
}
```

This version of **Client** does not depend on specific calculation classes. Instead, a third party must "inject" a specific calculation object, as part of creating a **Client** object:

```
	ICalculator aCalculator = new CalculatorV10();
	Client aClient = new Client(aCalculator);
```

This is exactly the same code we saw when discussing the Open/Closed principle, and Dependency Injection is indeed a very useful tool for implementing this principle. Furthermore, this makes testing of the **Client** class 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" or "mock" 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 mock[3] object can be used instead, which is likely to make the test both faster and more reliable.

There is not much more to Dependency Injection than this, except one – quite important – question: Who should be responsible for actually performing this dependency injection? The next chapter will provide a couple of possible answers to this question.

[3] https://en.wikipedia.org/wiki/Mock_object

Design Patterns – Creational Patterns

In the discussions of the Open/Closed and Dependency Injection principles, a recurring theme was to remove tight couplings between client code and specific classes. More specifically, we saw that client code should – if at all possible – abstain from explicitly creating new objects, since creation requires the creator to explicitly state the type of the created object. Instead, client code should receive a reference to an object created by a third party, since such a reference can be of e.g. an interface type. This provides much more flexibility with regards to how to couple client code and specific interface implementations together. The act of creating objects of a specific type has thus been externalised to this "third party".

How should we then implement this "third party"? Does it manifest itself in the form of a method, a class or something else? The set of design patterns in the **creational patterns** category offer various approaches to this issue. We will here discuss two of these patterns, which are fairly closely related: the **Factory Method** pattern and the **Abstract Factory** pattern.

Factory Method

Suppose we are developing software for a courier service, which needs to plan how to transport goods to certain locations, given e.g. a set of available vans. Such a planning process may have as objective to minimise the total number of kilometers driven, or perhaps to minimise the delays by which goods are delivered. The details are as such not important, but planning problems like these tend to be very hard (i.e. require a lot of CPU resources) to solve optimally, where "optimally" means a 100 % guarantee that no better solution exist. However, it is often the case that approximate solutions can be found much faster. There might even exist a whole suite of algorithms for producing solutions to the problem, where the quality of the solution will be directly proportional to the time it takes to produce the solution (the more time available, the better the solution).

This situation could be reflected in the implementation of the software. We imagine a class **PlanningManager**, with a central method **ExecutePlanning**. An initial implementation of the method might look like this:

```
public void ExecutePlanning()
{
	PlanningData data = ReadPlanningDataFromDB();
	PlanningResult result = null;

	int timeAvailable = GetTimeAvailableForPlanning();

	if (timeAvailable < 3)
	{
		ReallyFastPlannerV16 rfp16 = new ReallyFastPlannerV16();
		result = rfp16.CreatePlanFast(data);
	}
	else if (timeAvailable < 12)
	{
		PrettyFastPlannerV24 pfp24 = new PrettyFastPlannerV24();
		result = pfp24.CalculatePlan(data);
	}
	else
	{
		ExactPlannerV37 ep37 = new ExactPlannerV37();
		result = ep37.FindExactSolution(data);
	}

	WritePlanningResultToDB(result);
}
```

We assume that the planning process relies on some set of data, here represented by a class **PlanningData**. The planning process will produce a result in the form of a **PlanningResult** object, which is then saved to a database. The specific details about this are not important either. Furthermore, we assume that various algorithms for planning are available in the form of various classes; these classes might be developed in-house, or might be a third-party product. Such algorithms may be improved over time, so we assume that new versions will be produced regularly (indicated by the version numbers at the end of the class names).

The central feature of the **ExecutePlanning** method is the **multi-if-else** statement, which chooses a planning algorithm according to the available time. As it stands, this might be an acceptable implementation, but it is definitely quite fragile with regards to changes. Whenever a new version of one of the algorithms becomes available, we will have to update the code, which in turn may require re-testing, releasing the code into production, etc.. There is definitely room for improvement.

One major problem with the **ExecutePlanning** method is the dependency of specific planning classes, which induces the problem of having to frequently update the code. The **Factory Method** pattern is an appropriate tool for solving this problem. Instead of depending on specific classes, the client (here the **ExecutePlanning** method) will

use a “factory” object to produce planning objects, without knowing about – and thereby depend on – their specific type. A prerequisite for this is to perform a bit of “homogenisation” of the planning classes. We define a planning interface **IPlanner**, like this:

```
public interface IPlanner
{
	PlanningResult CreatePlan(PlanningData data);
}
```

We now require that all planning classes must implement this interface. For planning classes produced in-house, this is probably not a problem, since it just amounts to renaming an existing method. For third-party classes, it may a bit trickier to achieve, but we will later see a design pattern (named **Adapter**) which can help us with this. For now, we assume that this is possible, such that all planning classes contain a method **CreatePlan** for creating a plan.

The **IPlanner** interface does not in itself help us much. We now define an additional interface **IPlannerFactory**, like this:

```
public interface IPlannerFactory
{
	IPlanner Create(int timeAvailable);
}
```

The **Create** method in this interface is indeed a **Factory Method**. Calling this method will return an object which implements the **IPlanner** interface, but the caller will not be aware of the specific type of the object! This is the whole point; the caller should only know the interface type, since this is enough to make use of the object. We can now simplify the implementation of the **ExecutePlanning** method considerably:

```
public void ExecutePlanning(IPlannerFactory factory)
{
	PlanningData data = ReadPlanningDataFromDB();
	int timeAvailable = GetTimeAvailableForPlanning();

	IPlanner planner = factory.Create(timeAvailable);
	PlanningResult result = planner.CreatePlan(data);

	WritePlanningResultToDB(result);
}
```

We have added a parameter to **ExecutePlanning**, which is a reference (by interface type) to a planner factory object. We can then use this factory to create a planner

object when we need it, simply by calling **Create**. Since the returned object implements **IPlanner**, we can call **CreatePlan** on it. **ExecutePlanning** has no knowledge about specific planning classes, and will not need to change if new planning classes become available.

The **ExecutePlanning** method is obviously much cleaner now, but the choice of a specific planner class has of course not magically disappeared. It will now be implemented in a factory class, e.g. like this:

```
public class PlannerFactoryDay : IPlannerFactory
{
	public IPlanner Create(int timeAvailable)
	{
		if (timeAvailable < 3)
		{
			return new ReallyFastPlannerV16();
		}
		else if (timeAvailable < 12)
		{
			return new PrettyFastPlannerV24();
		}
		else
		{
			return new ExactPlannerV37();
		}
	}
}
```

Note that this class implements the **IPlannerFactory** interface. A call of the planning manager method could now look like this:

```
	PlanningManager pm = new PlanningManager();
	pm.ExecutePlanning(new PlannerFactoryDay());
```

We have thus extracted the responsibility of choosing the appropriate planning algorithm from the **ExecutePlanning** method, and placed it in a dedicated factory class instead. So, have we just moved code around? Indeed we have, but that is also a quality in itself. The **ExecutePlanning** method can now concentrate on coordinating the planning process – including reading and writing data to a data source – while the factory class can focus on producing the appropriate planner object. Also, this division provides more flexibility.

We have named the above implementation of the **IPlannerFactory** interface **PlannerFactoryDay**, to suggest that an alternative implementation – maybe named **PlannerFactoryNight** – could exist. We could imagine that the criteria for choosing a specific

implementation may vary between day and night, e.g. if more CPU resources are available at night. It will then be a simple matter to configure the **ExecutePlanning** method with a different factory implementation, since it only knows the factory objecty by its interface type.

In the above example, we have created a **Factory Method** which takes a single parameter. There are as such no rules w.r.t. how to parameterise a **Factory Method**, and you can also easily imagine factory methods without any parameters, e.g. like this:

```
public interface IPlannerFactorySimple
{
      IPlanner Create();
}
```

An implementation could then look like this:

```
public class PlannerFactoryVeryFast : IPlannerFactorySimple
{
      public IPlanner Create()
      {
            return new ReallyFastPlannerV16();
      }
}
```

This can be perceived as a more “low-level” factory, which only encapsulates a single aspect of planning object creation: creation of “very fast” planning objects. Such factories could be used as building blocks in our previous factory implementation:

```
public class PlanningFactoryGeneral : IPlannerFactory
{
      private IPlannerFactorySimple _veryFastFactory;
      private IPlannerFactorySimple _fastFactory;
      private IPlannerFactorySimple _exactFactory;

      public PlanningFactoryGeneral(
                  IPlannerFactorySimple veryFastFactory,
                  IPlannerFactorySimple fastFactory,
                  IPlannerFactorySimple exactFactory)
      {
            _veryFastFactory = veryFastFactory;
            _fastFactory = fastFactory;
            _exactFactory = exactFactory;
      }
```

```
        public IPlanner Create(int timeAvailable)
        {
            if (timeAvailable < 3)
            {
                return _veryFastFactory.Create();
            }
            else if (timeAvailable < 12)
            {
                return _fastFactory.Create();
            }
            else
            {
                return _exactFactory.Create();
            }
        }
}
```

The final touches could be to turn the hard-coded limits for choosing between the factories into parameters as well. The factory will then be fully configurable (at least with regards to be able to choose between three categories of planners…).

A valid objection to this line of reasoning could be "if I inject a simple factory object (where **Create** doesn't take any parameters) into a class, then why not simply inject the service object the factory will create, since the factory will always create this object…?". This is to some extent true, but consider the case where the receiving class will only make use of the service object if certain conditions are fulfilled, AND it is very expensive to create the service object (say, if it requires to read a large set of data from a database). In that case, the decision to actually create the service object should be left to the receiving class – still without knowing the exact type of the service object – which is exactly what becomes possible by providing the receiving class with a factory object instead. The receiving class can then – if it needs to – initiate the service object creation itself, by calling **Create** on the factory object.

Abstract Factory

The **Factory Method** pattern is aimed at creating objects which all implement the same interface. We can easily imagine that we would need to apply this principle several times, i.e. define factory interfaces and classes for multiple elements in an application. One example could be part of an operating system, which needs to display various GUI controls like Windows, Buttons and Icons on the screen. The code for management of this should not rely on specific control implementations, so it would be natural to define some interfacce for the controls, like **IWindow**, **IButttton** and **IIcon**. Specific implementations of these interfaces can then present the GUI con-

trol in question in a specific way. We could imagine classes like **WindowDefault**, **WindowNatureTheme**, etc., and likewise for other controls. We can then create factory interfaces and classes for each type of control, like this minimal example:

```
public interface IWindowFactory
{
	IWindow Create();
}
```

So far, so good. We can go ahead and create **IButtonFactory** and **IIconFactory** interfaces as well, and imagine that some central part of the GUI management code gets configured with instances of these factories:

```
public class GUIManager
{
	// ...

	public GUIManager(
		IWindowFactory windowFac,
		IButtonFactory buttonFac,
		IIconFactory iconFac)
	{
		// ...
	}
}
```

All seems to be well… but there is a slight catch. Nothing prevents a client from creating a new **GUIManager** object in this way:

```
	GUIManager gm = new GUIManager(
		new WindowFactoryDefault(),
		new ButtonFactoryNature(),
		new IconFactoryIndustrial());
```

This will likely result in a somewhat strangely looking GUI ☺. There is nothing in the code which prevents such unintended mixing between "families" of GUI controls. The solution is fortunately fairly straightforward; instead of defining three factory interfaces with one method in each, we define one factory interface with three methods:

```
public interface IGUIControlFactory
{
	IWindow CreateWindow();
	IButton CreateButton();
	IIcon CreateIcon();
}
```

With this interface in place, we can now implement factory classes that produce GUI controls of the same “family”, e.g. a **GUIControlFactoryNature** class. The **GUIManager** constructor should also be changed:

```
public class GUIManager
{
	// ...

	public GUIManager(IGUIControlFactory igcf)
	{
		// ...
	}
}
```

Creating a **GUIManager** object is now less error-prone:

```
	GUIManager gm = new GUIManager(new GUIControlFactoryNature());
```

This new strategy does not completely prevent mixing of GUI control factories, since we have no control over the actual content of a factory object. If someone is really intent on messing up the GUI, he could write a factory class which does produce GUI control objects from different families. That aside, this new approach does at least state the intention of GUI control creation much clearer; GUI controls do not stand alone; they are part of a GUI control “family”, and the object creation strategy should reflect this.

Design Patterns – Structural Patterns

The design patterns in this category are primarily concerned with the relationships needed between classes and objects, in order to achieve some sort of collaborative effort.

At the class level, such relationships are usually established through inheritance. A simple example could be a class that implements several interfaces. From the perspective of the client code – which we assume always refers to objects by interface-type references – it makes no difference if the behavior exposed through the interfaces is implemented by one or several objects, giving us freedom to choose such an approach, if it is indeed a sound approach by other measures as well.

At the object level, the relationships are established through composition. A class may refer to a completely unrelated class – be it by class type or interface type – in order to implement a certain functionality. We have seen this many times as well; a class that uses a class from the .NET class library – say, the **List<>** class – will often do this by composition, which in practice just means that the class contains an instance field of a certain type, and will either create a corresponding object itself, or receive a reference to an object through a constructor or method parameter (the latter approach is often referred to as **Aggregation**).

We will now take a closer look at four structural design patterns: First, we look at the **Adapter** and the **Proxy** pattern, which are both relatively simple. Next, we look at the **Composite** and the **Flyweight** pattern, which are a bit more involved.

Adapter

During the discussion of the **Factory Method** pattern, the starting point was a method **ExecutePlanning** used for creating a plan for optimal use of certain resources:

```
public void ExecutePlanning()
{
	PlanningData data = ReadPlanningDataFromDB();
	PlanningResult result = null;

	int timeAvailable = GetTimeAvailableForPlanning();

	if (timeAvailable < 3)
	{
		ReallyFastPlannerV16 rfp16 = new ReallyFastPlannerV16();
		result = rfp16.CreatePlanFast(data);
	}
	else if (timeAvailable < 12)
	{
		PrettyFastPlannerV24 pfp24 = new PrettyFastPlannerV24();
		result = pfp24.CalculatePlan(data);
	}
	else
	{
		ExactPlannerV37 ep37 = new ExactPlannerV37();
		result = ep37.FindExactSolution(data);
	}

	WritePlanningResultToDB(result);
}
```

One of the main characteristics of this method is that the specific choice of planning object depends on the value of a parameter. We thus have a set of planning services available, from which we select a single instance. These planning services may e.g. be third-party classes, and may therefore not have anything in common, except the ability to perform the given planning. The code above illustrates such a situation, where the three providers have methods with different names.

Our strategy for cleaning up this code required all planning services to implement the same interface **IPlanner**, as defined below:

```
public interface IPlanner
{
	PlanningResult CreatePlan(PlanningData data);
}
```

This may however not be feasible in practice. If the planning classes are provided by a third party, we cannot just change the definition of the classes, since we might not have access to the actual source code. What then? This is where the **Adapter** design pattern can be applied.

Suppose we create a new class named **ReallyFastPlannerAdapter**. This class must meet two requirements:

- It must implement the **IPlanner** interface.
- It should functionally be identical to the **ReallyFastPlannerV16** class.

This is not particularly hard to achieve:

```
public class ReallyFastPlannerAdapter : IPlanner
{
	private ReallyFastPlannerV16 _adaptee;

	public ReallyFastPlannerAdapter()
	{
		_adaptee = new ReallyFastPlannerV16();
	}

	public PlanningResult CreatePlan(PlanningData data)
	{
		return _adaptee.CreatePlanFast(data);
	}
}
```

We simply encapsulate the “adaptee” – which is the original class that did not implement the desired interface – and delegate the call of **CreatePlan** to the **CreatePlanFast** method of the adaptee. We can create similar adapter classes for all of the available planning classes, and then achieve our goal of having a common interface for all planning providers.

This is of course a quite simple example, since the adaptation essentially boils down to delegating a method call to an adaptee method, which has the same signature (parameter list and return type), but a different name. No pre- or post-processing is needed. An example which requires slightly more complex adaptation could be that of adapting a drawing library. Suppose we want to create a small drawing library, with a few methods defined by an interface **IDraw** (we assume a class **Point** exists):

```
public interface IDraw
{
	void DrawPolyLine(List<Point> points);
	void DrawRectangle(Point start, double xSide, double ySide);
}
```

An existing drawing library **SimpleDraw** is actually available, but it only contains a single method **DrawLine**, which takes two **Point** objects as parameters. We must therefore create an adapter class **SimpleDrawAdapter** – which must implement the **IDraw** interface – and implement the methods in **IDraw** by using the (single) method available in **SimpleDraw**:

```
public class SimpleDrawAdapter : IDraw
{
	private SimpleDraw _adaptee;

	public SimpleDrawAdapter()
	{
		_adaptee = new SimpleDraw();
	}

	public void DrawPolyLine(List<Point> points)
	{
		for (int i = 0; i < points.Count - 1; i++)
		{
			_adaptee.DrawLine(points[i], points[i+1]);
		}
	}

	public void DrawRectangle(Point start, double xSide, double ySide)
	{
		Point endX  = new Point(start.X + xSide, start.Y);
		Point endY  = new Point(start.X, start.Y + ySide);
		Point endXY = new Point(start.X + xSide, start.Y + ySide);

		_adaptee.DrawLine(start, endX);
		_adaptee.DrawLine(start, endY);
		_adaptee.DrawLine(endXY, endX);
		_adaptee.DrawLine(endXY, endY);
	  }
    }
```

Not rocket science by any means, but still a bit more effort needed in order to “close the gap” between the required interface and the available implementation.

Proxy

The purpose of the **Adapter** pattern was to bridge the gap between an interface requested by the client code, and the actual interface presented by a class offering the services the client needs. The **Adapter** class will typically contain a reference to the "adapted" object, and will implement the desired interface by calling methods in the interface of the adapted object. The **Proxy** pattern follows a similar strategy, but with a quite important difference; the object which is referenced has the same interface as the **Proxy** class itself!

How can that be useful? Why not just use the object directly, if the interface is the same? The **Proxy** pattern can be applied if there are reasons to restrict direct access to the object. If such reasons exist, it can be beneficial to hide access to the object behind a "placeholder" object, which is exactly what the **Proxy** pattern is about.

Suppose a class **RoutePlanner** exist, which is capable of route planning in a detailed road network. The class implements an interface **IRoutePlanner**. In order to be able to calculate a route plan, the **RoutePlanner** needs to load in a large data set, before calculation is started. The **RoutePlanner** class is written in such a way that the data set is loaded when a **RoutePlanner** object is created. Creating a **RoutePlanner** object is thus considered an "expensive" operation, since it may take a while to load the data set, and the loaded data set may also consume a significant amount of memory. It would therefore be useful only to create such an object if it is strictly necessary.

Let us also suppose that part of the client code uses a **RoutePlanner** object – or at least an object implementing **IRoutePlanner** – e.g. as a parameter to a method call:

```
public void PrepareForPlanning(IRoutePlanner rp)
{
    // Method body…
}
```

The **PrepareForPlanning** method thus expects to receive a **RoutePlanner** object, but the method might also have a nature where actual planning is only performed when certain conditions apply. In other words; we may often experience that **PrepareForPlanning** did not make active use of the **RoutePlanner** object, and the effort used for creating the object in the first place may therefore be wasted.

A problem like this can be resolved in various ways, depending on the degree of freedom you have with regards to revising the code for the **PrepareForPlanning** method and the **RoutePlanner** class, but you may well be in a situation where neither of the

two can be revised. Using the **Proxy** pattern will then be a feasible approach. We assumed that the **RoutePlanner** class implements an interface **IRoutePlanner**. For simplicity, we assume that the interface only contains a single method:

```
public interface IRoutePlanner
{
	PlanningResult CalculatePlan(PlanningData data);
}
```

We now create a new class **RoutePlannerProxy**, which also implements the **IRoute-Planner** interface. It also contains an instance field of type **RoutePlanner**:

```
public class RoutePlannerProxy : IRoutePlanner
{
	private RoutePlanner? _planner;

	public RoutePlannerProxy()
	{
		_planner = null;
	}

	public PlanningResult CalculatePlan(PlanningData data)
	{
		InitPlanner();
		return _planner.CalculatePlan(data);
	}

	private void InitPlanner()
	{
		_planner = _planner ?? new RoutePlanner();
	}
}
```

Let's review the key features of **RoutePlannerProxy**:

- It has a (nullable) reference to a **RoutePlanner** object, BUT the reference is set to ***null*** in the constructor.
- It has a method **InitPlanner**, which reads *"if the instance field **_planner** is **null** when I'm called, it will be set to refer to a new **RoutePlanner** object, otherwise **_planner** is left unchanged"*
- It implements **CalculatePlan**, BUT does so by first calling **InitPlanner**, and then calling **CalculatePlan** on the **RoutePlanner** object

With this new class available, it will be possible to call the client code method **Pre-pareForPlanning** in a new way:

```
IRoutePlanner plannerProxy = new RoutePlannerProxy();
PrepareForPlanning(plannerProxy);
```

From the perspective of the client code, nothing has changed. The method **PrepareForPlanning** is still called with an object implementing the **IRoutePlanner** interface, and it will use this object just as before. The fact that the "proxied" **RoutePlanner** object is only created if needed (often known as **lazy creation**) is completely hidden.

As the code stands now, the client code will now have explicit knowledge about using a proxy class, since it becomes responsible for creating a proxy object. If you wish to isolate the proxy creation further, you could encapsulate the proxy creation in a factory class. It will then be a configuration issue whether or not to use a proxy; or you could even let the factory decide what to do at run-time (we assume that a class **Context** has been defined, which will provide decision information to the factory) :

```
public class RoutePlannerFactory
{
    public IRoutePlanner Create(Context c)
    {
        if (UseProxy(c))
        {
            return new RoutePlannerProxy();
        }
        else
        {
            return new RoutePlanner();
        }
    }

    private bool UseProxy(Context c)
    {
        // Decide whether to use Proxy,
        // based on provided context.
    }
}
```

The above example captures the essence of the **Proxy** pattern; whenever there are reasons to safeguard the creation of and/or access to an object, consider using a proxy class, which maintains a reference to the original object, and manages creation and/or use of the object. These reasons are typically grouped into a few main categories; for a specific object, reasons from more than one category may apply:

Object is expensive to create: Use a proxy to delay creation of the object until it is really needed. Such a proxy is also known as a **virtual proxy**.

Object is expensive to use: In the above example, it might also be very expensive to actually perform a route planning. A proxy could be used to cache the results of previous calculations, or maybe delegate the calculation to an approximation algorithm, which may not produce the optimal result, but will be much faster to complete its calculation. Another example of expensive usage occurs if the object being proxied runs remotely, and some sort of network communication is required in order to use it. A proxy can then act as a local representative of that object. Such a proxy is also known as a **remote proxy**.

Object must only be accessed by callers with certain permission. Instead of hard-coding such permission rules into the object itself, they can be implemented in a proxy class, which will then act as a gatekeeper with regards to access to the object. Such a proxy is also known as a **protective proxy**.

Additional bookkeeping is needed when an object is accessed. It may be interesting to obtain various statistical data on how the object is used, or it may be required to enforce exclusive access to the object, if it contains elements which are not thread-safe. Just as it was the case for access protection, we gain much more flexibility if we place such "extra" functionality in a proxy class, instead of hard-coding them into the object itself. In the above example, we could e.g. choose to keep the **RoutePlanner** class as clean as possible, in the sense that it should focus entirely on being able to calculate routes, and leave concerns like access protection, lazy creation etc. to proxy classes. Proxies of this nature are also known as **smart proxies**.

If you dive deeper into the world of Design Patterns, you may also come across a pattern known as the **Decorator** pattern. The difference between the **Decorator** pattern and the **Proxy** pattern is small and somewhat subtle. As we also saw in the example above, a **Proxy** class will usually be responsible for creating the proxied object itself, either directly or by use of a factory. For the **Decorator** pattern, the object-to-be-decorated will be provided to the **Decorator**, usually as a parameter to the **Decorator** constructor. The **Decorator** pattern is thus a bit more flexible than **Proxy**, since decorators and decorated objects can be combined more freely at run-time. However, the scope for using decorators is also smaller, since a decorator will usually have no control over when the decorated object is created. Examples as the above are therefore not good candidates for use of the **Decorator** pattern, whereas the **Proxy** pattern is quite useful.

Composite

The **Proxy** and **Adapter** patterns are "simple" structural patterns, in the sense that they deal with various ways of configuring access to a single object. The **Composite** pattern concerns ways of organising a set of objects which have a so-called **part-whole** relation. What is that more precisely? Consider for instance the problem of representing items that need to be shipped to customers. For the items as such, we can initially just use very simplistic classes such as **Shirt**, **Soda**, **Tomato** and so on, i.e. classes that represent specfic items that customers may order online, and which we then need to distribute to the customers. The company perhaps receives thousands of orders per day, and need to have a system in place for managing the distribution logistics. What will the model for such a system look like?

We can imagine that each customer will buy many different items, so we will need to represent a set of items somehow. We could start out by defining a base class for all items named **SimpleItemBase**, which could contain a couple of useful properties:

```
public class SimpleItemBase
{
	public double Price { get; }
	public double Weight { get; }

	public SimpleItemBase(double price, double weight)
	{
		Price = price;
		Weight = weight;
	}
}
```

Now we can represent a set of items as e.g. a list of type **List<SimpleItemBase>**. So far, so good. But this may not be enough to model items on a larger scale, though. We may also want to represent *sets-of-sets* of items, e.g. all the orders placed by customers in a certain region, since they may be served from the same distribution center. This adds a sort of hierarchical aspect to our data, and we will probably very soon come to the conclusion that we need a tree-like data structure in order to properly represent the items.

This is not as such particularly complicated to implement. One way could be to define a class **Goods**, which represents a collection of items that need to be distributed at a certain destination. The destination itself could then be anything from an entire region, a specific city, a specific street, etc., all the way down to a specific customer. An outline of such a class is given below:

```
public class Goods
{
	public string Destination { get; }
	public List<Goods> GoodsCollection { get; }
}
```

Notice the recursive nature of this definition, which is quite common whenever data has a recursive nature. Hopefully, the structure makes sense; a **Goods** object will contain several other **Goods** objects, which again may contain **Goods** objects, and so on. But at some point, we do end at the "atomic" items themselves, such as the **Tomato** and **Shirt** classes mentioned above. In order for this to work, **GoodsCollection** must then also be able to store such objects without an enclosing **Goods** object. So, should a class like **Tomato** then also inherit from **Goods**? This may at first seem a bit strange, but that is indeed where we are going. We should think of goods and simple items as things that form a *part-whole* relationship. Wherever we are using a **Goods** object, we should also be able to use a simple object, and vice versa…

In order for this to be possible, we need to define some sort of base class (or interface) that **Goods** objects and simple objects can inherit from. The **SimpleItemBase** class from above is actually a good candidate. Let's have a look at a version that only contains the properties:

```
public class SimpleItemBase
{
	public double Price { get; }
	public double Weight { get; }
}
```

Does it make sense to force a **Goods** class to implement e.g. a **Weight** property? Sure it does! That would probably be a very relevant property to have available for **Goods** objects for distribution planning purposes. In fact, we will "promote" the base class to an interface right away:

```
public interface IItem
{
	double Price { get; }
	double Weight { get; }
}
```

The name for the interface is a bit hard to get right; it should somehow encompass both simple objects and "composed" objects, i.e. objects consisting of other objects, like the **Goods** class. We will therefore simply call it **IComposable**:

```
public interface IComposable
{
	double Price { get; }
	double Weight { get; }
}
```

We can then let our original base class **SimpleItemBase** inherit from this interface. This does not have any significant effect on our simple classes, since they will just continue to inherit from the base class. A couple of very simplistic implementations follow below:

```
public class Beer : SimpleItemBase
{
	public Beer() : base(6, 0.33) { }
}

public class Shirt : SimpleItemBase
{
	public Shirt() : base(19, 0.6) { }
}
```

Things get a bit more interesting when we try to re-implement the **Goods** class. This class will inherit directly from the new interface. A first attempt follows below:

```
public class Goods : IComposable
{
	private List<IComposable> _goods;

	public Goods(List<IComposable> goods)
	{
		_goods = goods;
	}

	public double Price
	{
		// What do we do here…?
	}

	public double Weight
	{
		// What do we do here…?
	}
}
```

First of all, we have now achieved the goal of being able to mix **Goods** objects and simple objects inside a **Goods** object, since they all inherit from the **IComposable** interface. But what about the properties? The answer to how to implement these is

actually quite simple, if you just use your common sense. What is the weight of the items in a **Goods** object? Well, isn't it just the sum of the weights of the elements in the **_goods** list? If you can agree on that definition, the implementation becomes very simple indeed:

```
public double Weight
{
    get { return _children.Sum(i => i.Weight); }
}
```

That's it! The beauty lies in the recursive nature of the structure. Every element in **_goods** can figure out its own weight: if the object is a simple object, the weight is directly available in the object itself (see the definition of **SimpleItemBase** above). If the object is a **Goods** object, it just sums up the weight of its items, which will work no matter if the objects are **Goods** objects or simple objects!

This powerful idea is exactly what the **Composite** pattern is all about. We define a common interface which both simple (or "atomic") objects and composed objects inherit from, and let them implement the properties and/or methods in that interface in a meaningful way. For the atomic objects, the implementation will usually only involve the object itself, while the implementation for composed objects will usually involve some sort of dispatching to the children objects. This can be as simple as just accumulating a sum, as we just saw in the example, but can also involve more complex logic. But as long as all objects implement the same interface, we can implement that logic without considering the true nature of the objects involved.

The "full" version of the **Composite** pattern will often also involve a set of methods related to management of children objects. This could be methods such as **AddChild**, **RemoveChild**, a property for retrieving all children , etc.. While these methods are of course relevant in a complete implementation of the pattern, they are not as such needed in order to understand the core concepts of the pattern, and have therefore been left out. They are, however, usually quite straightforward to implement.

Flyweight

Imagine that you are creating a new social platform for children (we could call it *Club-Kitty*). As part of the signup process – which includes providing some simple text-and-numeric information like name, age, etc. – you must also select a visual "avatar". In order to keep things children-friendly, you can only choose between a limited set of pre-defined avatars, say 20 different avatars. The platform becomes an instant success, and one million users quickly sign up. However, your server application which runs the platform intrastructure now starts to experience problems…

Upon closer examination, it turns out that the server application has (at least) two major problems:

1. All profiles are kept in memory at all times.
2. The avatar image data is explicitly stored on each profile object.

It turns out that all profiles do need to be in memory, so we need to address the second problem. There seems to be an obvious problem: we store image data explicitly on each profile – i.e. a total of one million "instances" of image data are stored – but there are only 20 different avatar images to choose from… So, a lot of this data has to be redundant. An obvious fix could then be to e.g. store these 20 distinct images in a well-known location, which can then be referred to by e.g. a URL. The gist of this fix is correct, but perhaps also a bit fragile; what happens if e.g. the URLs change? Instead, we need a more general solution to this kind of problem. So, what characterises this kind of problem?

The main issue here is **object state**. That is, the data stored inside the object. The size of that data is in this scenario problematic, since we need to create a large number of these objects. In general, we can divide object state into two categories:

- State which is truly unique for each object. In this example, it could e.g. be user name, password, address information, etc..
- State which can potentially be shared among many objects. In this example, this is primarily the avatar image data.

The part which is truly unique is usually called **extrinsic state**, while the part which is potentially sharable is called **intrinsic state**. The main idea in the **Flyweight** patten is to separate these states into separate objects.

Our starting point will be the below class **Profile**. For each member in *ClubKitty*, a **Profile** object will be created (we have chosen a very simplistic implementation of the class, with only two properties):

```
public class Profile
{
	public Profile(string userName, int imageId)
	{
		UserName = userName;
		ImageData = ImageDB.Read(imageId);
	}

	public string UserName { get;}
	public int[] ImageData { get; } // Raw binary image data

	public void Display()
	{
		// Uses UserName and ImageData
	}
}
```

We have assumed that – given a numeric identifier for an avatar image – the avatar image data can be read from e.g. a database, through the method **ImageDB.Read**.

In the **Profile** class, the property **UserName** is the extrinsic state, while the property **ImageData** is the intrinsic state. First, we split up this class into two classes representing the extrinsic and intrinsic state, respectively:

```
public class ProfileExtrinsic
{
	public ProfileExtrinsic(string userName)
	{
		UserName = userName;
	}

	public string UserName { get; }
}
```

This class is quite simple, since it only carries the data representing the extrinsic state. The class for intrinsic state is slightly more complicated:

```
public class ProfileIntrinsic
{
	public ProfileIntrinsic(int[] imageData)
	{
		ImageData = imageData;
	}

	public int[] ImageData { get; }

	public void Display(ProfileExtrinsic extrinsicState)
	{
		// Uses both extrinsic and intrinsic state data,
		// but extrinsic state is now a parameter!
	}
}
```

This class carries the intrinsic state data and the existing functionality, i.e. the **Display** method. This method still needs both intrinsic and extrinsic state data, but the extrinsic state data must now be provided by the caller, as a parameter to **Display**.

This division enables a drastic reduction in data consumption. Let's assume that the profile data takes up one kilobyte for text-and-numeric data, and 200 kilobyte for avatar image data. In the original setup, this would cause a data consumption of

1,000,000 x 200 kb = 200 GB.

In the improved setup, we now get away with

1,000,000 x 1 kb + 20 x 200 kb = 1 GB + 4 Mb = 1 GB (approx.)

Indeed a significant reduction. The only remaining problem is how to put the pieces together in a way that still makes it relatively easy to work with profile objects. First, let's revise the **Profile** class:

```
public class Profile
{
	public Profile(
		ProfileExtrinsic extrinsicState,
		ProfileIntrinsic intrinsicState)
	{
		ExtrinsicState = extrinsicState;
		IntrinsicState = intrinsicState;
	}

	public ProfileExtrinsic ExtrinsicState { get; }
	public ProfileIntrinsic IntrinsicState { get; }

	public void Display()
	{
		IntrinsicState.Display(ExtrinsicState);
	}
}
```

You can perceive this class as a sort of “wrapper class” around a pair of extrinsic and intrinsic state objects. Also, let’s introduce a **ProfileFactory** class:

```
public class ProfileFactory
{
	private Dictionary<int, ProfileIntrinsic> _intrinsicStateData;

	public ProfileFactory()
	{
		_intrinsicStateData = new Dictionary<int, ProfileIntrinsic>();
	}

	public Profile Create(string userName, int imageId)
	{
		ProfileExtrinsic pe = new ProfileExtrinsic(userName);
		ProfileIntrinsic pi = null;

		if (_intrinsicStateData.ContainsKey(imageId))
		{
			pi = _intrinsicStateData[imageId];
		}
		else
		{
			pi = new ProfileIntrinsic(ImageDB.Read(imageId));
			_intrinsicStateData[imageId] = pi;
		}

		return new Profile(pe, pi);
	}
}
```

It is very important to notice the difference in how extrinsic and intrinsic state are handled. We always create a new object for extrinsic state (this was the "light" part of the state data), while we only create a new object for intrinsic state (which was the "heavy" part of the state data) if we really need to. The dictionary will act as a cache of previously created intrinsic state objects.

Instead of calling the **Profile** constructor directly, like:

```
Profile pA = new Profile("Alice", 14);
```

we now use the factory class for object creation instead:

```
ProfileFactory pFac = new ProfileFactory();
Profile pA = pFac.Create("Alice", 14);
```

You could even argue that the use of a factory class should have been introduced from the start, making it completely transparent that the underlying "infrastructure" has been changed.

So, this is an example of how to apply the ideas in the **Flyweight** pattern to a specific scenario. Are we close to having a generally applicable solution as well? If we take a closer look at the revised **Profile** class, it is actually not that far from being a candidate for a **Flyweight** base class. Below is a suggestion for such a base class:

```
public abstract class FlyweightBase<Tex, Tin>
{
	public Tex ExtrinsicState { get; }
	public Tin IntrinsicState { get; }

	protected FlyweightBase(Tex extrinsicState, Tin intrinsicState)
	{
		ExtrinsicState = extrinsicState;
		IntrinsicState = intrinsicState;
	}
}
```

We have removed the scenario-specific method **Display**, and type-parameterised the types for intrinsic and extrinsic state. **Profile** could then be implemented like this:

```
public class Profile : FlyweightBase<ProfileExtrinsic, ProfileIntrinsic>
{
	public Profile(ProfileExtrinsic exState, ProfileIntrinsic inState)
		: base(exState, inState)
	{
	}

	public void Display()
	{
		// Uses UserName and ImageData
	}
}
```

Pretty good. With a bit of work, we can to some extent generalise the factory class as well (this is left as a challenge for the reader ☺), and thus end up with a rudimentary framework for implementation of the **Flyweight** pattern.

The above considerations apply to scenarios where it is the size of each object that is problematic, not the sheer number of objects. Our solution will still produce a million **Profile** objects, but each object will be much lighter w.r.t. memory usage. Could we somehow avoid even creating these light-weight (fly-weight ☺) objects? You could imagine scenarios where it is possible to calculate extrinsic states rather than store them explicitly. Suppose you are simulating the behavior of a large group of animals, which behave according to certain rules. Given the starting conditions and the set of rules, it might be possible to calculate e.g. the position of animal **#N** at any time. If we need to use this (extrinsic) information to e.g. draw the animal on a map, we can feed that information directly into an object which only holds the intrinsic state (like the **ProfileIntrinsic** class above). With a bit of effort, we may even hide the fact that extrinsic states are calculated, such that clients of the code will use these “virtual” objects just as if they were real objects.

Design Patterns – Behavioral Patterns

The design patterns in this category are primarily concerned with the "dynamic" relationships between classes and objects. By "dynamic" is meant that it is not just the way classes and objects are wired together which is of interest here; it is in particular how these classes and objects collaborate in order to achieve a certain functionality. Just as we saw for structural patterns, we can divide these patterns into class-level and object-level patterns.

Behavioral patterns at the **class** level realise collaboration through **inheritance**, and by carefully distributing responsibilities between the involved classes. A key issue is often how to distribute responsibilities between base classes and derived classes. The relationships between the involved classes is thus established at compile-time, and cannot easily be changed at run-time.

Behavioral patterns at the **object** level realise collaboration through **composition**. Distribution of responsibilities is also a key issue here. Due to the use of composition, the relationships between the involved objects can in principle be changed at run-time, which add another dimension of flexibility to these pattern.

We will take a closer look at three behavioral design patterns: the **Template Method** pattern (which is a class-level pattern) and then the **Chain of Responsibility (CoR)** and **Strategy** patterns (which are object-level patterns).

Template Method

A very common scenario in software design occurs when we can specify a general algorithm for solving a general problem, but cannot "flesh out" all of the steps in the algorithm, because some steps need to be implemented individually for e.g. each type of data we wish to apply the algorithm to. A classic example is an algorithm for converting data to a text format. If we assume that a small interface **IDataSource** has been defined with operations for **Load** and **Save**, the algorithm could look like this:

```
public abstract class DataToTextConverter<T>
{
	// General algorithm for data-to-string conversion
	public List<string> ConvertDataToText(IDataSource<T> source)
	{
		// Read data from the data source
		List<T> data = source.Load();

		// This list will contain the converted data
		List<string> dataAsText = new List<string>();

		// For each data item: convert the item to a string,
		// and add that string to the dataAsText List
		foreach (T item in data)
		{
			string itemAsText = ConvertItem(item);
			dataAsText.Add(itemAsText);
		}

		return dataAsText;
	}

	// This method needs to be implemented in subclasses.
	public abstract string ConvertItem(T item);
}
```

This is a general algorithm for *data-to-text* conversion (if we assume that each data item can be converted to a single self-contained string). The only catch is the method **ConvertItem**. This method has no general implementation, since a specific conversion will depend on both the specific text format we wish to convert to, and on the data type **T** itself. With the base class in place, we can however fairly easily create a type-and-format specific derived class, for e.g. converting **Car** objects to XML:

```
public class CarToXMLConverter : DataToTextConverter<Car>
{
	public override string ConvertItem(Car item)
	{
		// Code for converting a Car object to XML
	}
}
```

That's it. We can now create and use a **Car**-to-XML specific converter object. The derived class has two important features:

- It contains specific code for converting a specific type of object into a specific string format.
- It does not contain code that defines (or modifies) the general algorithm for data conversion.

A derived class cannot in any way interfere with the general algorithm; it only fleshes out a specific step, as the specific circumstances dictate. This is precisely the intention of the **Template Method** pattern. The base class maintains absolute control over the general algorithm, while the derived class only gets called when the specific steps need to be carried out.

A Template Method – which in our example will refer to the top-level method **ConvertDataToText** – will thus call methods that may be defined in the base class itself, or may be defined as abstract, and thereby being defined in derived classes. The standard terminology used for describing these categories of methods follows below:

Concrete operations: These are methods or steps which are defined within the base class itself, and cannot be changed by a derived class. In our example, the line:

```
List<T> data = source.Load();
```

belongs to the group of concrete operations.

Primitive operations: These are the methods which are declared as abstract in the base class; in our example the **ConvertItem** method. These methods must be overrided in a derived class.

Hook operations: A hook operation is a method which does have an implementation in the base class, but it may be overrided in a derived class. In C# terminology, this will be a virtual method. We do not have any such methods in our example, but it could make sense to turn the **ConvertItem** method into a hook operation:

```
protected virtual string ConvertItem(T item)
{
      return item.ToString();
}
```

Deciding whether to go for an abstract or virtual approach will – of course – be highly dependent of the speficic scenario to which the pattern is applied.

The above example follows the classic approach to applying the **Template Method** pattern; create a base class containing the Template Method itself, and create derived classes containing definitions of the primitive and/or hook operations. Client code for using the classes could then look like this:

```
CarToXMLConverter carToXml = new CarToXMLConverter();
List<string> carsAsText = carToXml.ConvertDataToText(source);
```

We create an object of the derived type, and use it for conversion. However, you can also achieve the same effect without having to define derived classes. In this example, we are in practice creating a derived class in order to add in a definition of one single method. But we have learned previously that methods themselves can also be parameters to other methods! Consider the below version of **ConvertDataToText** (comments have been omitted for brevity):

```
public List<string> ConvertDataToText(IDataSource<T> source, Func<T,string> converter)
{
	List<T> data = source.Load();
	List<string> dataAsText = new List<string>();

	foreach (T item in data)
	{
		string itemAsText = converter(item);
		dataAsText.Add(itemAsText);
	}

	return dataAsText;
}
```

Suppose now that the client code contains the below method definition:

```
private string CarToXML(Car item)
{
	// Code for converting a Car object to XML
}
```

The client code for data conversion would then look like this:

```
DataToTextConverter<Car> carConv = new DataToTextConverter<Car>();
List<string> carsAsText = carConv.ConvertDataToText(source, CarToXML);
```

In this case, the client code creates an object of the base class type (which may now be a misleading description, since we no longer need to create derived classes…), and simply provides the single type-specific method definition which is missing, as a method parameter to **ConvertDataToText**.

This approach is definitely worth considering in this case, since we only need to provide a single method parameter. If the Template Method contains several primitive and/or hook operations, the benefit becomes more doubtful. Suppose the caller has to provide four methods parameters. It then becomes the responsibility of the caller to ensure that the provided method parameters are compatible with each other. One

method may perform part of the conversion, and perform it with XML as the target format. Another method may perform a different part, and the caller might accidentally provide a method which targets a different format! That will probably not end well. In that case, it is probably a safer approach to apply the **Template Method** pattern in the traditional way, to avoid unintended mixing of incompatible methods.

Chain of Responsibility

A pattern like **Template Mehtod** is usually targeted at scenarios where one particular class will perform (a variant of) a specific algorithm. Some steps may vary from class to class, but the template method defines the algorithm as a whole. This also implies that the complete algorithm for "handling" a certain request lies within that single method. Some algorithms do however have a nature that is not modeled particularly well by this approach. Consider for instance a help system for an application. The user can try to look up help on any part of the system in a context-sensitive way, e.g. by pressing the **F1** key on the keyword. The algorithm for looking up help could then look like this:

1. If detailed help is available for the specific part of the application the user is currently using, then display that help information. Otherwise:
2. If more general help is available for the general part of the application the user is currently using, then display that help information. Otherwise:
3. Open the web site for the product in a browser.

In short; we try to find help which is as detailed as possible, and if that action fails, we try to find help at a more general level. How will this strategy look in code? If we wish to express this logic in one single method **LookupAndDisplayHelp**, it could look like this (we assume that a number of more specific methods like e.g. **FindDetailedHelp** and **DisplayHelp** have been defined):

```
public void LookupAndDisplayHelp(Context c)
{
      HelpInfo info = FindDetailedHelp(c);

      if (info == null) // No detailed help found
      {
            info = FindGeneralHelp(c);
      }

      if (info != null) // General help found
      {
            DisplayHelp(info); // Display help info
      }
      else
      {
            OpenProductWebsite(); // Go to website
      }
  }
```

It's not super-complicated logic, but it is somewhat fragile to change. If we at some point introduce a medium-level help system, we need to update this code. Furthermore, the code has a static nature; we cannot alter the order of alternatives at runtime, if that should become necessary.

The **Chain of Responsibility** (**CoR**) pattern offers an alternative strategy for organising code of this nature. The central principle is to implement each handling alternative in a separate class, but let all classes inherit from the same interface. Objects of these classes can then be chained together to form such a **chain of responsibility**. That is, each object of these classes will contain a reference to another object implementing the same interface. If an object is not capable of handling the given request, it simply forwards the request to the object it references. At some point along this chain of handlers, someone will actually handle the request, and the forwarding of the request stops. The last object in the chain will not refer to any object, so this should be some sort of *if-all-else-fails* handler.

If we wish to apply this principle in the above example, we first need to define an interface for a "help handler". It looks like we only need a single method:

```
public interface IHelpHandler
{
    void LookupAndDisplayHelp(Context c);
}
```

Let's try to implement this method in a handler for detailed help. We could call a class corresponding to this level of handling for **HelpHandlerDetailed**:

```
// In class HelpHandlerDetailed
public void LookupAndDisplayHelp(Context c)
{
      HelpInfo info = FindDetailedHelp(c);
      if (info != null) // Help found!
      {
            DisplayHelp(info);
      }
      else
      {
            NextHandler?.LookupAndDisplayHelp(c);
      }
}
```

The logic is as described above; try to handle the request yourself, and forward the request if this proves impossible. As such, the code is pretty straightforward. However, the class itself contains a few additional elements worth discussing:

```
public class HelpHandlerDetailed : IHelpHandler
{
      public HelpHandlerDetailed(IHelpHandler nextHandler = null)
      {
            NextHandler = nextHandler;
      }

      public IHelpHandler NextHandler { get; set; }

      public void LookupAndDisplayHelp(Context c)
      {
            // As above
      }

      public HelpInfo FindDetailedHelp(Context c)
      {
            // Code for looking up detailed help
      }

      public void DisplayHelp(HelpInfo h)
      {
            // Show help on screen
      }
}
```

First, we note that the class contains a property **NextHandler**. The key features of this property are:

- It has the type **IHelpHandler**.
- It is set in the constructor, through a constructor parameter (i.e. using Dependency Injection). If the parameter is not explicitly specified, it defaults to **null**.
- Its value can be changed later, i.e. it is possible at run-time to change the handler to which the object refers.
- If the object cannot handle the request, it forwards the request to the handler referred to by **NextHandler** (unless it is **null**, in which case nothing happens).

All handler classes should follow this strategy, so there seems to be good reasons to introduce a handler base class, e.g. named **HelpHandlerBase**. The **NextHandler** property is an obvious candidate for being moved to the base class:

```
public class HelpHandlerBase : IHelpHandler
{
	public HelpHandlerBase(IHelpHandler nextHandler = null)
	{
		NextHandler = nextHandler;
	}

	public IHelpHandler NextHandler { get; set; }
}
```

Can we move additional code into the base class? The method **DisplayHelp** seems to be a general method for displaying help information, so it can likely also be moved to the base class. This leaves the method **FindDetailedHelp**, and the interface method **LookupAndDisplayHelp** itself. **FindDetailedHelp** seems to be quite specific for this level of help retrieval, so it probably needs to stay in the specialised class. But what about the interface method? Let's have a look at it again:

```
	// In class HelpHandlerDetailed
	public void LookupAndDisplayHelp(Context c)
	{
		HelpInfo info = FindDetailedHelp(c);
		if (info != null) // Help found!
		{
			DisplayHelp(info);
		}
		else
		{
			NextHandler?.LookupAndDisplayHelp(c);
		}
	}
```

How much of this code is general, and how much of it is specific? It is actually only the highlighted part which is specific! The rest of the code will look exactly the same if we implemented it in a class for general help handling, e.g. named **HelpHandler-General**:

```
// In class HelpHandlerGeneral
public void LookupAndDisplayHelp(Context c)
{
	HelpInfo info = FindGeneralHelp(c);
	if (info != null) // Help found!
	{
		DisplayHelp(info);
	}
	else
	{
		NextHandler?.LookupAndDisplayHelp(c);
	}
}
```

So, we have a general algorithm, with a single step which might vary under specific circumstances… doesn't that sound familiar? This is exactly the **Template Method** pattern, which we have just learned about! Applying the **Template Method** pattern here will give us this implementation of **LookupAndDisplayHelp** in the base class:

```
public void LookupAndDisplayHelp(Context c)
{
	HelpInfo info = FindHelp(c);
	if (info != null) // Help found!
	{
		DisplayHelp(info);
	}
	else
	{
		NextHandler?.LookupAndDisplayHelp(c);
	}
}

public abstract HelpInfo FindHelp(Context c);
```

All that remains is a fairly trivial implementation of **FindHelp** in the specialised classes, e.g. like this in **HelpHandlerDetailed**:

```
public override HelpInfo FindHelp(Context c)
{
	return FindDetailedHelp(c);
}
```

What about the *if-all-else-fails* class? Can we implement this class by using the base class? We can, if we are a little bit creative when implementing **FindHelp**:

```
// In class HelpHandlerNoHelpFound
public override HelpInfo FindHelp(Context c)
{
	OpenProductWebsite(); // Go to website
	return null;
}
```

This will force execution of the **else**-part in the base class implementation of **LookupAndDisplayHelp**, and since the **NextHandler** property must be null (otherwise, this would not be the *if-all-else-fails* handler…), the forwarding stops. If this feels like twisting the intention of the base class implementation of **LookupAndDisplayHelp** too much, you can still just implement **HelpHandlerNoHelpFound** by only inheriting from the interface **IHelpHandler**:

```
public class HelpHandlerNoHelpFound : IHelpHandler
{
	public void LookupAndDisplayHelp(Context c)
	{
		OpenProductWebsite();
	}

	public void OpenProductWebsite()
	{
		// Code for opening product website
	}
}
```

The requirement for participating in this particular Chain of Responsibility is (still) only that you implement the interface, not that you inherit from the base class. The base class is just a convenient implementation of the “typical” request handlers.

Proceeding in this way, we can create a set of request handler classes, which are unaware of each other w.r.t. specific types. Handlers only reference each other by the interface type. The last remaining issue is how to set up a specific chain of responsibility. This will be done in a part of the client code, probably a part dedicated to application configuration in general:

```
IHelpHandler noHH = new HelpHandlerNoHelpFound();
IHelpHandler generalHH = new HelpHandlerGeneral(noHH);
IHelpHandler detailedHH = new HelpHandlerDetailed(generalHH);
// …hand reference to detailedHH to relevant parts of the application
```

The above example creates a single chain of responsibility, which may be the correct approach in many cases. You could also imagine that separate classes for different types of detailed help handling were created, which would lead to a slightly more complex setup procedure, like this:

```
IHelpHandler noHH = new HelpHandlerNoHelpFound();
IHelpHandler generalHH = new HelpHandlerGeneral(noHH);
IHelpHandler createHH = new HelpHandlerCreate(generalHH);
IHelpHandler deleteHH = new HelpHandlerDelete(generalHH);
// …hand references to createHH and deleteHH
// to relevant parts of the application
```

This effectively creates two chains of responsibility; one starting at **createHH**, and another starting at **deleteHH**. Both of these detailed handler objects will forward a request to the (single) general handler object, which makes perfect sense. There is nothing in the pattern as such that forbids many handlers to refer to the same handler down the chain.

As the **Chain of Responsibility** pattern is described here, a request will only be handled by a single handler, i.e. the first handler which can handle the request properly. A useful variant of this could be to allow the handler to handle the request and forward the request down the chain. This could be relevant if several parties are involved in handling a single request. If we express the original handling principle in more general terms, it could look like this (**Handle** is an abstract method, which returns **true** if the request was successfully handled):

```
public void HandleRequest(Request r)
{
      if (!Handle(r))
      {
            NextHandler?.HandleRequest(r);
      }
}
```

Changing this to giving handlers down the chain the opportunity to the handle the request requires a very modest code change:

```
public void HandleRequest(Request r)
{
      Handle(r);
      NextHandler?.HandleRequest(r);
}
```

This can be compared with the ”rethrowing” strategy used in exception handling.

Strategy

The **Template Method** pattern described above is suited for scenarios where we can define a general behavior for some part of our application, but need to defer one or more specific steps to classes implementing a specific variation of these. The premise is thus that the behavior in question follows this general nature. We may, however, also face scenarios where these variations in behavior are so fundamental that no such template can be defined, except for a very high-level interface like this:

```
public interface IBehavior
{
	void Act();
}
```

Let's dive into a specific example: Suppose that a game contains a combat element, where players in the game can combat each other. Also, the game contains two concepts which will influence the way a player fights during combat:

Affinity: a religion-like concept. If a player has a certain affinity, the player can make use of various combat elements which are specific for that affinity.
Tactic: A tactic may be e.g. either offensive or defensive, and will also influence the way a player will conduct a fight.

In our example, we assume that three such affinities exist (named *sun*, *moon* and *stars*), along with the two tactics mentioned above (*offensive* and *defensive*)). In total, this yields six combinations of affinity and tactic, each resulting in a "strategy" for fighting that is unique for that combination. The looming question is now: how do we implement a **Player** class that can fight according to any of these six strategies? A first place to start could be to define an interface that captures the idea of two players fighting against each other:

```
public interface IFight
{
	void Fight(IPlayer self, IPlayer opponent);
}
```

This interface uses an **IPlayer** interface, which we will return to in a moment. The **IFight** interface can now be implemented by a number of concrete classes, each implementing a fight strategy for a specific affinity/tactic combination. That is, each implementation should define how the player **self** fights; the player **opponent** may of course use a different strategy.

```
public class SunFightDefensive : IFight
{
	public void Fight(IPlayer self, IPlayer opponent)
	{
		// Fight using Sun abilities,
		// in a defensive manner.
	}
}
```

In total, we will have to create six such classes. This may in itself sound repetitive, but remember that one of the premises of the example was that these strategies have a nature that makes it hard to define a common algorithm for the strategies. Also, the implementations may still be able to e.g. draw upon a shared library of useful methods, since the pattern does not make any assumptions about how these individual strategies are implemented.

With the specific strategy classes in place, we can focus on implementing a player class. Again we start with an interface:

```
public interface IPlayer
{
	string Name { get; }
	int HP { get; set; }
	Affinity CurrentAffinity { get; set; }
	FightTactic CurrentTactics { get; }
	void DoCombat(IPlayer opponent);
}
```

The types **Affinity** and **FightTactic** are assumed to be simple **enums** containing the valid values for affinity and tactics, respectively. With this interface in place, we can finally implement a **Player** class:

```
public class Player : IPlayer
{
	private int _initialHP;
	private int _currentHP;

	private IFight _sunDefStrat;
	private IFight _sunOffStrat;
	private IFight _moonDefStrat;
	private IFight _moonOffStrat;
	private IFight _starsDefStrat;
	private IFight _starsOffStrat;

	// Rest of class omitted for brevity
}
```

The **Player** class contains more than this – and we will return to some of these parts very soon – but hopefully you can begin to see where some of the challenges lie in creating a multiple-strategy implementation. The constructor of **Player** can now set up the references to **IFight** implementations properly:

```
public Player(string name, int hp, Affinity aff)
{
	Name = name;
	CurrentAffinity = aff;
	HP = hp;

	_sunDefStrat = new SunFightDefensive();
	_sunOffStrat = new SunFightOffensive();
	_moonDefStrat = new MoonFightDefensive();
	_moonOffStrat = new MoonFightOffensive();
	_starsDefStrat = new StarsFightDefensive();
	_starsOffStrat = new StarsFightOffensive();
}
```

The implementation of **DoCombat** can then look like this:

```
public void DoCombat(IPlayer opponent)
{
	if (  CurrentAffinity == Affinity.sun &&
		CurrentTactics == FightTactic.defensive)
	{
		_sunDefStrat.Fight(this, opponent);
	}

	// And so on, for all six combinations...
}
```

This code is functional, but it is clearly quite fragile to change. The fairly obvious drawbacks are:

- The class makes very explicit use of those six affinity/tactic combinations that currently exist, and will need to be updated if new affinities and/or tactics are added later on.
- The class is hard-coded to to use six specific implementations of **IFight**, and cannot be reconfigured to use a different “family” of implementations, if such a need should arise.

These drawbacks are definitely in conflict with – at least – the **Open/Close** principle. However, the fact that we have defined a common interface **IFight** for figthing tactics does make it relatively easy to improve the implementation.

A first shot at improving the implementation could be to drop the six individual instance fields for **IFight** implementations, and replace them with a more general system. Since the affinity/tactic combination is a "key" which should correspond to a specific **IFight** implementation, we can use a **Dictionary** for storing and retrieving such implementations:

```
public class Player : IPlayer
{
	private int _initialHP;
	private int _currentHP;

	private Dictionary<Tuple<Affinity, FightTactic>, IFight> _strategies;

	// Rest of class omitted for brevity
}
```

With this change, we can revise the implementation of **DoCombat**:

```
public void DoCombat(IPlayer opponent)
{
	Tuple<Affinity, FightTactic> key = new Tuple<Affinity, FightTactic>(
		CurrentAffinity, CurrentTactics);

	if (_strategies.ContainsKey(key))
	{
		throw new Exception(…);
	}

	_strategies[key].Fight(this, opponent);
}
```

The **Tuple** syntax is a bit long-winded, and could perhaps justify the definition of a class representing an affinity/tactic combination, but that is an implementation detail. In any case, the **DoCombat** method now has a much more robust implementation, and we have achieved a sort of method-level **Open/Close** compliance. This is in itself a significant improvement.

What remains is to consider how a **Player** object should establish dependencies to specific **IFight** implementations. A simple solution could just be to rewrite the constructor to create and add specific **IFight** implementations to **_strategies**, but that would retain the hard coupling between **Player** and these implementations. A better solution would be to "inject" **IFight** implementations into a **Player**. This could be part of the constructor implementation:

```
public Player(string name, int hp, Affinity aff,
              Dictionary<Tuple< Affinity, FightTactic>, IFight> strategies)
{
      _strategies = strategies;

      // Rest of constructor omitted for brevity
}
```

Fairly simple use of **Dependency Injection**, but it does take away the last bit of tight coupling between **Player** and specific **IFight** implementations.

The current implementation is now quite close to following the essence of the **Strategy** pattern. In its general formulation, the **Strategy** pattern involves:

1. An **IStrategy** interface (this is our **IFight** interface)
2. Specific implementations of the **IStrategy** interface (this is our six specific implementations of **IFight**)
3. A client which makes use of the **IStrategy** interface (this is **Player** in our case)

A last element in the **Strategy** pattern is the concept of a **context**. A context is simply the information a strategy needs to carry out its implementation. We have defined the **Fight** method in the **IFight** interface to take two arguments of type **IPlayer**, with the tacit assumption that an implementation of **IFight** needs to be able to access these objects, in order to properly implement a fighting strategy. The two **IPlayer** obejcts are thus the context for an **IFight** implementation. The fact that the role of "client" and "context" is played by the same object is not in contradiction with the pattern; it is actually a rather common setup when implementing this pattern.

The **Strategy** pattern is not very specific about the way dependencies are created between clients and strategy implementations. In the above, we ended up with a solution based on Dependency Injection, which is also a very common solution. If we want to introduce an extra layer of decoupling, we could introduce **Factory** classes into the mix. This could be useful if we want to be able to change the configuration of used **IFight** implementations at run-time. If we imagine that a **Player** could e.g. be affected by some sort of magic, which should result in a complete reconfiguration of fighting strategies, a set of new fighting strategies could then be retrieved from a factory object, for instance with a **Create** method taking an argument describing the magic the player is affected by. A **Player** object could then receive a reference to a factory object in its constructor.

Exercises

Exercise	**OOP3.0**
Solutions	NaiveRPG
Purpose	Try to find opportunities to apply some of the SOLID principles 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 SOLID 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	**OOP3.1**
Solutions	WeaponFactory
Purpose	Use the **Factory Method** design pattern in practice
Description	The **WeaponFactory** project contains: • Interfaces for weapon classes and weapon factory classes, in the folder **Interfaces**. This folder also contains the enumeration **WeaponType**. • A weapon base class **WeaponBase**, and several classes representing specific weapons, all in the folder **Weapons**. • Two (incomplete) factory classes **WeaponFactoryMedieval** and **WeaponFactoryFuture**, in the **Factories** folder.
Steps	1. Complete the implementation of the two factory classes. The intention of both classes is: given a weapon type (the parameter **type** to the **Create** method), create and return a weapon object which matches the weapon type and the "era" for the factory, i.e. medieval weapons for the **WeaponFactoryMedieval** factory, and futuristic weapons for the **WeaponFactoryFuture** factory. 2. Test your implementation by uncommenting the indicated lines of code in **Program.cs**. 3. This approach to weapon object generation is perhaps a bit fragile. What could go wrong (with regards to how the factory classes are implemented)? What could be improved?

Exercise	**OOP3.2**
Solutions	EnvironmentGenerator
Purpose	Use the **Abstract Factory** design pattern in practice
Description	The **EnvironmentGenerator** project can be seen as an extension of the **Weapon-Factory** project from the previous exercise. In addition to weapon objects, we now also want to generate building and creature objects, as part of a general environment generation algorithm. The **EnvironmentGenerator** project contains: • Interfaces for the three "environment element" types (weapon, building and creature) and corresponding interfaces for factory classes, all in the folder **Interfaces**. • Specific classes and factories corresponding to the medieval era, in the folder **ImplMedieval**, and likewise for the future era in **ImplFuture**. • An **Environment** folder (more details below)
Steps	1. Take a quick cruise through the classes, enumerations and interfaces in the three folders **Interfaces**, **ImplMedieval** and **ImplFuture**. They are all quite simple. Do however pay some attention to the grouping of classes. How is it ensured that e.g. all medieval factories can only produce objects belonging to the medieval era? (Hint: does **Create** take any parameters?). 2. In the **Environment** folder, take a look at the **IEnvironmentGenerator** interface, and the **EnvironmentGeneratorBase** class. They are intended to define the interface and implementation of an Abstract Factory. There is however a problem in the **EnvironmentGeneratorBase** class… See if you can spot the problem (Hint: the objects generated by an Abstract Factory should belong to the same "family" of objects). 3. Take a look at the **RunExisting** method in **Program.cs**. Can you now see what the problem from 2) is? If not, try to run the application, and check the output again. 4. Open the (empty) class **EnvironmentGeneratorFuture**. This class should implement the **IEnvironmentGenerator** interface, but in a way that ensures that all objects created by the factory belong to the future era. Implement the class, and implement **EnvironmentGeneratorMedieval** in a similar way. 5. Modify the **Create** method in the **EnvironmentGeneratorFactory** class, such that it returns the correct environment factory objects. 6. Uncomment the code lines in the **RunImproved** method in **Program.cs**, run the application, and see the effect ☺.

Exercise	**OOP3.3**
Solutions	DataAccess
Purpose	Use the **Adapter** design pattern in practice
Description	The project models a scenario where a legacy database access tool needs to be used still, but through a different interface. The **DataAccess** project contains: • Two simple domain classes **Car** and **Customer**, plus an interface **IHasKey**, in the folder **DomainClasses**. • A database access tool named **DBTool**, which can access a database (simulated by the class **Database**). Both are found in the **DBToolClasses** folder. • The **DataSource** folder, which contains the **IDataSource** interface. This is the interface the adapter class must implement. The folder contains some additional classes, which we return to below.
Steps	1. Take a look at the domain classes **Car** and **Customer**. What is the most important difference between those two classes? 2. Take a look at the **Database** class. It is not so important to understand it in details, since it is accessed through the existing **DBTool** class. 3. Take a more thorough look at the **DBTool** class. The most important things to note are: a. What public methods and properties does **DBTool** contain? b. What parameters do the methods require? What is returned? c. What do they do (see the comments for each method/property) 4. Run the application – it contains a small test of the existing **DBTool** class. Take note of the number of objects printed after each operation. 5. Now open the (incomplete) **DBToolAdapter** class. This class is supposed to become an adapter for **DBTool**. That is, it must implement all methods and properties in the **IDataSource** interface, by using the methods and properties available on the **DBTool** class. 6. Implement all methods and properties in the **DBToolAdapter** class. Some implementations are quite simple (i.e. a single line of code), while others will require a bit more work. 7. Once you have finished the implementation, open **Program.cs**. Uncomment the first set of code lines, i.e. the lines testing the **DBToolAdapter** class. Run the application; hopefully, the test should produce the same results as the test from step 4). 8. **[Extra]** We now also want to be able to use (an adapted) **DBTool** with domain classes that do NOT implement **IHasKey**. Explore the classes **KeyAdapter** and **DataSourceAdapter**, and see if you can make that happen. A test for such an adapter is also found in **Program.cs**, using the **Car** class as the adapted domain class.

Exercise	**OOP3.4**
Solutions	CalculationProxy
Purpose	Use the **Proxy** design pattern in practice
Description	This project tries to simulate a complex calculation, which has the following features: • It takes an integer x- and y-coordinate as input (in the project, x and y are per default integer numbers between 1 and 5, both included). • A calculation takes between 300ms and 700ms, which is simulated by a call to **Thread.Sleep** in the calculation method. • It always returns the same result for a given x- and y-coordinate It could therefore be beneficial not to calculate the same result over and over, but rather save a calculated result in a "cache", which can then be used to look up results rather than calculating them again.
Steps	1. Take a look at the **ICalculate** interface in the **Common** folder. It is quite minimal. You can also peek into the **Coordinate** class (it essentially contains an X and a Y coordinate pair). 2. Take a look at the **CalculatorFactory** class in the **Common** folder. Note how it returns either a **Calculator** object or a **CalculatorProxy** object. How come that it is possible to return either of these objects? What do they have in common? 3. Take a look at the **Calculator** class in the **ActualCalc** folder. What method must be implemented by this class? 4. Take a thorough look at the (incomplete) **CalculatorProxy** class in the **CalcByProxy** folder. What method must be implemented by this class? What does this class and the **Calculator** class have in common? How are they different? 5. The method **Calculate** in **CalculatorProxy** currently just calls **Calculate** on the proxied calculator object. If you run the application, you will see that the choice of strategy doesn't make any difference. This must be fixed… 6. Implement the **Calculate** method in **CalculatorProxy** properly, by following the description given in the comments to the method. This may also require that you peek into the **Cache** class. 7. Re-run the application, and (hopefully) observe the positive effect of using the "calculation-by-proxy" strategy.

Exercise	**OOP3.5**
Solutions	Fight1v1
Purpose	Use the **Template Method** design pattern in practice
Description	The **Fight1v1** project contains: • An interface for 1v1 fight management, and an enumeration for specific fight types, in the folder **Interfaces**. • A general implementation of 1v1 fight management, in the class **Fight1v1-Manager**, and two (incomplete) classes **Fight1v1ManagerFair** and **Fight-1v1ManagerBiasedA**, both derived from **Fight1v1Manager**. These classes are found in the folder **FightManagers**. • A factory class **Fight1v1ManagerFactory.**
Steps	1. Take a look at the enumeration and interface in the **Interfaces** folder. They are both quite simple. 2. Take a look at the **Player** class. It is as such not at the center of attention in this exercise, but make sure you understand its general functionality. 3. Take a thorough look at the class **Fight1v1Manager** in the **FightManagers** folder. This is the central class in the project. Make sure you understand the roles of the methods **Fight**, **SingleFight** and **ExchangeBlows**. 4. In the **Template Method** design pattern terminology, what kind of method is **Fight**? What kind of operation is **ExchangeBlows**? 5. The method **ExchangeBlows** is abstract in **Fight1v1Manager**. Where do you suppose it will be implemented? 6. Open the classes **Fight1v1ManagerFair** and **Fight1v1ManagerBiasedA**. How are they related to **Fight1v1Manager**? 7. Implement the **ExchangeBlows** method in both of the classes above, and run the application to see how the fighting plays out in both cases (a test which tests both cases is already implemented in **Program.cs**). Feel free to adjust the setting for the fighters (in **Program.cs**), to see how it affects the fighting. 8. Review your implementation of **ExchangeBlows** in the two derived classes. Could both strategies be implemented in a single class, if we are allowed to add more parameters to the **ExchangeBlows** method?

Exercise	**OOP3.6**
Solutions	SupportManagement
Purpose	Use the **Chain of Responsibility** (CoR) design pattern in practice
Description	A fictitious IT Support Center receives error tickets (reports on errors in some IT system), that can be at one of four levels of criticality: *Light*, *Moderate*, *Severe* or *Catastrophic*. Furthermore, an error ticket may be written in either Danish or English. An error ticket can be handled by one of four support departments: • **Local Support**: Can handle *Light* error tickets in Danish or English. • **National Support**: Can handle *Moderate* error tickets in Danish or English. • **Regional Support**: Can handle *Severe* error tickets written in English. • **World Support**: Can handle *Catastrophic* error tickets written in English. The Support Center also contains a Translation Service, which can translate an error ticket from Danish to English. However, translation is considered expensive, so translation is only done if strictly necessary. In practice, this means that error tickets at level *Severe* or *Catastrophic* are translated, if they are written in Danish. The **SupportManagement** project implements a model of the Support Center's strategy for processing of error tickets. The project contains: • The folder **Error**, which contains the class **ErrorTicket** plus a couple of related enumerations. • The folder **SupportCommon**, which contains two interfaces **ISupport-Action** and **ISupportCenter**, plus corresponding base classes. • The folder **SupportOriginal**, which contains the class **SupportCenter-Original** (implements **ISupportCenter**), plus classes representing the four support departments plus the translation service. • The folder **SupportCoR** (CoR: *Chain of Responsibility*), which we return to later.
Steps	1. Start out in the **SupportCommon** folder. Study the two interfaces and the corresponding base class implementations. They define the general structure of a Support Center. Notice the methods **HandleOpenTickets** and **TryHandleTicket**. Which design pattern are these methods implementing? 2. Move on to the **SupportOriginal** folder, and take a quick tour of the four **…Support** classes. What do they have in common? Also take a look at the **TranslatorService** class. How is this class different from the **…Support** classes? 3. Take a thorough look at the class **SupportCenterOriginal**, in particular the method **TryHandleTicket**. Why must this class implement this method? What do you think of this method in terms of flexibility? Can we change the way it works at run-time?

	4. Run the application – it will run a small test of **SupportCenterOriginal**. The test creates 10 error tickets, and tries to process them. The test should show that all 10 tickets end up in the list of Closed tickets. 5. Move on to the folder **SupportCenterCoR**. First, go into the subfolder **SupportHandler**, and study the interface **ISupportHandler**. It defines the idea of a "support handler", which either handles an error ticket itself, or passes the ticket on to another handler. Move on to the **SupportHandler-Base** class. Make sure you understand how the methods from the interface are implemented. Can you recognise the design pattern being used? 6. Still in the **SupportHandler** subfolder, study the class **SupportHandler-Aggregation**. Can you figure out how this makes it possible to use the existing support classes from the **SupportOriginal** folder? 7. Move on to the **Adapters** subfolder. The idea is to consider translation and the *if-all-else-fails* handling from the **SupportCenterOriginal** implementation as support handlers as well. These two operations must therefore be "adapted" to fit the **ISupportHandler** interface. Study the two **…Adapter** classes, to see how this adaptation is performed. 8. The final step is to build a *Chain of Responsibility* for a Support Center. In the **Config** folder, take a look at the **ISupportCoRConfiguration** interface. Then take a look at the **SupportCoRConfigurationDefault** class. The CoR built in **SetupSupportCoR** is currently very short – it only contains a single handler. Run the application, and observe that the test of **SupportCenter-CoR** shows that all 10 error tickets end up in the list of Unhandled tickets. 9. Change the implementation of **SetupSupportCoR**, such that the resulting CoR defines the same support strategy as used in **SupportCenterOriginal**. You can use the test to see if you are on the right track. Hint: The complete CoR should contain a total of six handler objects. 10. Compare this implementation of support strategy with the original implementation. Can you change the support strategy without changing any code in **SupportCenterCoR**? Where is the decision about using a specific strategy taken?

Exercise	**OOP3.7**
Solutions	CompositeShapes
Purpose	Work with the **Composite** pattern in a relatively simple scenario.
Description	The **Composite** pattern is used for modeling *part-whole* relationships between objects. This fits very well with the realm of geometric shapes. A shape can be a "simple" shape like a Point, Circle, etc.., but can also be a "composed" shape, consisting of a set of other shapes, themselves being either simple or composed. The project **CompositeShapes** contains a single interface **IShape**, plus a small set of classes implementing that interface.
Steps	1. Take a look at the **IShape** interface, and at the four classes in the **Shapes** folder that implement it. Both the interface and the classes are all rather simple, even though some of the implementations of specific methods and properties may require you to recall a bit of geometry calculus ☺. However, it is not the geometric correctness as such of the methods that is the primary concern here. 2. Take a look at **Program.cs**. Here we create and use a few simple shapes. The test itself should also be fairly easy to comprehend. Again, do not spend too much time pondering if the result are geometrically correct. 3. To warm up, we now want to add a so-called **PolyLine** shape to the system. A **PolyLine** is a line consisting of connected line segments, which in practice means that a **PolyLine** is simply defined by a set of **Point** objects. Implement the **PolyLine** class; remember that **PolyLine** must implement the **IShape** interface, so you have to figure out how to implement all methods and properties in that interface for **PolyLine**. Once done, remember to write some code that tests **PolyLine** as well. 4. Next, we want to be able to define and use composite shapes, i.e. shapes consisting of other shapes (that may themselves be composite shapes…). Try to implement this using the **Composite** pattern. More specifically, you could define a new class **CompositeShape** that can contain a set of shape objects, but also implements **IShape**. You can probably use your implementation of **PolyLine** as a very useful starting point, since the implementation of **CompositeShape** will likely be quite similar. Again, remember to test your code, preferably with a couple of more complex shapes. 5. **[Extra]** Now that you have defined a **CompositeShape** class, do you then really need the **PolyLine** class? Or could you maybe re-implement **PolyLine** using **CompositeShape**? How do we then ensure that such an object will only contain **Point** objects?

Exercise	**OOP3.8**
Solutions	NPCAnimationManager
Purpose	Work with the **Flyweight** pattern in a relatively simple scenario.
Description	Computer games and movies involving CGI (Computer-Generated Imagery) may sometimes involve scenarios where thousands of participants are involved – e.g. in large battle scenes – and therefore need to be animated and visually rendered. We will denote such participants as **NPC**s (Non-Player Characters), even though this term usually has a slightly different meaning. The data needed for animating and rendering a single NPC may in itself be of considerable size, and it can therefore easily become problematic if we store all this animation data directly on each NPC, even though this may make perfect sense from a modeling point of view. This is a situation where the **Flyweight** pattern can be put to good use.
Steps	1. Take a look at all the types, interfaces and classes in the folder **Common**. They form the basis for implementing NPC-related functionality. There are quite a few files, but each one is relatively simple. 2. Now take a look at the two classes **NPC** and **NPCFactory** in the **Original** folder. They represent a fairly straightforward implementation of creation and representation of NPCs. 3. Open **Program.cs**, and take a look at the **TestOriginal** method. Make sure you understand what it does. 4. Run the application *as-is* (the constant **noOfNPCs** in **Program.cs** should be set to 5,000). Before closing the application, make sure to open the **Diagnostic Tool** window (found under **Debug | Windows…**). Under **Process Memory**, you should see that the application uses around 500 MB of memory. Try setting **noOfNPCs** to 10,000, and run the application again… 5. Clearly, a lot of memory is used by each **NPC** object. Take another look at how **NPC** is implemented, and see if you can figure out why each object uses about 0.1 MB. Is it really needed…? (Hint: are we duplicating some data over and over…?) 6. We are going to apply the **Flyweight** pattern to try to fix this. Create a new folder **Flyweight**, and create two new classes **NPCExState** and **NPCInState**. **NPCExState** should contain the non-shared part of the NPC state, i.e. all those properties which are truly specific for each NPC. **NPCInState** should contain that part of the state which can be shared. 7. Now create a third class named **NPCFlyweight**. This class should contain one property of type **NPCInState**, plus a re-implementation of **Animate** that takes a parameter of type **NPCExState**.

	8. Finally, create a fourth class named **NPCFlyweightFactory**. This class should contain methods for creating **NPCFlyweight** objects and for creating **NPCExState** objects. **NB**: Note that this is where we need to be smart! We should only create truly new **NPCFlyweight** objects if we really need to (hint: how many different **NPCFlyweight** objects do we really need to create, given that such an object only contains the intrinsic state?) 9. With all this in place, you can write a test for **NPCFlyweight** similar to **Test-Original** in **Program.cs**. Note, however, that you will probably need to implement a couple of helper methods as well, since you cannot directly use the existing helper methods like **CreateNPCs** and **AnimateAll** (why not…?). Once you have implemented a test, it should – hopefully – show that you can now create a lot more object than before (you should be able to create and use millions of **NPCExState** objects). 10. **[EXTRA]** The steps above should illustrate how the **Flyweight** pattern can be applied in practice. However, some of our existing test code did not "survive", in the sense that it is not compatible with the new classes. See if you can figure out how to create a revised implementation of **NPC**, that still uses the principles from the **Flyweight** pattern, but does so in a transparent way. As a minimum, the revised class should implement **INPC**, and it should still be possible to create millions of these revised NPC objects.

Exercise	**OOP3.9**
Solutions	AnimalBehavior
Purpose	Work with the **Strategy** pattern. Develop a solution from a very small, interface-based starting point.
Description	In this setting, we assume that we can model animal behavior in this way: 1. All animals have an internal state, which can be either *aggressive*, *fearful* or *idle*. 2. The state can be changed externally. 3. An animal will exhibit a behavior which reflects its current state. 4. The behaviors for each state need not have anything in common, i.e. they can follow completely individual patterns. The project **AnimalBehavior** will be used for implementing this model. The project is initially very small, consisting only of a couple of interfaces in the **Interfaces** folder, plus some test code in **Program.cs**.
Steps	1. Study the interfaces in the **Interfaces** folder (you can for now skip over the **IAnimalBehaviorFactory** interface). The purpose for each interface should hopefully be clear from the comments for each interface. 2. Now study the code in **Program.cs**. The test it implements is quite simple: create an animal object, and invoke its behaviors several times. The state of the animal is (possibly) updated in each iteration of the loop. 3. **Your job** is now to create a suitable implementation of the **IAnimal** class. The implementation should strive at following the **Strategy** pattern, which in broad terms implies that your implementation should: a. Store a set of behaviors internally (these behaviors will be provided through the **SetBehavior** method). b. Exhibit a behavior corresponding to the current state (the behavior will be invoked when calling the **Act** method) c. NOT include a big ***if-else*** statement used for selecting the behavior to invoke… because that would break the **Open/Close** principle! 4. Once you have created an implementation, you can simply update the assignment statement in **Program.cs**, and run the application. You should now see how your animal implementation acts. 5. **[Extra]** maybe the **IAnimalBehaviorFactory** interface could be useful for a more structured way of creating animal objects?