


*This handbook is designed for students of the DVA222 course and aims to introduce the fundamentals of object-oriented programming and design, rather than focusing on the latest features of C#. By clicking the  symbol located at the top-right corner of an example, you access an online IDE where you can edit and test the code to enhance your understanding.*



# Contents

<b>I</b>	<b>Object Oriented Programming</b>	<b>3</b>
<b>1</b>	<b>Classes &amp; Objects</b>	<b>3</b>
1.1	Access Modifiers . . . . .	5
<b>2</b>	<b>Encapsulation &amp; Information Hiding</b>	<b>5</b>
<b>3</b>	<b>Inheritance &amp; Composition</b>	<b>10</b>
<b>4</b>	<b>Abstraction</b>	<b>13</b>
4.1	Polymorphism . . . . .	13
4.2	Abstract Classes & Interfaces . . . . .	18
4.2.1	Abstract Classes . . . . .	18
4.2.2	Interfaces . . . . .	20
4.3	Generics . . . . .	22
4.4	Delegates . . . . .	25
<b>5</b>	<b>Odds &amp; Ends</b>	<b>27</b>
5.1	The Object Superclass . . . . .	27
5.2	Reference Types & Value Types . . . . .	29
5.3	The Structure Type . . . . .	31
5.4	The Enumeration Type . . . . .	32
5.5	More Modifiers . . . . .	32
5.6	Exceptions . . . . .	35
5.7	Properties . . . . .	37
5.8	Tuples . . . . .	39
5.9	Events . . . . .	39
<b>II</b>	<b>Object Oriented Design</b>	<b>41</b>

<b>6</b>	<b>UML Class Notation: the Basics</b>	<b>41</b>
<b>7</b>	<b>Cohesion &amp; Coupling</b>	<b>45</b>
<b>8</b>	<b>The S.O.L.I.D. Principles</b>	<b>46</b>
8.1	Single Responsibility Principle . . . . .	47
8.2	Open Closed Principle . . . . .	49
8.3	Liskov Substitution Principle . . . . .	51
8.4	Interface Segregation Principle . . . . .	53
8.5	Dependency Inversion Principle . . . . .	54
<b>9</b>	<b>Design Patterns</b>	<b>56</b>
9.1	Iterator . . . . .	57
9.2	Observer . . . . .	61
9.3	Visitor . . . . .	63
9.4	Builder . . . . .	68
9.5	Adapter . . . . .	70
9.6	Factory Method . . . . .	71
9.7	Strategy . . . . .	73
9.8	Chain of Responsibility . . . . .	75
9.9	Singleton . . . . .	76
9.10	Prototype . . . . .	77
9.11	Flyweight . . . . .	79

## Part I

# Object Oriented Programming

Object Oriented Programming (OOP) is just one of the viable solutions that have been proposed to address the *software crisis* discussed during a NATO Software Engineering Conference held in 1968 in Germany [1]. Even if OOP facilitates the design of complex software systems, remember that OOP is not an universal remedy and it has its own pitfalls.

While the procedural software approach focuses on the flow of functions to be followed (e.g., a C program just to be understood), OOP is a programming paradigm where the software is designed around units, so-called *classes*, that aggregate data and the functions accessing such data. OOP is based on some fundamental principles which are encapsulation, inheritance, and abstraction. With the help of these core principles, it becomes easier to create modular, reusable, and maintainable software. The rest of this part is organized as it follows:

- ✚ A class is the blueprint for defining a data structure and implementing its behavior, whereas an object is the instance of a class (Section 1).
- ✚ Encapsulation allows an object to separate its interface from its implementation. The data and the implementation code for the object is hidden behind its interface (Section 2).
- ✚ Inheritance is useful for improving code organization allowing to reuse the existing code by reducing repetitions (Section 3)
- ✚ Abstraction consists of different techniques (e.g., polymorphism, abstract classes and interfaces, generics,...) that make the code more flexible and easier to maintain (Section 4).

## 1 Classes & Objects

In OOP, a program is made of a set of interactive *objects* that are instances of the classes modeling the state and the behavior of such objects. When the program starts, the class defining the `Main` function is instantiated that in turn creates a number of objects that start to cooperate. Often, classes and objects are used as synonyms, but a class is a blueprint that objects implement: you need a class before you can create an object.

When defining a class, it's mainly required to specify its *members*: *attributes* (or *fields*) and *methods* (or *functions*). The attributes define the characteristics of the class that jointly capture all the information about the entity to represent. Attributes should be protected by their enclosing class to reduce the number of members visible from outside the class. An attribute may be a built-in value type ■ (e.g., `int` or `string`) or a reference (a memory pointer in practice) to another class. Methods define the

actions that a class can perform and how the class attributes are accessed to read or update their values.

In C# you can define a new class by means of the keyword `class` and instantiate it by means of the `new` operator. Briefly, the `new` operator allocates a block of memory to store the new object which is then initialized. The initialization phase is implemented by a dedicated method (so-called *constructor*) which has the same name as the class and it is declared with no return type. In case no constructor has been defined for the class, an implicit parameterless *default constructor* initializes all the class attributes with their default value (e.g., zero for the numeric ones). Once the new instance is ready, the `new` operator returns a reference which can be used to access the public members of the new object using the dot-notation. Example:

```
class Rectangle {
    double Width, Height;
    public Rectangle(double w, double h) { // constructor
        Width = w;
        Height = h;
    }
    public void Scale(double factor) {
        Width *= factor;
        Height *= factor;
    }
    public double Area() {
        return Width*Height;
    }
}

class Program {
    static void Main() {
        Rectangle r = new Rectangle(4, 6);
        Console.WriteLine("Area={0}", r.Area());
        r.Scale(0.5);
        Console.WriteLine("Area={0}", r.Area());
        //r.Width = 3;
        //Console.WriteLine("Width={0}", r.Width);
    }
}
```

In the example, the attributes `Width` and `Height` of the class `Rectangle` cannot be accessed directly as with a `struct` in C, but exclusively through the public methods provided by the class that are: `Scale`, `Area`, and the constructor `Rectangle`. You can actually have direct access to a class attribute if it is declared `public`, but we aim to avoid this in general. Try it by yourself forking the example and

uncommenting the last two lines of the function `Main`: the program will not compile due to the violation of the protection level of the attribute.

## 1.1 Access Modifiers

As it will be discussed in Section 2, there are several good reasons to keep as many class members as possible hidden from the other objects. To this end, *access modifiers* are used to specify the protection level of the class members and their visibility. For the moment we introduce just two of them:

- 👉 A *public* member is visible from outside the class hence can be “used” by any other object (e.g., the methods `Scale` and `Area` in the example of Section 1).
- 👉 A *private* member is only visible inside the class where is defined (like the attributes `Width` and `Height` in the example of Section 1). To emphasize the member hiding aspect of OOP, members with no access modifier are considered private.

More modifiers will be introduced in Section 3 and Section 5.5.

## 2 Encapsulation & Information Hiding


Encapsulation and information hiding are two important principles in OOP that contribute to design and implement robust and modular software which in turn improves the code maintenance efficiency and reduces dependencies.

**Encapsulation.** It is the concept of enclosing data and implementing the functions that operate on such data into a single unit. Purpose:

- 👉 *Access Control*: it allows the control of access to the internal details of a class. It defines which parts of the class are accessible from the outside and which are not. This helps in protecting the integrity of the data and ensuring that it is modified only in a controlled manner.
- 👉 *Modularity*: by encapsulating related data and methods within a class, you create a modular unit that can be easily understood, modified, and extended. This promotes code organization and maintenance.

**Information Hiding.** It is a concept relying on encapsulation and involves restricting the visibility of certain details within a class, preventing the access to some internal implementation details from outside of the class. Purpose:

- 🔖 *Abstraction*: it helps in creating a clear abstraction by exposing only what is necessary while hiding the rest. When instantiating a class, programmers do not need to know how the internal methods work: they only need to know how to use the public interface.
- 🔖 *Reduced Dependency*: by hiding implementation details, you reduce the dependencies between different parts of the program. This makes it easier to change the internal implementation of a class without affecting the external code that uses it.

The two next examples present two implementations of the queue data structure  that are successively compared: the first one is written in C while the second is written in C# and makes use of encapsulation and information hiding. *Why did we not use the same programming language for both implementations?* C and C# refer to two different programming paradigms that are *procedural programming* and *object oriented programming*, respectively. Each paradigm has its own set of principles and methodologies. Procedural programming basically consists of organizing a program into a group of functions. Important data can be placed as global or they may be accessed by any function which, in turn, makes data more vulnerable to an inadvertent modification. For example, in case a data structure needs to be revised, to identify the whole set of functions accessing a specific data becomes very hard.

Here is the C implementation consisting of 3 files `main.c`, `queue.c`, and `queue.h`:

```
/* file: main.c */

#include "queue.h"
#include <stdio.h>
#include <stdlib.h>

int main()
{
    struct queue_t queue = create_queue(10);
    enqueue(&queue, 3);
    enqueue(&queue, 18);
    enqueue(&queue, 24);
    dequeue(&queue);
    enqueue(&queue, 36);
    for(int i=0; i!=queue.size; ++i)
        printf("%d\n", queue.data[(queue.tail+i)%queue.capacity]);
    dispose_queue(&queue);
    return EXIT_SUCCESS;
}

/* file: queue.h */
```

```
#ifndef __QUEUE__
#define __QUEUE__

struct queue_t {
    int *data;
    int head, tail, capacity, size;
};

struct queue_t create_queue(int capacity);
void enqueue(struct queue_t *queue, int value);
int dequeue(struct queue_t *queue);
struct queue_t dispose_queue(struct queue_t *queue);

#endif

/* file: queue.c */

#include "queue.h"
#include <stdlib.h>
#include <assert.h>

struct queue_t create_queue(int capacity) {
    assert(capacity>0);
    struct queue_t queue;
    queue.data = (int*)malloc(sizeof(int)*capacity);
    assert(queue.data);
    queue.capacity = capacity;
    queue.head = 0;
    queue.tail = 0;
    queue.size = 0;
    return queue;
}

void enqueue(struct queue_t *queue, int value) {
    assert(queue!=NULL && queue->capacity!=queue->size);
    queue->data[queue->head] = value;
    queue->head = (queue->head+1)%queue->capacity;
    queue->size += 1;
}

int dequeue(struct queue_t *queue) {
    assert(queue!=NULL && queue->size>0);
    int i = queue->tail;
```



```

    queue->tail = (queue->tail+1)%queue->capacity;
    queue->size -= 1;
    return queue->data[i];
}

int size_queue(const struct queue_t *queue) {
    assert(queue!=NULL);
    return queue->size;
}

struct queue_t dispose_queue(struct queue_t *queue) {
    free(queue->data);
    queue->data = NULL;
    queue->size = 0;
    queue->capacity = 0;
    queue->head = 0;
    queue->tail = 0;
}

```

The C version above presents some of the aforementioned issues. For example, a *client* of `queue_t` (i.e., another block of code making use of such a data structure, e.g., the `main.c` file) is allowed to define some new ad-hoc functions that extend the set declared in `queue.h` and possibly create some misuses in the logic of the data structure by compromising, in this way, the integrity of the program. Furthermore, if the queue implementation changes (e.g., replacing the array with a linked list), all the “extra” implemented functionalities and the `for` statement in the `main` function need to be changed accordingly. To remove some of these flaws, OOP ties the data and the functions that operate on them into objects that protect data from accidental external modification by establishing how to access them: this brings us back to the encapsulation and information hiding concepts. Last but not the least, the programmers in charge of coding the client function need to the internal functioning of `queue_t` to be able to write the `for` loop which is beyond their purview.

Here is a possible C# implementation of the queue:

```

class MyQueue : IEnumerable {
    public MyQueue(int capacity) {
        if(capacity<1)
            throw new Exception("capacity cannot be less than 1");
        data = new int[capacity];
        this.capacity = capacity;
        this.size = 0;
        this.head = 0;
        this.tail = 0;
    }
}

```

```
public void Enqueue(int value) {
    if(capacity==size)
        throw new Exception("queue is full");
    data[head] = value;
    head = (head+1)%capacity;
    size += 1;
}

public int Dequeue() {
    if(size==0)
        throw new Exception("queue is empty");
    int i = tail;
    tail = (tail+1)%capacity;
    size -= 1;
    return data[i];
}

public IEnumerator GetEnumerator() {
    for(int i=0; i!=size; ++i)
        yield return data[(tail+i)%capacity];
}

private int[] data;
private int size;
private int capacity;
private int head, tail;
}

class Program {
    static void Main() {
        MyQueue queue = new MyQueue(10);
        queue.Enqueue(3);
        queue.Enqueue(18);
        queue.Enqueue(24);
        queue.Dequeue();
        queue.Enqueue(36);
        foreach(var elem in queue)
            Console.WriteLine(elem);
    }
}
```

In this case, the only way to interact with the queue is using the public methods defined by the class. Furthermore, by implementing the `GetEnumerator` method, as required by the interface `IEnumerable` that the class implements (more about this in Section 4.2.2 and Section 9.1), the implementation also provides the client with a default way to iterate over its elements (i.e., by means of the `foreach` statement) whatever the internal implementation is: it's responsibility of `MyQueue` to

decide whether to include it in its public interface and, in case, how to implement it.

Mind that this does not preclude the client from extending the behavior of `MyQueue`. In general, this is possible, e.g., by using *inheritance* which is presented in Section 3.

### 3 Inheritance & Composition

Another key feature of OOP is *inheritance*: a mechanism that allows a new class to inherit (part of) the definition of an existing class. Inheritance promotes code reuse and extensibility by creating a hierarchical structure among classes. Here are some terms related to inheritance:

- 📌 **Base Class or superclass or parent class:** The class whose members are inherited by another class.
- 📌 **Derived Class or subclass or child class:** The class that inherits properties and behaviors from another class. In most OOP languages, a derived class can access only public or *protected* members of the base class (protected is a further access modifier in addition to those presented in Section 1.1) while the private members of the base class are not accessible from the derived classes.

However, if not carefully used, inheritance has the potential to break encapsulation and information hiding as in the following example, decrease the readability of the code by spreading the code related to a class over several class definitions, or produce unexpected behaviors as it is also described in Section 8.3:

```
class Rectangle {
    public Rectangle(double w, double h) {
        Width = w;
        Height = h;
    }
    public double Area() {
        return Width*Height;
    }
    protected double Width, Height;
}

class RotatedRectangle : Rectangle {
    public RotatedRectangle(double w, double h, double a) : base(w, h) {
        Angle = a;
    }
    public void Scale(double factor) {
```

```
        Width *= factor;
        Height *= factor;
    }
    double Angle;
}

class Program {
    static void Main() {
        RotatedRectangle r = new RotatedRectangle(4, 6, 90);
        r.Scale(0.5);
        Console.WriteLine("Area={0}", r.Area());
    }
}
```

In the above example, the attributes `Width` and `Height` are protected in the base class, and they cannot be modified once a `Rectangle` has been instantiated. However, this becomes possible for the subclass `RotatedRectangle` by means of the method `Scale`. Last but not the least, deep class hierarchies lower the encapsulation leading to maintenance challenges.

Some further notes about inheritance and C#:

- 🔑 Some languages, like C++, support multiple inheritance which means that a class can extend multiple classes at the same time, while in other languages, like C#, a child class can have only one parent class. In languages that support multiple inheritance, be aware of the diamond problem occurring when a class inherits from two classes that have a common ancestor because this can lead to ambiguity and maintenance challenges.
- 🔑 Only the default constructor (the one with no parameters) of the base class is invoked automatically when a subclass instance is created. To make use of a parameterized constructor of the superclass, you need to use the `base` keyword (as it is done in the for the constructors of the subclass `RotatedRectangle`).
- 🔑 The `sealed` modifier applied to a class prevents other classes from inheriting it.

As an alternative to inheritance, we can opt for *composition*. With composition, we build complex objects by composing them of simpler ones. Example:

```
class Engine {
    public Engine(double hp) {
        HorsePower = hp;
    }
    public override string ToString() {
        return HorsePower.ToString("0.0")+"hp";
    }
}
```

```
    }  
    double HorsePower;  
}  
  
class ElectricMotor : Engine {  
    public ElectricMotor(double hp, int v) : base(hp) {  
        Voltage = v;  
    }  
    public override string ToString() {  
        return Voltage.ToString()+" Volt, "+base.ToString();  
    }  
    int Voltage;  
}  
  
class CombustionEngine : Engine {  
    public CombustionEngine(double hp, int nc) : base(hp) {  
        NoCylinders = nc;  
    }  
    public override string ToString() {  
        return NoCylinders.ToString()+" cylinders, "+base.ToString();  
    }  
    int NoCylinders;  
}  
  
class Car {  
    public Car(Engine e, int np) {  
        engine = e;  
        noPassengers = np;  
    }  
    public override string ToString() {  
        return "engine ["+engine.ToString()+"], "+  
            noPassengers.ToString()+" passengers";  
    }  
    int noPassengers;  
    Engine engine;  
}  
  
class Program {  
    static void Main() {  
        Car aCar = new Car(new Engine(110), 2);  
        Car Tesla = new Car(new ElectricMotor(325, 400), 5);  
        Car Panda = new Car(new CombustionEngine(75, 4), 4);  
        Console.WriteLine("generic car: "+aCar);  
    }  
}
```

```
        Console.WriteLine("Tesla: "+Tesla);  
        Console.WriteLine("Panda: "+Panda);  
    }  
}
```

The above example shows both inheritance and composition. In fact, both `ElectricMotor` and `CombustionEngine` are subclasses of the superclass `Engine` (i.e., inheritance), whereas `Car` includes an instance of `Engine` among its attributes (i.e., composition).

The decision to use inheritance or composition depends on the specific design requirements of your application. Here are two relationships that can help you to choose the most appropriate option:

- 🔗 **“Has-a” Relationship.** We can opt for *composition* when there is a “has-a” relationship for one object containing or using another object or, even an easier case, more than one. In the previous example, a `Car` **has** an `Engine`.
- 🔗 **“Is-a” Relationship.** Use inheritance when there is a clear “is-a” relationship between the subclass and superclass. In the previous example, the `Engine` class might have subclasses like `ElectricMotor` and `CombustionEngine` because these **are** two types of engines.

## 4 Abstraction

One of the core features of OOP is *abstraction*. Similarly to information hiding seen in Section 2, the different techniques that realize an abstraction aim to expose the minimal set of the features of a component to the outside to make it simpler to break down large and complex problems into smaller and independent pieces that are more understandable and easy-to-reuse.

A programming language that features polymorphism allows developers to access objects of different types through the same type (Section 4.1). Abstract classes and interfaces define an incomplete blueprint acting as backbone for concrete classes that share it (Section 4.2). Generics are a facility to parameterize our code over types (Section 4.3) while delegates allow us to do the same over methods with the same signature (Section 4.4).

### 4.1 Polymorphism

*Polymorphism*<sup>1</sup> is another important feature of OOP and it relies on inheritance. It allows to refer the instance of a subclass by means of a reference declared as the superclass type. For example, in the program at page 11 [🔗](#), the attribute `engine` is capable to hold an instance of `Engine`, which is its native type, but also `ElectricMotor` or `CombustionEngine` given that they inherit from `Engine`.

---

<sup>1</sup> Polymorphism comes from the Greek words *polys* meaning many and *morphe* meaning form.

Indeed, polymorphism can reduce the amount of the programmers' work who can initially create a base class with all the attributes and methods they envision for it. More specific subclasses with certain unique attributes and behaviors can be added successively by leaving the existing implementations unvaried.

But before illustrating polymorphism in more detail, we introduce the *method binding* mechanism which is responsible for choosing which method-implementation to use in response to a method-call. There are two options to resolve it that rely on the *static type* and *dynamic type* of an object: the first one denotes the type used to **declare** the object, while the second refers to the type used to **instantiate** the object. Example:

```
class Parent {
    virtual public string Foo() { return "Parent.Foo()"; }
}


class Child : Parent {
    public override string Foo() { return "Child.Foo()"; }
}

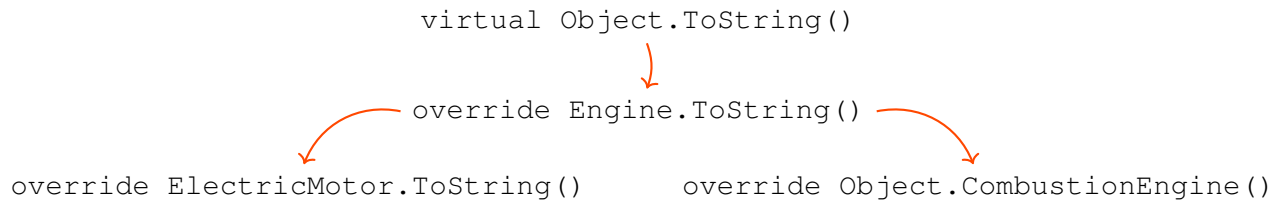
class AnotherChild : Parent {
    public new string Foo() { return "AnotherChild.Foo()"; }
}

class Program {
    static void Main() {
        Parent obj = new Child();
        Console.WriteLine(obj.Foo());
        obj = new AnotherChild();
        Console.WriteLine(obj.Foo());
    }
}
```

In the example above, the static type of the reference `obj` is `Parent` while its dynamic type initially is `Child` and then it becomes `AnotherChild`: which implementations will be used in response to the two calls to the method `Foo`? The answer is discussed in the next two sections.

## Method Overriding

The redefinition of `ToString` in the program at page 11  is an example of *runtime polymorphism* also called *method overriding*. `ToString` is a method declared as `virtual` in the `Object` class, from which the `Engine` class implicitly inherits (more about this in Section 5.1). The method is then overridden by the three different classes of the example:



The peculiarity of runtime polymorphism is that it applies *dynamic binding*. This means that the method binding is resolved when the program is already running according to dynamic type of the object rather than its static type. In the example, the static type of the attribute `engine` in `Car` is `Engine` while the implementation actually used in response to the call `engine.ToString()` depends on the type of the object held by `engine` that is: the implementation given in the subclass `ElectricMotor` for `Tesla` and the implementation given in `CombustionEngine` for `Panda`.

### Method Hiding (or Shadowing)

The alternative to method overriding is *method hiding* (or *shadowing*) which consists of providing a different implementation of a method in the subclass that hides the one in the superclass, but in a polymorphic manner. In fact, this approach differs from the previous one since the method binding is resolved at compile time and the implementation selected in response to a method call is based on the static type (from which the name *static binding*) that is fixed, rather than the dynamic type. To let a method defined in the subclass hide the one in the base class, its declaration must be preceded by the keyword `new` regardless of the fact if the hidden method is declared `virtual` or not in the parent class.

Here is a compact example to summarize how the two techniques work:

```

class Parent {
    public virtual void Foo() => Console.WriteLine("Parent.Foo");
    public void Bar() => Console.WriteLine("Parent.Bar");
}

class Child : Parent {
    public override void Foo() => Console.WriteLine("Child.Foo");
    public new void Bar() => Console.WriteLine("Child.Bar");
}

class Program {
    static void Main() {
        Parent Pa = new Parent();
        Child Ch = new Child();
        Pa.Foo(); //output: Parent.Foo
    }
}
  
```



```
    Pa.Bar(); //output: Parent.Bar
    Ch.Foo(); //output: Child.Foo
    Ch.Bar(); //output: Child.Bar
    Pa = Ch;  //using polymorphism
    Pa.Foo(); //output: Child.Bar
    Pa.Bar(); //output: Parent.Bar
}
}
```

In the example above:

- ➦ After assigning `Ch` to `Pa`, the output of the call `Pa.Foo()` changes since `Foo` makes use of the runtime polymorphism and the type of the object that `Pa` refers to is now `Child`. For the method `Bar` the implementation used remains the one defined in the superclass since in this case we make use of shadowing, but the static type of `Pa` is still `Parent`.
- ➦ The method `Bar` in the `Child` class hides the `Parent` implementation of the homonym method.
- ➦ Removing the keyword `new` from the definition of `Bar` in the `Child` class, the compiler shows a warning asking you to “use the `new` keyword if hiding was intended”.

A final note about these two techniques regards their performance and how dynamic binding is implemented under the hood. Briefly, any class containing at least a virtual method defines a table storing the pointers to such methods (the so-called *virtual method table* or *vtable* for short). A pointer to that table is then included in each new instance of that class which is therefore capable of accessing the methods of its native type. Furthermore, when a class with virtual methods is specialized, the child class inherits a copy of the vtable and replaces the entries for the virtual methods that overrides. That said, with the dynamic binding, we pay the overhead for performing a lookup operation on the vtable at runtime to access the called-method implementation. Differently, the static binding is resolved at compile time so that there is no such overhead.

## Method Overloading

A different type of polymorphism is *method overloading* that refers to the possibility of defining multiple methods with the same name, but different list of input-parameter types (the returned type of the method is not considered for this purpose). C# offers the possibility to overload even binary and unary operators (like `+`, `-`, `*`, etc.) for the user-defined types (more details can be found here [■](#)). Because the compiler decides which method to call at compile time based on the method signature, method overloading is classified as *static polymorphism* (or *compile time polymorphism*).

In the following example, the operator `+`, the constructor `Clock`, and the method `Add` have been overloaded. Note that it is possible for one implementation to rely on another one through the keyword

this for the constructor or by calling another version of the overloaded method (note that this is not recursion since the two methods are distinct):

```
class Clock {
    public Clock(int h, int m) {
        hour = h;
        minute = m;
    }
    public Clock(int h, int m, int s) : this(h,m) {
        second = s;
    }
    public void Add(int m) {
        int q = (minute+m)/60;
        minute = (minute+m)%60;
        hour = (hour+q)%24;
    }
    public void Add(int m, int s) {
        int q = (second+s)/60;
        second = (second+s)%60;
        Add(m+q);
    }
    public override string ToString() {
        return hour.ToString("00")+":"+minute.ToString("00")+":"+second.ToString("00");
    }
    public static Clock operator + (Clock c, int seconds) {
        int s = c.second+seconds;
        int m = c.minute+s/60;
        int h = c.hour+m/60;
        return new Clock(h%24,m%60,s%60);
    }
    int hour, minute, second;
}

class Program {
    static void Main() {
        Clock c = new Clock(10, 30);
        Console.WriteLine("It's "+c.ToString());
        c.Add(40); // add 40 minutes
        Console.WriteLine("It's "+c.ToString());
        c = new Clock(22, 30, 45);
        Console.WriteLine("It's "+c.ToString());
    }
}
```

```
c.Add(120,30); // add 120 minutes and 30 seconds
Console.WriteLine("It's "+c.ToString());
Clock c2 = c+135; // add 135 seconds to c
Console.WriteLine("It's "+c2.ToString());
}
}
```

## 4.2 Abstract Classes & Interfaces

With inheritance, developers can design a fully implemented and instantiable class then specialize it by defining a child class. This section introduces the concept of “incomplete implementation” which gives us the possibility to gather the commonalities among different classes in the same entity without the obligation of providing a detailed implementation of all its parts.

### 4.2.1 Abstract Classes

An abstract class is a class that cannot be instantiated which means that it must be inherited by a derived class to be used. Abstract classes are created using the `abstract` keyword and can contain any number of non-abstract methods, attributes, and at least one abstract method. Example:

```
abstract class Polygon {
    public abstract double Area();
    public abstract double Perim();
    protected int NoSides;
    public override string ToString() =>
        "Area="+Area()+" Perim="+Perim()+" #Sides="+NoSides;
}

class Hexagon : Polygon {
    public Hexagon(double s) { Side = s; NoSides = 6; }
    public override double Area() { return 3.0*Math.Sqrt(3)*Side; }
    public override double Perim() { return NoSides*Side; }
    public override string ToString() =>
        "SideLen="+Side+" "+base.ToString();
    double Side;
}

abstract class Quadrilateral : Polygon {
    protected Quadrilateral() { NoSides = 4; }
}
```

```

class Square : Quadrilateral {
    public Square(double s) { Side = s; }
    public override double Area() { return Side*Side; }
    public override double Perim() { return NoSides*Side; }
    public override string ToString() =>
        "Size="+Side+"x"+Side+" "+base.ToString();
    double Side;
}

class Rectangle : Quadrilateral {
    public Rectangle(double w, double h) { Width = w; Height = h; }
    public override double Area() { return Height*Width; }
    public override double Perim() { return Height*2+Width*2; }
    public override string ToString() =>
        "Size="+Height+"x"+Width+" "+base.ToString();
    double Height, Width;
}

class Program {
    static void Main() {
        Polygon[] shapes = new Polygon[3];
        shapes[0] = new Square(3);
        shapes[1] = new Rectangle(4,5);
        shapes[2] = new Hexagon(2);
        foreach(var s in shapes)
            Console.WriteLine(s.GetType()+" : "+s);
    }
}

```

It is worth noting that:

- 👉 You need the `override` keyword to implement an abstract method.
- 👉 You can inherit an abstract class from another abstract class, e.g., `Quadrilateral` is inherited from `RegularPolygon`. In these cases, you can override an abstract method or leave this to its child classes.
- 👉 Abstract classes can have a constructor even if they cannot be instantiated: this is useful for centralizing the initialization of some common attributes in one method (e.g., the constructor of `Quadrilateral` is used to initialize `NoSides` which is the same for squares and rectangles).
- 👉 An abstract class is a superclass and as such it can be used as type for grouping objects belonging to different subclasses of it, e.g., the array `shapes`. In this case the only methods accessible for

the grouped elements are those defined in the abstract class.

- 🔗 An abstract class is a superclass and as such it allows to use the keyword `base` to access the superclass members, e.g., for overriding the method `ToString`.

#### 4.2.2 Interfaces

To give you an idea of what interfaces are, you can think of them as abstract classes without attributes: only public methods (and suchlike, see properties in Section 9.10). Differently from abstract classes, a class can implement multiple interfaces, and this overcomes the lack of multiple inheritance like other languages have, e.g., C++ (see notes in Section 3). Similarly to class polymorphism, even interfaces can be used for gathering a set of different-type objects that implement the interface: in these cases, only the methods defined in the interface can be invoked on the objects. Interfaces are also useful to realize more flexible classes with methods relying on a specific interface rather than requiring the instance of an object type (more about this in Section 8.5). Here an example:

```
// note that interface names usually start with 'I'
interface IShape {
    double Area();
}

interface IPolygon {
    int GetEdgeNumber();
    double GetEdgeSize();
}

class Circle : IShape {
    public Circle(double r) { Radius = r; }
    public double Area() => Radius * Radius * Math.PI;
    double Radius;
}

class Square : IPolygon, IShape {
    public Square(double e) { EdgeSize = e; }
    public double Area() { return EdgeSize * EdgeSize; }
    public int GetEdgeNumber() { return 4; }
    public double GetEdgeSize() { return EdgeSize; }
    double EdgeSize;
}

class EqTriangle : IPolygon, IShape {
    public EqTriangle(double b, double h) {
```

```

        Base = b; Height = h;
    }
    public double Area() { return Base * Height / 2; }
    public int GetEdgeNumber() { return 3; }
    public double GetEdgeSize() { return Base; }
    double Base, Height;
}

class Program {
    // it accepts whatever object implementing the interface
    static double PerimeterCalculator(IPolygon p) {
        return p.GetEdgeNumber() * p.GetEdgeSize();
    }
    static void Main(string[] args) {
        var shapes = new List<IShape>(); // a list of IShape
        shapes.Add(new Circle(6));
        shapes.Add(new Square(7));
        double AreaSum = 0;
        foreach(var s in shapes)
            AreaSum += s.Area();
        Console.WriteLine(AreaSum);
        Console.WriteLine(PerimeterCalculator(new Square(14)));
        Console.WriteLine(PerimeterCalculator(new EqTriangle(2, 5)));
    }
}

```

Here is a resuming table of the main differences between abstract classes and interfaces:

Abstract Class	Interface
It may contain declaration and implementation parts.	It contains only the declaration of methods.
It can contain constructor.	It does not contain constructor.
It can contain different types of access modifiers like public, private, protected, ...	It only contains public access modifier because everything in the interface is public.
It is used to implement the core identity of class.	It is used to implement abilities of class.
A class can inherits from one abstract class.	A class can implement multiple interface.
Abstract class can contain methods and attributes.	Interface can only contains methods and similar members.

### 4.3 Generics

A different form of abstraction available for several OOP languages consists in separating a class (or method) from the data type it is working on. In C#, this goes under the name of *generics* which let programmers design classes and methods that defer the type specification until the class (or method) is instantiated (or invoked) in the code. For example, think to the `MyQueue` class defined in Section 2: independently from the datatype stored, the set of operations that we intend to perform on the queue is always the same, hence it would be more practical and maintainable to keep one definition of the data-structure and instantiate it for any specific type when it's needed. In the following program, we instantiate a queue for integers and one for strings starting from the same class definition which is implemented by referring the generic type parameter `T`. When the client needs to instantiate a queue with a specific type, it is done by specifying the target type between the angular parenthesis:

```
class MyQueue<T> : IEnumerable<T> {
    public MyQueue(int capacity) {
        if(capacity<1)
            throw new Exception("capacity cannot be less than 1");
        data = new T[capacity];
        this.capacity = capacity;
        this.size = 0;
        this.head = 0;
        this.tail = 0;
    }
    public void Enqueue(T value) {
        if(capacity==size)
            throw new Exception("queue is full");
        data[head] = value;
        head = (head+1)%capacity;
        size += 1;
    }
    public T Dequeue() {
        if(size==0)
            throw new Exception("queue is empty");
        int i = tail;
        tail = (tail+1)%capacity;
        size -= 1;
        return data[i];
    }
    public IEnumerator<T> GetEnumerator() {
        for(int i=0; i!=size; ++i)
            yield return data[(tail+i)%capacity];
    }
}
```

```

IEnumerator IEnumerable.GetEnumerator() {
    return this.GetEnumerator();
}

private T[] data;
private int size;
private int capacity;
private int head, tail;
}

class Program {
    static void Main() {
        var QueueInt = new MyQueue<int>(5);
        QueueInt.Enqueue(3);
        QueueInt.Enqueue(18);
        foreach(var elem in QueueInt)
            Console.WriteLine(elem);
        var QueueString = new MyQueue<string>(8);
        QueueString.Enqueue("hello");
        QueueString.Enqueue("world");
        foreach(var elem in QueueString)
            Console.WriteLine(elem);
    }
}

```

Note the following aspects:

- ✚ An apparently equivalent implementation would make use of the superclass `Object` in the implementation since all `C#` classes inherit from it (see Section 5.1). However, the generics implementation has the advantage of maintaining compile-time type checking (avoiding, e.g., to add different-type items in the same queue) as well as it does not require useless casts (i.e., from `Object` to the intended type).
- ✚ When moving to the generics implementation, it's convenient to abstract even the interface by passing to `IEnumerable<T>` which extends the old `IEnumerable`. In this way, the `foreach` statement will return a sequence of items of type `T` instead of `Object` instances. This requires that the class implements two methods having the same signature but from distinct interfaces, i.e., `GetEnumerator` and `IEnumerable.GetEnumerator`, where the first one is the one used by default while the second is kept private (more details are here [🔗](#)).

Another interesting feature about generics is the possibility to define one or more constraints over the actual type assigned to the parametric type `T`. In the following example, we define a generic sorting algorithm which requires that `T` is pairwise comparable for obvious reasons. To specify such a



requirement, we add where `T: IComparable<T>` to the method definition. This makes mandatory for the type replacing `T` to implement the interface `IComparable<T>` so that `BubbleSort<T>` can freely use the method `CompareTo` defined in the required interface:

```
class Program
{
    static void BubbleSort<T> (T [] arr) where T: IComparable<T> {
        for (int i = 0; i < arr.Length-1; i++)
            for (int j = 0; j < arr.Length-i-1; j++)
                if ( arr[j].CompareTo(arr[j+1]) > 0 ) {
                    T tmp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = tmp;
                }
    }

    public static void Main() {
        int[] values = { 15, 47, 13, 92, 18 };
        BubbleSort(values);
        foreach(var elem in values)
            Console.WriteLine(elem);
        char[] letters = { 'a', 'z', 'c', 'h' };
        BubbleSort(letters);
        foreach(var elem in letters)
            Console.WriteLine(elem);
    }
}
```

This also means that if you want to use a method like `BubbleSort<T>` on your own classes then they must implement the aforementioned interface and, optionally, override the operators '`<`' and '`>`' as it has been done in the following example:

```
class Product : IComparable<Product> {
    public Product(string name, int price) {
        this.name = name;
        this.price = price;
    }

    public int CompareTo(Product other) => price-other.price;
    public override string ToString() => name;
    public static bool operator < (Product left, Product right)
        => left.CompareTo(right) < 0;
    public static bool operator > (Product left, Product right)
        => left.CompareTo(right) > 0;
}
```

```

    public readonly string name;
    public readonly int price;
}

class Program {
    public static void Main() {
        Product a = new Product("apple", 12);
        Product b = new Product("bread", 30);
        if (a>b) Console.WriteLine(a+" costs more than "+b);
        if (a<b) Console.WriteLine(a+" costs less than "+b);
    }
}

```

Note that the class `Product` implements the interface `Comparable<Product>` using itself as parametric type since we aim to compare two objects of that class. You can try to combine the two previous examples to sort a `Product` array by price.

## 4.4 Delegates

The purpose of *delegates* in `C#` is the same as function pointers in `C` with the advantage of being type safe. In fact, a delegate is a new proper type of object used to bind any method with a predefined signature: the parameter-types list and return type. Delegates are typically used for passing methods as arguments to other methods to promote code reusability and improve flexibility. Through a delegate, you can call the bound method with the same syntax of a method call or using the method `Invoke` belonging to the `Delegate` class:

```

// RealFunction is a new delegate-type
public delegate double RealFunction(double x);

class LineEq {
    public LineEq(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public double Calculate(double x) => a*x+b;
    double a, b;
}

class Program {

    public static double Parabola1(double x) { return 2*x*x+5*x-6; }
}

```

```

static void Main() {
    //f is a delegate of type RealFunction
    RealFunction f;
    //we can assign f to an anonymous method defined inline:
    f = delegate (double x) { return Double.NaN; };
    Console.WriteLine("f(9.0)={0}={1}", f(9.0), f.Invoke(9.0));
    //we can assign a static function:
    f = Parabolal;
    Console.WriteLine("f(1.0)={0}={1}", f(1.0), f.Invoke(1.0));
    //we can assign the method of an object:
    var l = new LineEq(10, 5);
    f = l.Calculate;
    Console.WriteLine("f(5.0)={0}={1}", f(5.0), f.Invoke(5.0));
}
}

```

Here `f` is an object of type `RealFunction` which is the delegate type declared at the beginning of the example. Delegates are also able to hold more than one method. In particular, methods can be added and removed by means the operators `+=` and `-=` so that, all the registered methods will be invoked when the delegate is called. Example:

```

class Program {

    delegate void PrintRealFunctionResult(double x);

    static void Half(double x) => Console.WriteLine($"{x}/2={x/2}");
    static void Double(double x) => Console.WriteLine($"{x}*2={x*2}");

    static void Main(string[] args) {
        PrintRealFunctionResult f = Half;
        f(3);
        f += Double;
        f(6);
        f -= Half;
        f(9);
        f -= Double;
        f?.Invoke(1);
    }
}

```

In case a delegate object is invoked when no method is linked, the runtime support throws the exception `System.NullReferenceException` (more about exceptions in Section 5.6). To avoid this,

we can use the null-conditional operator `'?.'` which applies the right-hand operation (e.g., `Invoke`) only if the left-hand operand evaluates to non-null by also ensuring that the operand stays the same after being verified<sup>2</sup>.

## 5 Odds & Ends

This section presents some relevant features of `C#` that are not directly related to the OOP principles.

### 5.1 The Object Superclass

Every class in `C#` is directly or indirectly derived from the `Object` class. Here are some of the methods defined in the class `Object` that can be overridden except for `ReferenceEquals` which must be kept the same:

- ✚ `Equals` returns true if and only if the two compared objects are evaluated equal. The default implementation of this method just compares objects by reference so that two objects are considered equal if they have the same address in memory. So, if you want to compare two objects based on their state, you must override such a method and the next one accordingly.
- ✚ `GetHashCode` generates an integer value (so-called *hash code*) calculated on the basis of the object state. Mind that overriding `Equals` requires to override `GetHashCode` as well because it is assumed that two equal objects return the same hash-code while the opposite may occur (the so-called *collisions* 🟠 shown in the next example). It's up to the programmers to provide a good hash function that minimizes the number of possible collisions.
- ✚ `GetType()` returns the type of the current object.
- ✚ `MemberwiseClone` creates a shallow copy of the current object (see also Section 9.10).
- ✚ `ReferenceEquals` returns true if and only if two objects correspond to the same instance in memory and it has nothing to do with the state of the objects unlike the `Equals` method. It is almost useless for value types since the value is directly stored inside the variable hence each of them has a different memory position.
- ✚ `ToString` returns a string that represents the current object.

Here is an example of them:

---

<sup>2</sup> This further feature is needed to create thread-safe context where objects run simultaneously and may alter the operand in the time between the null-check and the subsequent operation execution, thereby causing a crash or an unexpected behavior.

```
class Point
{
    public Point(int x, int y) {
        X = x;
        Y = y;
    }
    public override bool Equals(object other) {
        return other!=null && other is Point &&
            X==(other as Point).X && Y==(other as Point).Y;
    }
    public override int GetHashCode() => X^Y;
    public override String ToString() => $"({X},{Y})";
    public int X, Y;
}

public class Program {
    static void Main() {
        var p1 = new Point(1,2);
        var p2 = new Point(1,2);
        var p3 = p1;
        var p4 = new Point(2,1);
        Console.WriteLine(p1.ToString()); //output: (1,2)
        Console.WriteLine(p2.ToString()); //output: (1,2)
        Console.WriteLine(p3.ToString()); //output: (1,2)
        Console.WriteLine(p1.GetHashCode()); //output: 3
        Console.WriteLine(p2.GetHashCode()); //output: 3
        Console.WriteLine(p3.GetHashCode()); //output: 3
        Console.WriteLine(Object.Equals(p4, p2)); // //output: False
        Console.WriteLine(p4.GetHashCode()); //output: 3 [collision]
        Console.WriteLine(Object.Equals(p1, p2)); //output: True
        Console.WriteLine(Object.ReferenceEquals(p1, p2)); //output: False
        Console.WriteLine(Object.Equals(p1, p3)); //output: True
        Console.WriteLine(Object.ReferenceEquals(p1, p3)); //output: True
        int n = 3;
        int m = n;
        Console.WriteLine(Object.ReferenceEquals(m, n)); //output: False
    }
}
```

## 5.2 Reference Types & Value Types

C# has two main categories of variable types: *value types* and *reference types*. Differently from what said about reference types in Section 1, value types do not need to allocate any additional memory beyond the variables in which they reside. A value type can be either a structure type (see Section 5.3), an enumeration type (see Section 5.4), or *simple types* consisting of the basic built-in types like: integral numeric types (e.g., `int`), floating-point numeric types (e.g., `double`), `bool` representing the Boolean values, `char` representing the UTF-16 characters, ...

Mind that, by default, the assignment operation copies the variable values into another one. In the case of value-type variables, the corresponding type instances are copied and, after that, a modification to one of the variables does not affect the other. Things change for reference types since, after the copy, we have two references to the same object so that any modification to the referred object is reflected on both references.

A final note about `string` type which is a reference type, but it is *immutable*. Therefore, a new chunk of memory is allocated to store a new string and copying a string variable creates a new reference to the block of memory storing the original string (similarly to the reference types). However, any change to the content of one of the two copies causes the allocation of a new block of memory for storing the new string so that the other keeps its content (similarly to the value types).

Example:

```
struct Point {
    public Point(double x, double y) {
        X = x;
        Y = y;
    }
    public double X, Y; // value-type variables
}

class Circle {
    Point Center; // value-type variable
    double Radius; // value-type variable
    public Circle(double x, double y, double r) {
        Center = new Point(x,y);
        Radius = r;
    }
    public double Area() {
        return Radius*Radius*Math.PI;
    }
    public void SetCenter(double x, double y) {
        Center = new Point(x,y);
    }
}
```

```
}  
public void SetRadius(double r) {  
    Radius = r;  
}  
public void ShowMe() {  
    Console.WriteLine("I'm a circle centered in ({0},{1}) and radius {2}",  
        Center.X, Center.Y, Radius);  
}  
}  
  
class Program {  
    enum Seasons { Winter, Spring, Autumn, Summer };  
    static void Main() {  
        Circle c1, c2; // refernce-type variables  
        c1 = new Circle(1,1,10);  
        c1.ShowMe();  
        c2 = c1;  
        c2.SetCenter(2,2);  
        c2.SetRadius(4);  
        c1.ShowMe();  
        int x1, x2; // value-type variables  
        x1 = 10;  
        x2 = x1;  
        x1 = 5;  
        Console.WriteLine("x1={0} and x2={1}", x1, x2);  
        Seasons s1, s2; // value-type variables  
        s1 = s2 = Seasons.Winter;  
        s1 = Seasons.Summer;  
        if(s1!=s2)  
            Console.WriteLine("s1 and s2 differ");  
        string a = "Hello "; // refernce-type variable  
        string b = a;  
        Console.WriteLine(Object.ReferenceEquals(a,b)); // true  
        Console.WriteLine(a+b);  
        b = "world!";  
        Console.WriteLine(Object.ReferenceEquals(a,b)); // false  
        Console.WriteLine(a+b);  
    }  
}
```

### 5.3 The Structure Type

Like classes, structure types (structs for short) encapsulate attributes and the related functionalities (e.g., methods and properties) but, unlike classes, structs are value types (see Section 5.2), which means they are stored on the stack (thus they are more memory-efficient than reference types like classes) even if they are instantiated by means of the `new` operator. Creating a struct, you cannot explicitly define a parameterless constructor while a parameterized one must assign all attributes. Moreover, structs do not support inheritance but can implement interfaces. Example:

```
public interface IShape {
    double Area();
}

public class Point {
    public Point(int x, int y) {
        X = x;
        Y = y;
    }
    public int X { get; set; }
    public int Y { get; set; }
    public override string ToString() => $"({X},{Y})";
}

public struct Triangle
{
    public Triangle(double b, double h, Point p) {
        Base = b;
        Height = h;
        Position = p;
    }
    public void ScaleHeight(double f) => Height *= f;
    public Point Position;
    public double Base { get; private set; }
    public double Height { get; private set; }
    public double Area() => Base*Height*0.5;
    public override string ToString() => $"({Base}*{Height}) at {Position}";
}

class Program {

    static void Main() {
        Triangle t1 = new Triangle(7,6, new Point(3,3));
    }
}
```



```
Triangle t2 = t1;
t1.ScaleHeight(0.5);
t1.Position.X += 10;
Console.WriteLine("t1 : "+t1);
Console.WriteLine("t2 : "+t2);
}
}
```

## 5.4 The Enumeration Type

The enumeration type (enum for shorts) is a value type defined by a list of constants mapped to a set of integral values (of type `int` by default). Enums can be customized by choosing a different integral numeric type (e.g., `sbyte` which requires less bits than `int` to be represented in memory) as well as we can explicitly assign constants to specific values. Furthermore, an enum constant can be cast to the corresponding integral value and vice-versa. Example:

```
enum Season : sbyte { Spring=1, Summer=2, Autumn=3, Winter=4 }

class Program {
    static void Main() {
        int NumOfSeasons = Enum.GetNames(typeof(Season)).Length;
        Season s = (Season) new Random().Next(1, NumOfSeasons+1);
        Console.WriteLine("Random season: "+s);
        DayOfWeek d = DateTime.Now.DayOfWeek;
        if(d==DayOfWeek.Saturday || d==DayOfWeek.Sunday)
            Console.WriteLine(d+", it's weekend :)");
        else
            Console.WriteLine(d+", it's a working day :(");
    }
}
```

In the example above, `Enum.GetNames` is a static method returning an array of strings containing the names of the constants specified in the enum passed as input, while `DayOfWeek` is a system-defined enumeration type.

## 5.5 More Modifiers

In what follows, we discuss some modifiers that have been used in the examples of this handbook.

**out & ref.** These are two keywords used to pass arguments to methods. By default, parameters are passed by value while using `ref` or `out` we pass the reference. The difference between them is that `out` requires that the method assigns a value to the related variable while `ref` does not. However, a parameter passed with `ref` must be initialized before to be passed to a method.

```
class Program {
    static void Swap(ref int a, ref int b) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    static bool SolveQuadEq(int a, int b, int c, out float x1, out float x2) {
        float dsc = b*b-4.0*a*c;
        if(dsc<0) {
            x1 = Single.NaN; // Not a Number
            x2 = Single.NaN;
            return false;
        }
        float sqrt = (float) Math.Sqrt(dsc);
        x1 = (-b + sqrt) / (2*a);
        x2 = (-b - sqrt) / (2*a);
        return true;
    }
    static void Main() {
        int x = 1, y = 4; // mandatory init
        Console.WriteLine(x+" "+y);
        Swap(ref x, ref y);
        Console.WriteLine(x+" "+y);
        float x1, x2;
        SolveQuadEq(2, 15, 7, out x1, out x2);
        Console.WriteLine(x1+", "+x2);
    }
}
```

**const & readonly.** Both modifiers denote a constant attribute that cannot change: `const` is for declaring compile-time constants while `readonly` allows a value to be calculated at runtime and set in the constructor, i.e., it becomes constant after the object has been instantiated:

```
public class Circle {
    public Circle(double r) {
        Radius = r;
    }
}
```

```
    }  
    public double Area{  
        get { return PiSquare*Radius; }  
    }  
    public readonly double Radius;  
    const double PiSquare = Math.PI*Math.PI;  
}  
  
public class Program {  
    static void Main() {  
        var c = new Circle(2);  
        Console.WriteLine("r={0} and area={1:N2}", c.Radius, c.Area);  
    }  
}
```

**static.** The `static` keyword refers to a modifier for declaring class members which belong to the type itself rather than to an object. Defining a class, this modifier can be applied to attributes, methods, and properties (i.e., attributes with attached the methods to get/set their values, see Section 5.7) that do not require an instance of the class to be used. Furthermore, the value of static attributes is shared among all the instances of the class (if any). Example:

```
public class Square {  
    public Square(int size) {  
        Size = size;  
        SquareCounter += 1;  
    }  
    static Square() {  
        SquareCounter = 0;  
    }  
    public static int Counter {  
        get => SquareCounter;  
    }  
    public override string ToString() => Size+"x"+Size+" square";  
    static int SquareCounter = 10;  
    int Size;  
}  
  
class Program {  
    static void Main() {  
        Console.WriteLine("Now you have "+Square.Counter+" squares.");  
        Square s1 = new Square(5);  
    }  
}
```

```

    Console.WriteLine("A new "+s1+" has been created");
    Console.WriteLine("Now you have "+Square.Counter+" squares.");
    Square s2 = new Square(7);
    Console.WriteLine("A new "+s2+" has been created");
    Console.WriteLine("Now you have "+Square.Counter+" squares.");
}
}

```

Note that, since the static members are not attached to objects, they are accessed by using the dot-notation with the class name instead of the object name. For the same reason, static methods cannot access to non-static attributes which belong to objects (while the opposite is allowed). Note that to initialize the static members a static parameterless constructor can be defined which is called automatically before any static member is accessed or any instance of the class is created.

Besides members, also classes can be static in C# and they are usually used for gathering utility methods that do not require to maintain a state. As a consequence, static classes are allowed to contain static or const members only. Example:


```

static class ConversionHelper {
    public static double DegToRad(double d) => d * ConversionRatio;
    public static double RadToDeg(double r) => r / ConversionRatio;
    public const double ConversionRatio = Math.PI / 180.0;
}

class Program {
    static void Main() {
        double d = 180.0;
        double r = ConversionHelper.DegToRad(d);
        d = ConversionHelper.RadToDeg(r);
        Console.WriteLine("{0:0.0###} deg = {1:0.0###} rad", d, r);
        Console.WriteLine("{0:0.0###} rad = {1:0.0###} deg", r, d);
    }
}

```

## 5.6 Exceptions

Exceptions are unexpected events thrown at runtime and they must be managed to prevent the program crashing or unexpected results. There are exceptions generated by the application (see the example at page 8 ) or by the system (e.g., `System.DivideByZeroException`) if an operation fails. The whole mechanism lets the client customize how to deal with errors. To make it possible, the code that might generate an exception must be enclosed in a `try` block followed by a number of

`catch` blocks each of them handling a different type of exception. When an exception occurs, only one catch block is executed, and it is chosen examining the ones available in the code according to their order.

Moreover, the `Exception` class can be specialized to better identify the type of exception generated. In these cases, it's required that the related `catch` blocks in the client code appear in order from most specific to most general. When creating your own exceptions, the user-defined exception name implements the common constructors, as shown in the following example:

```
public class QueueEmptyException : Exception {
    public QueueEmptyException() : base () {}
    public QueueEmptyException(string msg) : base (msg) {}
}

public class QueueFullException : Exception {
    public QueueFullException() : base () {}
    public QueueFullException(string msg) : base (msg) {}
}

class MyQueue : IEnumerable {
    public MyQueue(int capacity) {
        if(capacity<1)
            throw new Exception("capacity cannot be less than 1");
        data = new int[capacity];
        this.capacity = capacity;
        this.size = 0;
        this.head = 0;
        this.tail = 0;
    }
    public void Enqueue(int value) {
        if(capacity==size)
            throw new Exception("queue is full");
        data[head] = value;
        head = (head+1)%capacity;
        size += 1;
    }
    public int Dequeue() {
        if(size==0)
            throw new Exception("queue is empty");
        int i = tail;
        tail = (tail+1)%capacity;
        size -= 1;
        return data[i];
    }
}
```

```
}
public IEnumerator GetEnumerator() {
    for(int i=0; i!=size; ++i)
        yield return data[(tail+i)%capacity];
}
private int[] data;
private int size;
private int capacity;
private int head, tail;
}

class Program {
    static void Main() {
        var q = new MyQueue(2);
        try {
            q.Enqueue(3);
            q.Enqueue(4);
            q.Enqueue(14);
        }
        catch (QueueEmptyException) {
            Console.WriteLine("q is Empty!");
        }
        catch (QueueFullException) {
            Console.WriteLine("q is Full!");
        }
        catch (Exception e) {
            Console.WriteLine(e);
        }
        Console.WriteLine("q.Pop() = {0}", q.Dequeue());
    }
}
```

## 5.7 Properties

Properties are part of the “syntactic sugar” of C# making things easier to read or to express. They are a special type of class-members that allow to elegantly access the value of a private field by means of the ‘=’ operator (the same as if the field is a simple variable) instead of defining the heavier GetValue and SetValue methods. Example:

```
public class Person {
    public string Name {
```



```

        get { return name; }
        set { name = CultureInfo.CurrentCulture.TextInfo.ToTitleCase(value); }
    }
    public string Surname {
        get { return surn; }
        set { surn = CultureInfo.CurrentCulture.TextInfo.ToTitleCase(value); }
    }
    public override string ToString() => $"{name} {surn}";
    string name, surn;
}

class Program {
    static void Main() {
        var p = new Person();
        p.Name = "bruno";
        p.Surname = "rosa";
        Console.WriteLine(p+", the surname is "+p.Surname);
    }
}

```

Note that `value` is a keyword, and it refers to the value passed to the `set` accessor.

Moreover, when defining a property, it's not mandatory to define both `get` and `set` so that it's possible to declare a property which is read-write, read-only, or write-only; as well as we can append an access modifier to limit the visibility of an accessor, e.g., `private` or `protected` (see Section 1.1 and Section 3). When an accessor does not include any extra logic, but it just assigns/retrieves the property value, we can further simplify the code letting the compiler the burden of completing the implementation by hiding it. Last but not the least, properties can even be used to define an interface:

```

public interface IAreable {
    float Area { get; }
}

public class Square : IAreable {
    public float Size { private get; set; }
    public float Area {
        get { return Size*Size; }
    }
}

class Program {
    static void Main() {
        var s = new Square();
    }
}

```

```

    s.Size = 9;
    Console.WriteLine("The area of the square s is "+s.Area);
}
}

```

## 5.8 Tuples


Tuples let group an arbitrary number of data elements in a lightweight data structure and, optionally, name each of them (otherwise an element can be referred with the keyword `Item` followed by the ordinal number of the element in the tuple, e.g., `Item1` refers the first one). Moreover, tuples are mutable value-types of which elements are public and support by default the operators `'=='` and `'!='` that perform a pairwise comparison of the left and right operands. Example:

```

class Program {
    static void Main() {
        (double Real, string, int Natural) tuple = (4.5, "foo", 6);
        Console.WriteLine(tuple.ToString()); //output: (4.5, foo, 6)
        Console.WriteLine(tuple.Real); //output: 4.5
        Console.WriteLine(tuple.Item1); //output: 4.5
        Console.WriteLine(tuple.Item2+" "+tuple.Natural); //output: foo 6
        Console.WriteLine(tuple.Real+tuple.Natural); //output: 10.5
        var copyof_tuple = tuple;
        Console.WriteLine(tuple == copyof_tuple); //output: True
        tuple.Item2 = "bar";
        Console.WriteLine(tuple.ToString()); //output: (4.5, bar, 6)
        Console.WriteLine(copyof_tuple.ToString()); //output: (4.5, foo, 6)
        Console.WriteLine(tuple == copyof_tuple); //output: False
    }
}

```

## 5.9 Events

An *event* enables an object (so-called *publisher*) to notify other objects (so-called *subscribers*) when something of interest occurs while subscribers, for their part, execute a procedure (so-called *handler*) in response to an event notification (similarly to the scenarios described in Section 9.2). This mechanism is built upon delegates (see Section 4.4) that are used to bind handlers to events. However, differently from delegates, events can only be declared within a class (by means of the `event` keyword) while subscribers are only allowed to bind and unbind their handlers to public events so that, e.g., an event cannot be directly triggered from outside the class it belongs to . Attaching, detaching, and invoking the registered handlers is done with the same syntax introduced for delegates:



```
public class Stopwatch {
    public Stopwatch() => Start = DateTime.Now;
    public void Trigger() {
        double elapsed = (DateTime.Now-Start).TotalSeconds;
        Handlers?.Invoke(this, new StopwatchEventArgs(elapsed));
    }
    public event EventHandler<StopwatchEventArgs> Handlers;
    DateTime Start;
}

public class StopwatchEventArgs : EventArgs {
    public StopwatchEventArgs(double elapsed) {
        Elapsed = elapsed;
    }
    public readonly double Elapsed;
}

class Program {
    static void Main(string[] args) {
        var s = new Stopwatch();
        s.Handlers += QPressed; // handler subscription
        Console.Write("Press 'q' to quit... ");
        while (true)
            if (Console.ReadKey(true).KeyChar == 'q')
                s.Trigger();
    }
    // event handler
    static void QPressed(object sender, StopwatchEventArgs e) {
        Console.WriteLine("'q' pressed after {0:0.000} seconds", e.Elapsed);
        Environment.Exit(0);
    }
}
```

The example above shows how to declare an event named `Handlers` which is associated with the `EventHandler` delegate and raised in a method named `Trigger`. Once the instance `s` of `Stopwatch` has been generated, the client attaches its own `QPressed` method to `s` so that, when the user presses the 'q' key, the method `Trigger` invokes the client method. Also, the `StopwatchEventArgs` class specializes `EventArgs` to include the necessary data we want to transmit to the subscriber when the event occurs, i.e., the elapsed time since the publisher has been created. Doing this mind that any specialization of `EventArgs` should always include the source of the event, i.e., `sender`, and the event data.

## Part II

# Object Oriented Design

Now we have the basics of C# and OOP, let us shift our focus on how to design a software system made of multiple objects interacting with each other. Identifying the proper set of classes for a given problem is an ability that gets better over time. For example, modeling too many classes may result in poor performance and unnecessary complexity, while too-few large classes tend to make them hard to reuse and difficult to maintain and understand.

We firstly introduce the UML notation that facilitates the representation of a class and the relations among a set of them (Section 6). Cohesion and coupling (Section 7) are two concepts used to evaluate the quality of our solutions. The final sections discuss the SOLID principles (Section 8) and the design patterns (Section 9) that will guide us to produce a more maintainable and scalable code.

## 6 UML Class Notation: the Basics

When a program is running, objects interact with each other by invoking methods and exchanging their references, as well as a class definition may rely on other classes by means of composition and inheritance or implement one or more interfaces. This creates a sort of network that connects various parts of our code. Example:

```
public class Point {
    public Point(double x, double y) {
        X = x;
        Y = y;
    }
    public double X { get; set; }
    public double Y { get; set; }
    public override string ToString() => "("+X+", "+Y+") ";
}

public interface IArea {
    double Area();
}

public class EqTriangle : IArea {
    public EqTriangle(double side) => Side = side;
    public double Area() => Side*Side*Math.Sqrt(3)/4;
    double Side;
}
```

```
}

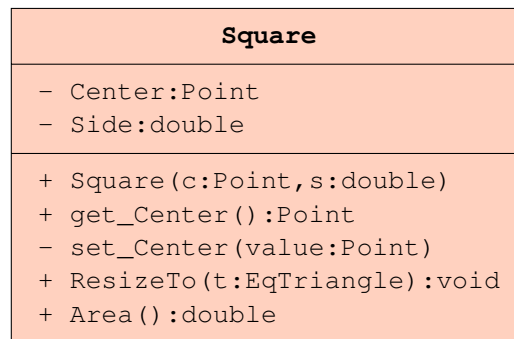
public class Square : IArea {
    public Square(Point c, double s) {
        Center = c;
        Side = s;
    }
    public Point Center { get; private set; }
    public void ResizeTo(EqTriangle t) {
        Side = Math.Sqrt(t.Area());
    }
    public double Area() => Side*Side;
    double Side;
}

public class ColoredSquare : Square {
    public ColoredSquare(Point c, double s, ConsoleColor cl)
    : base(c, s) {
        Color = cl;
    }
    public ConsoleColor Color { get; private set; }
}

public class Program {
    static void Main(string[] args) {
        Point pt = new Point(3, 3);
        ColoredSquare sq = new ColoredSquare(pt, 10, ConsoleColor.Blue);
        Console.WriteLine("center: "+sq.Center);
        Console.WriteLine("area: "+sq.Area());
        Console.WriteLine("color: "+sq.Color);
        sq.ResizeTo(new EqTriangle(6));
        Console.WriteLine("area: "+sq.Area());
    }
}
```

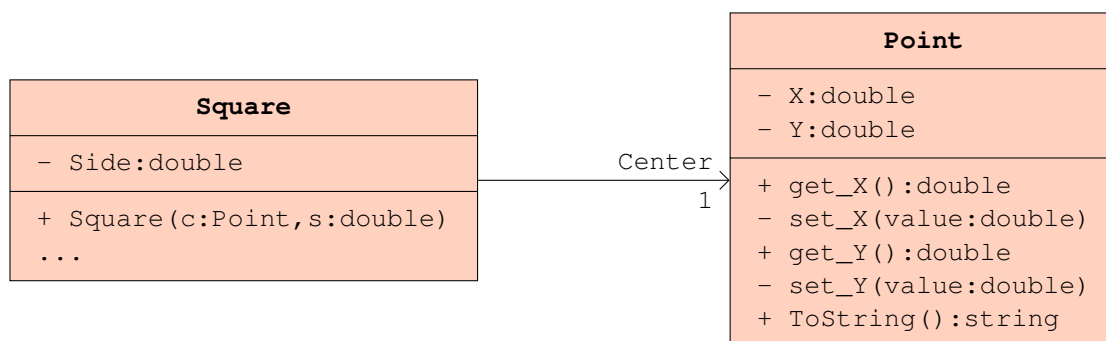
A *class diagram* schematizes the relationships among the entities composing a object oriented software system and Unified Modeling Language (UML) is a standardized modeling language used in software engineering for creating such diagrams. In what follows, we present some UML definitions that will be used in the rest of this document (for a more comprehensive guide about UML you can refer to this book [2]).

**Class.** A class is represented by a rectangle containing three sections stacked vertically: the top one shows the class name, the middle one lists the attributes, and the one at the bottom lists the methods. When representing a class in a diagram, the only mandatory section is the top one with the name while the others are optional, and they may come with or without signature (it depends on the level of detail required). Moreover, it is a widespread practice to state the visibility level of members with the symbols + or - (respectively for public and private members). Here is the representation of the class `Square` from the previous example:



From the above representation, we can learn that the class `Square` has: two private attributes `Side` and `Center` of type `double` and `Point` respectively, one private method `set_Center` which is implicitly part of the property `Center`, and four public methods: the constructor `Square` which does not need to declare the return-type and takes a `Point` reference and a `double` value as input, the method `get_Center` returning a `Point` object, `ResizeTo` taking an `EqTriangle` reference as input and returning nothing, and `Area` returning a `double`. Note that UML has no standard representations for the C# properties so that the one used above is a workaround.

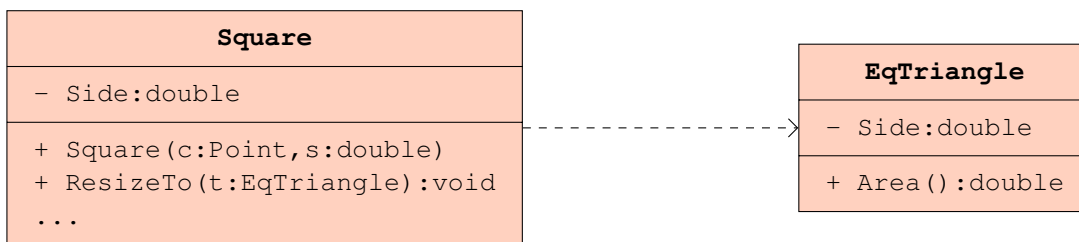
**Association.** It indicates a permanent relationship between two classes and represents the composition case seen in Section 3, i.e., when a class holds a reference to another one in its state (e.g., `Point` is part of `Square` in the previous example). This type of connection is represented by a solid line with an open arrowhead pointing to the independent class that may contain the role name and multiplicity value for the known class:



In the above example, the `Square` class knows about the `Point` class that plays the role of `Center`. However, the `Point` class has no idea that it is associated with `Square`. The multiplicity value next to the `Point` class is 1 which means that an instance of a `Square` has exactly one instance of `Point` associated with it. The table below lists some examples of multiplicity values along with their meanings:

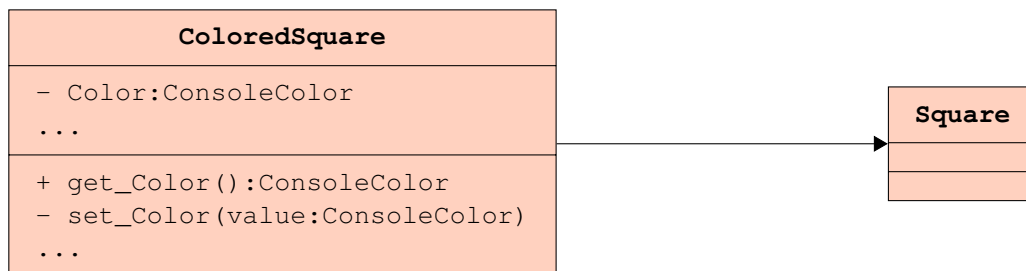
Indicator	Meaning
0..1	zero or one
2..10	from two to ten
1	one only
*	zero or more
1..*	at least one

**Dependency.** This type of relationship is temporary and identifies the cases when an object of a given class takes part in the method implementation of another class, e.g., it occurs as reference among the input parameters, like `EqTriangle` in the method `ResizeTo`. This type of connection is represented by a dashed line with an open arrowhead pointing to the independent class:



Note that an association often produces a dependency, like `Square` and `Point`, while the opposite is not necessarily true.

**Generalization.** This relationship is used to describe that a class inherits from another one. It is represented with a solid line with a filled triangle as head where the tail is attached to the subclass while the head points to the more general class. Example:



**Realization.** An interface is considered a sort of child class thus it is represented like a class adding the label `interface` to top section of the box. To remark that a class implements an interface, we use a dashed arrow with an unfilled triangle as head pointing the interface. Example:



## 7 Cohesion & Coupling

In software engineering, cohesion and coupling are two recurrent concepts for measuring the quality and design of software systems. Defined by Stevens et al. [3] in the '70s, they play a crucial role in determining how maintainable, scalable, and understandable a software system is. The definition of these two metrics will be given in terms of classes while their application can be extended to any type of software module.

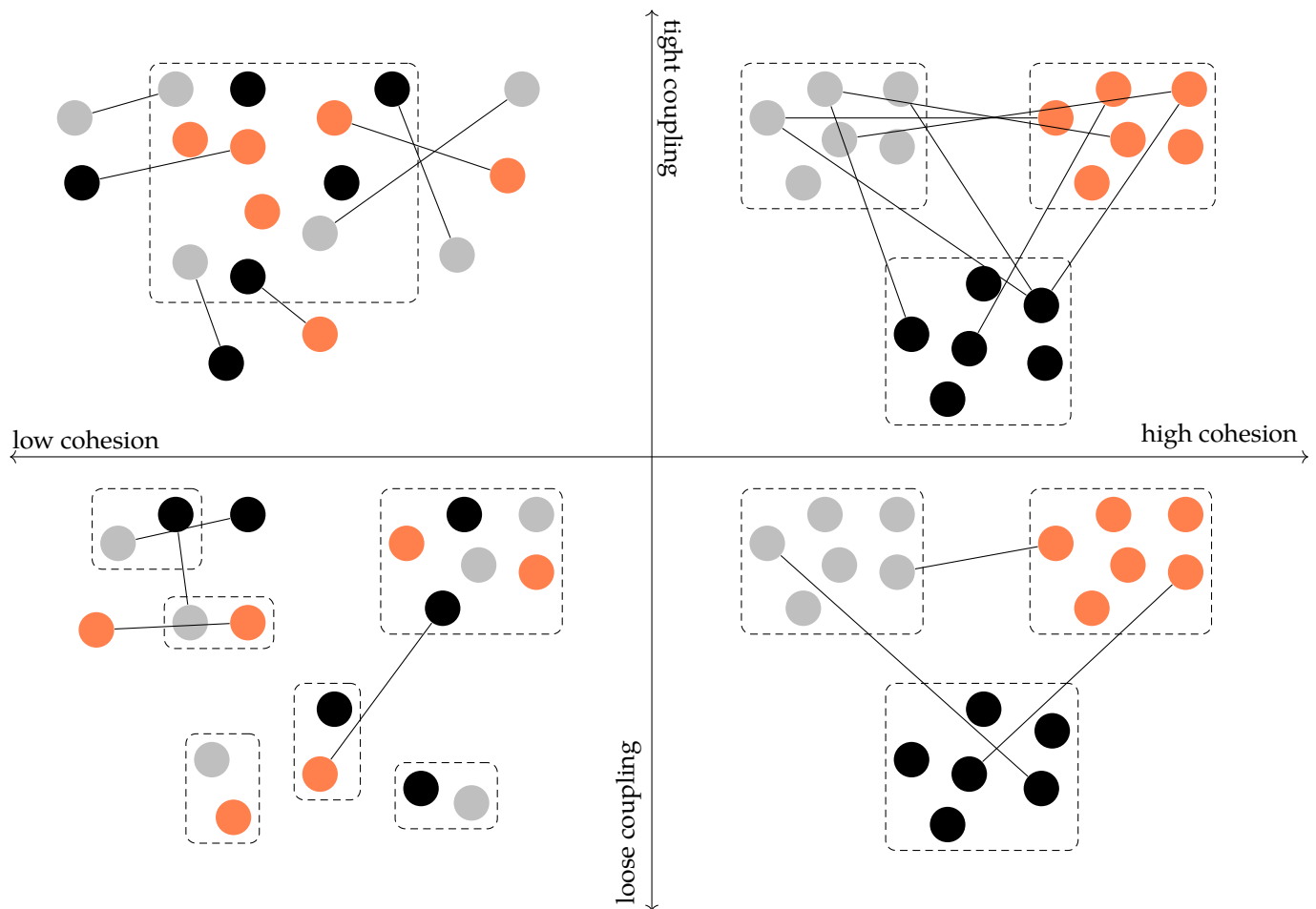
**Cohesion.** It refers to the degree to which the distinct parts of a class work together for achieving a common purpose. A class dealing with many poorly related responsibilities and having multiple independent tasks to fulfill is a symptom of low cohesion. On the contrary, a cohesive class is designed around one specific and well-defined purpose. High cohesion makes it easier to conduct code correction and update as well as it reduces the chance of having duplicated functionalities across many classes. Moreover, having single-task classes facilitates the design of efficient and effective test units that improves error isolation.

**Coupling.** It represents the degree to which a class depends on others. For instance, two classes are said to be tightly coupled, if they are connected so that changing one implies to change also the other. A good design aims to have loosely coupled classes that, e.g., can be developed and tested separately. Furthermore, a class which is tightly coupled to others also has few chances to be reused due to all the dependencies and associations that come with it.

**Cohesion vs Coupling.** The previous considerations show that cohesion and coupling are often related. High cohesion correlates with low coupling: a class of which parts are tightly related and serving a single purpose would rarely depend on other classes to promote loose coupling. However, tight coupling can be a sign of low cohesion: the extreme case occurs when a single object (so-called *god object*) manages most of the data and/or responsibilities of the system which makes it connected to a lot of minor satellite classes and types. Sometimes, some solutions consist of a set of cohesive

classes that depend on each other: in these cases, we can make some of these relationships more flexible by adding more abstractions like interfaces (see Section 8.5). The last scenario is made of many fragmented classes that are not very dependent on each other, but also not focused and do unrelated stuff, making them harder to read, understand and maintain.

The next figure tries to visualize the four possible cases listed above: the dashed shapes represent the boundaries of a class, the circles with the same color represent class members that are related to the same responsibility, the edges connecting pairs of circles represent dependencies and associations.



The principles discussed in Section 8 will help us to produce solutions that improve the cohesion and coupling aspects.

## 8 The S.O.L.I.D. Principles

The SOLID principles [4] are a set of five generic rules helping us to provide loosely coupled and encapsulated solutions when designing a software system. SOLID is the acronym of:

- 🔖 Single Responsibility Principle (Section 8.1),
- 🔖 Open Closed Principle (Section 8.2),
- 🔖 Liskov Substitution Principle (Section 8.3),
- 🔖 Interface Segregation Principle (Section 8.4),
- 🔖 Dependency Inversion Principle (Section 8.5).

## 8.1 Single Responsibility Principle

The Single Responsibility Principle (SRP) states that a class (as well as methods, interfaces, ...) should have only one reason to change. This means that each class should stay focused on a single responsibility, avoiding merging different ones together. In other words, the principle aims to isolate a class from the complexities of the diverse types of functionalities that it is required to implement so that a request for changing to one of them does not affect the others that are completely unrelated to requested one. When applied properly SRP has several benefits: it promotes code modularity by making your solutions more readable, easy-to-test, maintainable, reusable, and reduces the risk that one change can inadvertently interfere with other functionalities. On the contrary, a design lacking SRP usually ends up in a god object which gathers most of the data and does too much with them.

Imagine you are being asked to define an interface for identifying a generic regular polygon, calculating its area and perimeter, and saving its string representation into a text file. At the first attempt the interface looks like the following:

```
public interface IRegularPolygon {
    void SaveOnTxtFile(string path);
    double Area { get; }
    double Perimeter { get; }
    double Side { get; set; }
}

public class Square : IRegularPolygon {
    public void SaveOnTxtFile(string path) {
        string text = this.ToString();
        if(File.Exists(path)) File.AppendAllText(path, text);
        else File.WriteAllText(path, text);
    }
    public double Area { get => Side*Side; }
    public double Perimeter { get => Side*4; }
    public double Side { get; set; }
    public override string ToString() => "Type="+GetType()+" Side="+Side;
```



```
}  
  
class Program {  
    static void Main() {  
        Square s = new Square();  
        s.Side = 8;  
        Console.WriteLine(s);  
        Console.WriteLine(s.Area);  
        Console.WriteLine(s.Perimeter);  
        s.SaveOnTxtFile("data.txt");  
    }  
}
```

The main issue of the solution from the SRP perspective is that while the three properties (i.e., Area, Perimeter, and Side) are inherent with regular polygons, the method `SaveOnTxtFile` is about implementing an additional functionality. Consequently, we try to separate it from the interface so that a change to such a functionality (e.g., write onto a pdf file instead) will not affect the rest of the interface and the classes that implement it.

For example, we could apply the Strategy design pattern which will be introduced in Section 9.7 implementing the additional functionality in a distinct class. Furthermore, this solution let us extend the set of destination-file types by defining further classes that implement `ISaveOnFile`:

```
public interface IRegularPolygon {  
    double Area { get; }  
    double Perimeter { get; }  
    double Side { get; set; }  
}  
  
public interface ISaveOnFile {  
    void Save(string path, string text);  
}  
  
public class Square : IRegularPolygon {  
    public void SetSaver(ISaveOnFile s) => saver = s;  
    public void SaveOnFile(string path) {  
        string text = this.ToString();  
        saver.Save(path, text);  
    }  
    ISaveOnFile saver;  
    public double Area { get => Side*Side; }  
    public double Perimeter { get => Side*4; }  
    public double Side { get; set; }  
}
```

```
    public override string ToString() => "Type="+GetType()+" Side="+Side;
}

public class SaverOnTxtFile : ISaveOnFile {
    public void Save(string path, string text) {
        if(File.Exists(path)) File.AppendAllText(path, text);
        else File.WriteAllText(path, text);
    }
}

class Program {
    static void Main() {
        Square s = new Square();
        s.Side = 8;
        Console.WriteLine(s);
        Console.WriteLine(s.Area);
        Console.WriteLine(s.Perimeter);
        s.SetSaver(new SaverOnTxtFile());
        s.SaveOnFile("data.txt");
    }
}
```

Another solution would be to reverse the perspective of the problem inspired by the Visitor design pattern in Section 9.3. In practice, instead of charging the polygon class of implementing the writing functionality, that responsibility is passed to a different component that accepts an object implementing `IRregularPolygon` as input and provides to write it into a file.

## 8.2 Open Closed Principle

The Open Closed Principle (OCP) states that a class (as well as methods, interfaces, ...) should be open for accepting new functionalities without changing the existing code preventing situations in which a change to one of your classes also requires you to adapt all dependent classes. Most of the time, this can be achieved by creating an abstraction that is fixed and then a group of entities that realize the abstraction and can be extended. In practice, we can have an abstract class inherited by different subclasses as well as an interface implemented by multiple classes. The following example of the open-closed principle shows a class using a chain of if-statements to produces the sound of a given animal:

```
public class Dog {}

public class Lion {}
```



```
public class Cat {}

public class Player {
    public static string Sound(object animal) {
        if (animal is Lion)
            return "Roar!";
        if (animal is Cat)
            return "Meow!";
        if (animal is Dog)
            return "Arf!";
        throw new ArgumentException("Not supported");
    }
}

class Program {
    static void Main() {
        Console.WriteLine(Player.Sound(new Dog()));
    }
}
```

However, there is a real possibility you want to grab a wider set of animals which requires you to modify the class which in turn violates the OCP. To fix such an issue, we can define an interface that each animal class implements independently so that adding a new animal does not require any modification in the existing ones:

```
public interface ISound {
    string Sound();
}

public class Dog : ISound {
    public string Sound() => "Arf!";
}

public class Lion : ISound {
    public string Sound() => "Roar!";
}

public class Cat : ISound {
    public string Sound() => "Meow!";
}
```

```
public class Player {  
    public static string Sound(ISound animal) => animal.Sound();  
}  
  
class Program {  
    static void Main() {  
        Console.WriteLine(Player.Sound(new Dog()));  
    }  
}
```

### 8.3 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that it should be possible to make use of a subclass object where one belonging to the superclass is required (e.g., as actual parameter in a method call) without introducing any unexpected behavior when the program runs, otherwise the LSP is considered violated. Imagine you have the class `Rectangle` shown below and you are asked to also implement the square class for which there must be defined the methods for calculating perimeter and area as `Rectangle` does. From the mathematical perspective, squares are a sub-type of rectangles so that you decide to inherit `Square` from `Rectangle` overriding the properties for handling `Width` and `Height` given that for squares they must be the same:

```
public class Rectangle {  
    public virtual double Width { get; set; }  
    public virtual double Height { get; set; }  
    public virtual double Area() => Width*Height;  
    public virtual double Perimeter() => 2*Width+2*Height;  
    public override string ToString() => $"{Width:.00}x{Height:.00}";  
}  
  
public class Square : Rectangle {  
    public Square(double s) {  
        Size = s;  
    }  
    public override double Width {  
        get { return Size; }  
        set { Size = value; }  
    }  
    public override double Height {  
        get { return Size; }  
        set { Size = value; }  
    }  
}
```

```
    double Size;
}

class Program {
    static void Scale(Rectangle rect, double ratio) {
        // change the rectangle aspect ratio by keeping the same area
        double a = rect.Area();
        double x = Math.Sqrt(ratio*a)/ratio;
        rect.Width = ratio*x;
        rect.Height = x;
    }
    static void Main() {
        Rectangle rect = new Square(6);
        Scale(rect, 2);
        Console.WriteLine(rect);
    }
}
```

The method `Scale` is supposed to work with any `Rectangle` instance, but it does not with the instances of the subclass `Square` due to the implementation of `Width` and `Height` that refer to the same attribute and this in turn violates the LSP. Hence, we can remove the inheritance by gathering the two classes under a common interface that includes the required methods, i.e., `Area` and `Perimeter`:

```
public interface Quadrilateral {
    double Area();
    double Perimeter();
}

public class Rectangle : Quadrilateral {
    public double Width { get; set; }
    public double Height { get; set; }
    public double Area() => Width*Height;
    public double Perimeter() => 2*Width+2*Height;
    public override string ToString() => $"{Width:.00}x{Height:.00}";
}

public class Square : Quadrilateral {
    public Square(double s) {
        Size = s;
    }
    public double Area() => Size*Size;
    public double Perimeter() => 4*Size;
}
```

```
    public double Size { get; set; }
}

class Program {
    static void Scale(Rectangle rect, double ratio) {
        // change the rectangle aspect ratio by keeping the same area
        double a = rect.Area();
        double x = Math.Sqrt(ratio*a)/ratio;
        rect.Width = ratio*x;
        rect.Height = x;
    }
    static void Main() {
        Rectangle rect = new Rectangle();
        rect.Width = rect.Height = 6;
        Scale(rect, 2);
        Console.WriteLine(rect);
    }
}
```

## 8.4 Interface Segregation Principle

The Interface Segregation Principle (ISP) suggests that a class should not be forced to implement methods which are irrelevant to it, for example when implementing an interface. Like the SRP, this principle aims to obtain classes having small and focused interfaces to reduce the side effects from gathering methods dealing with different responsibilities in the same class. Instead of having a “fat” and monolithic interface that may lead to code bloat, increased coupling, and incorrectly implemented interfaces, it would be better to split it in multiple ones so that each class can implement only those inherent to itself.

```
public interface IBird {
    void GetOutOfEgg();
    void Fly();
}

public class Eagle : IBird {
    public void GetOutOfEgg() => Console.WriteLine("Ok! No problems");
    public void Fly() => Console.WriteLine("Ok! No problems");
}

public class Penguin : IBird {
    public void GetOutOfEgg() => Console.WriteLine("Ok! No problems");
}
```

```
    public void Fly() => throw new NotImplementedException();
}

class Program {
    static void Main() {
        IBird bird;
        bird = new Penguin();
        bird.Fly();
    }
}
```

In the example above, we can divide the interface `IBird` in two interfaces defining one method each, e.g., `IGetOutOfEgg` and `IFly`, and then define the two classes as it follows:

```
public interface IGetOutOfEgg {
    void GetOutOfEgg();
}

public interface IFly {
    void Fly();
}

public class Eagle : IGetOutOfEgg, IFly {
    public void GetOutOfEgg() => Console.WriteLine("Ok! No problems");
    public void Fly() => Console.WriteLine("Ok! No problems");
}

public class Penguin : IGetOutOfEgg {
    public void GetOutOfEgg() => Console.WriteLine("Ok! No problems");
}
```

## 8.5 Dependency Inversion Principle

The general idea of the Dependency Inversion Principle (DIP) deals with improving the design of situations where an object depends on another object (like the class composition seen in Section 3). DIP suggests modeling such relationships by means of abstractions (e.g., interfaces and abstract classes) rather than references to concrete classes. Given that interfaces change less frequently than concrete classes, DIP goes to the full advantage of the reusability and flexibility of the dependent components. The following example shows the case of the class `Switch` that is dependent on `Lamp` while there are multiple devices that could be connected to it (e.g., a ventilator):

```
public class Lamp {
```



```
public bool State { get; private set; }
public void ChangeState() => State = !State;
public override string ToString() => State ? "Light!" : "Dark!";
}

public class Switch {
    public Switch(Lamp l) {
        lamp = l;
    }
    public string Press() {
        lamp.ChangeState();
        return lamp.ToString();
    }
    Lamp lamp;
}

class Program {
    static void Main() {
        Switch s = new Switch(new Lamp());
        Console.WriteLine(s.Press());
    }
}
```

From the DIP perspective, the ideal is to define an interface, e.g., `ISwitch`, with the minimal set of methods required to make a device class compatible with `Switch` which in turn needs to be modified to host any object implementing the new interface:

```
public interface ISwitch {
    void OnPress();
}

public class Lamp : ISwitch {
    public bool State { get; private set; } = false;
    public void OnPress() => State = !State;
    public override string ToString() => State ? "Light!" : "Dark!";
}

public class Fan : ISwitch {
    public enum Speed { Off, Low, Medim, High };
    public void OnPress() => speed=(Speed) ((1+(int) speed)%4);
    public override string ToString() => speed.ToString();
    Speed speed = Speed.Off;
}
```



```
public class Switch {
    public Switch(ISwitch d) {
        device = d;
    }
    public string Press() {
        device.OnPress();
        return device.ToString();
    }
    ISwitch device;
}

class Program {
    static void Main() {
        Switch s1 = new Switch(new Lamp());
        Console.WriteLine("s1:" + s1.Press());
        Switch s2 = new Switch(new Fan());
        Console.WriteLine("s2:" + s2.Press());
        Console.WriteLine("s2:" + s2.Press());
    }
}
```

## 9 Design Patterns

When designing a software system, it happens to stumble into similar problems that may be solved applying similar conceptual approaches. *Design patterns* are designed precisely for this purpose: to provide us with standard solutions that we can customize to solve a design problem in OOP. They also define a common language making team communications more efficient. Design patterns can be categorized by their goal into three main groups:

- 👉 *Creational* patterns provide object creation mechanisms that increase flexibility and reuse of existing code.
- 👉 *Structural* patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- 👉 *Behavioral* patterns take care of effective communication and the assignment of responsibilities between objects.

As it is shown in the rest of the section, C# provides a built-in support for implementing several design patterns as well as some of the C# features rely on design patterns. It is also worth noting that not all

principles discussed previously are absolute and some of the next design patterns go against some principles, however, applying a pattern we assume that the benefit from it is greater than the benefit of not using it at all.

Last but not the least, we mention the *antipatterns* that are solutions that look plausible in the beginning of the design, but eventually lead to inefficiency and maintainability issues in the system. Here two examples:

- 🔖 *god objects* introduced in Section 7;
- 🔖 *magic numbers* that are numeric values occurring in the code having no obvious meaning. It's important to replace them with, e.g., constants having names that make sense to improve the readability and maintainability of the code especially when that value occurs in different parts of the implementation like in this example:

```
public class MyArray {
    public MyArray() {
        data = new int[100];
    }
    public void RandomAssignment(int x) {
        data[rnd.Next(100)] = x;
    }
    public int LastPosition() => 99;
    int[] data;
    Random rnd = new Random();
}

class Program {
    static void Main() {
        var a = new MyArray();
        a.RandomAssignment(9);
        Console.WriteLine(a.LastPosition());
    }
}
```

To hardcode the value 100 makes the implementation confusing and difficult to maintain, e.g., in case you need to change the capacity of the array which is referred (directly or implicitly) in several places of the source. In this specific case, we could use the `Array.Length` property.

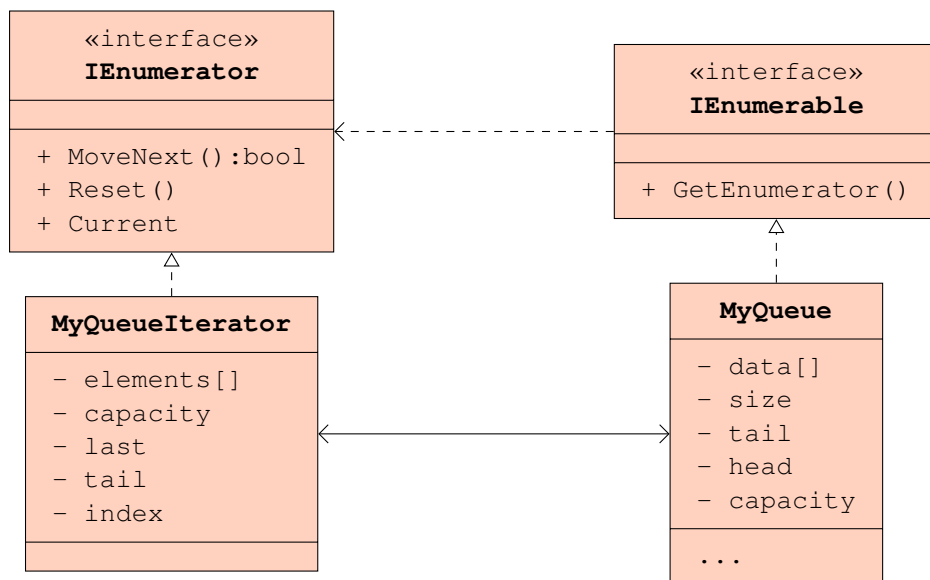
## 9.1 Iterator

*behavioral design pattern*

A set of similar data can be managed more efficiently when the elements are gathered into a *collection* and, once you have a collection, it is common to be able to scan it. To this end, the *iterator* pattern

is a behavioral pattern that allows to visit a collection without exposing how data are really stored (e.g., linked list, array, tree, hash table, etc.). For example, in the class implemented at page 8 [🔗](#), the method `GetEnumerator` returns an `IEnumerator` object for iterating over the items by means of the `foreach` keyword. This is how C# provides built-in support for this design pattern, i.e., through the `IEnumerable` and `IEnumerator` interfaces. In particular, the `IEnumerator` interface is the base interface for all non-generic enumerators where the `IEnumerator<T>` interface is the equivalent for generics. The same happens for `IEnumerable` and `IEnumerable<T>`.

For the sake of completeness, let us analyze a more “canonical” implementation of `MyQueue`, i.e., without the aid of the `foreach` statement or the `yield return` statement. This requires implementing from the scratch a new class, `MyQueueIterator`, implementing the `IEnumerator` interface as represented in the following diagram:



And here is the related code which describes how the interface works in detail:

```

class MyQueue : IEnumerable {
    public MyQueue(int capacity) {
        if(capacity<1)
            throw new Exception("capacity cannot be less than 1");
        data = new int[capacity];
        this.capacity = capacity;
        this.size = 0;
        this.head = 0;
        this.tail = 0;
    }
    public void Enqueue(int value) {

```

```
        if(capacity==size)
            throw new Exception("queue is full");
        data[head] = value;
        head = (head+1)%capacity;
        size += 1;
    }
    public int Dequeue() {
        if(size==0)
            throw new Exception("queue is empty");
        int i = tail;
        tail = (tail+1)%capacity;
        size -= 1;
        return data[i];
    }
    public IEnumerator GetEnumerator() =>
        new MyQueueIterator(data, size, tail, capacity);

    private int[] data;
    private int size;
    private int capacity;
    private int head, tail;
}

public class MyQueueIterator : IEnumerator {
    public MyQueueIterator(int[] data, int size, int tail, int capacity) {
        this.data = data;
        this.tail = tail;
        this.capacity = capacity;
        this.last = size-1;
        this.index = -1;
    }
    public bool MoveNext() {
        if(index>=last) {
            Current = default(int);
            return false;
        }
        Current = data[(++index+tail)%capacity];
        return true;
    }
    public object Current { get; private set; }
    public void Reset() => index = -1;
    int last, tail, capacity, index;
    readonly int[] data;
```

```
}  
  
class Program {  
    static void Main() {  
        MyQueue q = new MyQueue(10);  
        q.Enqueue(3);  
        q.Enqueue(18);  
        q.Enqueue(-1);  
        // what follows is equivalent to foreach(...)  
        var iter = q.GetEnumerator();  
        while(iter.MoveNext())  
            Console.WriteLine(iter.Current);  
    }  
}
```

Initially, index equals `-1` and the iterator is positioned before the first element in the collection. By calling the `MoveNext` method, the iterator moves to the next element of the collection and the method returns `true` if the new position is a valid one otherwise, if `MoveNext` passes the end of the collection, the iterator is positioned after the last element and `MoveNext` returns `false`. When the iterator points to a valid position of the collection, `Current` returns the same object until `MoveNext` is invoked otherwise, `Current` is undefined and it returns, for example, the default value of the elements' type. `Reset` is used to rewind the iterator and, in case it is not implemented, we need to create a new enumerator to start a new visit. Last but not the least, the set of elements in the collection changes (i.e., adding, modifying, or deleting one or more elements) the behavior of the enumerator is undefined and usually it is not allowed: this let us iterate over the same collection in parallel because each iterator object contains its own iteration state.

For some collections, implementing an iterator class is an unnecessary burden. Instead of defining a new class, we can implement the iterator pattern directly in the collection as in the original `MyQueue` example. This can be done by using the `yield return` statement which works in this way: the expression following the statement is assigned to the `Current` property of the enumerator object. Then, the execution of the iteration is suspended, and it is resumed on the next call of the `MoveNext` method. Here is another (and more compact) example that can clarify the functioning of `yield`:

```
class Program {  
    static IEnumerable ThreeTimesTable() {  
        for (int i = 1; i<=10; i++)  
            yield return i*3;  
    }  
  
    static void Main() {  
        foreach (int n in ThreeTimesTable())
```



```

        Console.WriteLine(n);
    }
}

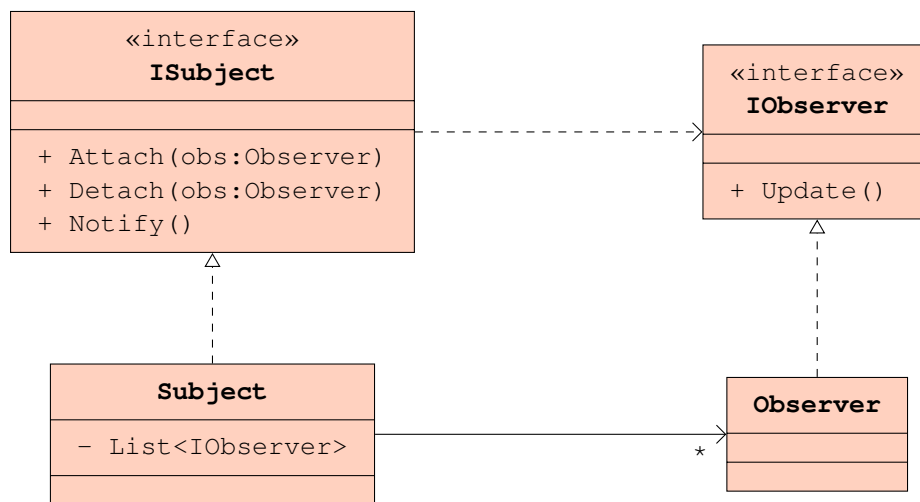
```

The method `ThreeTimesTable` returns the values of the  $3 \times$  table one by one so that the `Main` can print out each of them (or to do whatever is needed). At each iteration of the `foreach` loop, the execution of the loop in `ThreeTimesTable` is resumed and the next value is yielded until the `for` is over.

## 9.2 Observer

*behavioral design pattern*

Imagine having a system where an object, called *subject*, changes its state asynchronously and a set of one or more objects, called *observer*, reacting to such changes. A mechanism based on polling (i.e., iterating a test with a given frequency) turns out to be inefficient most of the time. As an alternative, the *observer* pattern is a behavioral design pattern based on a subscription mechanism where the subject directly notifies multiple objects about any events that happen to its state.



The `Subject` class implements the subscription methods `Attach` and `Detach` that let, respectively, an observer to join or leave the list of observers at runtime according on the behavior of the system. Through the `Notify` method, the `Subject` object calls the method `Update` of all registered observers.

The whole mechanism recalls the *events* seen in Section 5.9 that implement this design pattern as it is shown in the following example where an instance of `Timer` is used as subject for three different types of observing classes:

```
interface IObservable {
```



```
    void Update(Object source, ElapsedEventArgs e);
}

class WallClock : IObservable {
    public void Update(Object source, ElapsedEventArgs e) {
        Console.ForegroundColor = ConsoleColor.Blue;
        Console.WriteLine("It's {0:HH:mm:ss}!", e.SignalTime);
        Console.ResetColor();
    }
}

class TickCounter : IObservable {
    int Counter = 0;
    public void Update(Object source, ElapsedEventArgs e) {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("{0} ticks counted!", ++Counter);
        Console.ResetColor();
    }
}

class Countdown : IObservable {
    int Counter;
    public Countdown(int n) {
        Counter = n;
    }
    public void Update(Object source, ElapsedEventArgs e) {
        Console.ForegroundColor = ConsoleColor.Red;
        if(Counter>0)
            Console.WriteLine("-{0}", Counter--);
        else {
            Console.WriteLine("Go!");
            (source as System.Timers.Timer).Elapsed -= this.Update; //detach
        }
        Console.ResetColor();
    }
}

class Program {
    static void Main() {
        //create a subject
        var subject = new Timer();
        subject.Interval = 1000;
        subject.Enabled = true;
    }
}
```

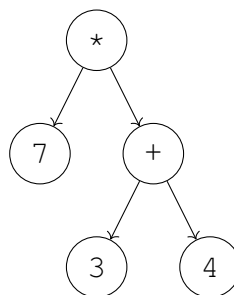
```
subject.AutoReset = true;
//create three observers
var WallClock = new WallClock();
var counter = new TickCounter();
var countdown = new Countdown(3);
//subscription
subject.Elapsed += WallClock.Update;
subject.Elapsed += counter.Update;
subject.Elapsed += countdown.Update;
//start
Console.WriteLine("Press Enter once to detach the TickCounter...");
Console.ReadLine();
subject.Elapsed -= counter.Update; //detach
Console.WriteLine("TickCounter detached!");
Console.WriteLine("Press Enter again to EXIT the application...");
Console.ReadLine();
subject.Stop();
subject.Dispose();
Console.WriteLine("Bye!");
}
```

Differently from the original diagram, the `Update` method in the interface `IObserver` has been modified to match the delegate definition and the operators `'+='` and `'-='` have replaced the methods `Attach` and `Detach` in the subscription mechanism implementation.

### 9.3 Visitor

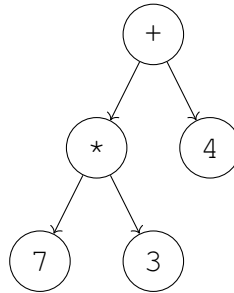
*behavioral design pattern*

This design pattern allows us to add further operations to objects without modifying their structure. To explain it, we will make use of the expression trees. They are binary trees representing mathematical expressions made of a finite well-formed combination of binary operations (internal nodes) and values (leaves), e.g., the formula  $7 * (3 + 4)$  is associated to the following tree:





It is worth noting the importance of parenthesis to disambiguate a formula. For example, removing the parenthesis from the previous one, the corresponding tree would be the following one which has a different result (since multiplication has higher priority than addition).



Expression trees are generated, for example, during the parsing phase of a program which is not a topic of this course. Now, we are interested in finding the “best” design to implement different types of visit for the target data structure (in this case the expression trees) by leaving the solution open to define new kinds of visits. The *visitor* design pattern has been designed specifically for these situations.

Assume that we need to visit the expression tree for evaluating its result. The naïve solution to this problem would be to add a specific method to each type of tree-node. For example, to evaluate the result of the expression tree we could write:

```
interface IExpression {
    int Evaluate();
}

class Literal : IExpression {
    public readonly int Value;
    public Literal(int Value) {
        this.Value = Value;
    }
    public int Evaluate() => Value;
}

class Mul : IExpression {
    public Mul(IExpression Left, IExpression Right) {
        this.Left = Left;
        this.Right = Right;
    }
    public int Evaluate() => Left.Evaluate()*Right.Evaluate();
    public readonly IExpression Left, Right;
}
```

```

}

class Add : IExpression {
    public Add(IExpression Left, IExpression Right) {
        this.Left = Left;
        this.Right = Right;
    }
    public int Evaluate() => Left.Evaluate()+Right.Evaluate();
    public readonly IExpression Left, Right;
}

class Program {
    static void Main() {
        IExpression t = new Mul(new Literal(7),
                                new Add(new Literal(3), new Literal(4)));
        Console.WriteLine("7*(3+4)="+t.Evaluate());
    }
}

```

Each class properly implements the interface `IExpression` defining the method `Evaluate` according to its semantic. In particular, the leaves (i.e., the `Literals` class) return their corresponding values while the internal nodes representing the binary operations (i.e., the classes `Add` and `Mul`) rely on the recursive evaluation of the left and right operands.

After some time, it becomes necessary to have a second type of visit for transforming the tree into a human-readable string (i.e., the initial formula). To implement the second type of visit, we need to add a further method to each class to traverse the tree and build the resulting string. This, however, breaks the open/closed principle. What the visitor pattern suggests is to extract the methods related to any visit from the tree classes and gather them into an ad-hoc class which takes an expression tree as input:

```

interface IVisitable {
    void Accept(IVisitor vtor);
}

interface IVisitor {
    void Visit(Literal elem);
    void Visit(Add elem);
    void Visit(Mul elem);
}

class Evaluator: IVisitor {
    public Evaluator(IVisitable t) {

```

```
s = new Stack<int>();
t.Accept(this);
}
public void Visit(Literal e) {
    s.Push(e.Value);
}
public void Visit(Add e) {
    e.Left.Accept(this);
    int left = s.Pop();
    e.Right.Accept(this);
    int right = s.Pop();
    s.Push(left+right);
}
public void Visit(Mul e) {
    e.Left.Accept(this);
    int left = s.Pop();
    e.Right.Accept(this);
    int right = s.Pop();
    s.Push(left*right);
}
public void Clear() => s.Clear();
public override string ToString() => s.Peek().ToString();
private Stack<int> s;
}

class Stringer: IVisitor {
    public Stringer(IVisitable t) {
        s = new StringBuilder();
        t.Accept(this);
    }
    public void Visit(Literal e) {
        s.Append(e.Value);
    }
    public void Visit(Add e) {
        s.Append("(");
        e.Left.Accept(this);
        s.Append("+");
        e.Right.Accept(this);
        s.Append(")");
    }
    public void Visit(Mul e) {
        e.Left.Accept(this);
        s.Append("*");
    }
}
```

```
        e.Right.Accept(this);
    }
    public void Clear() => s.Clear();
    public override string ToString() => s.ToString();
    private StringBuilder s;
}

class Literal : IVisitable {
    public readonly int Value;
    public Literal(int Value) {
        this.Value = Value;
    }
    public void Accept(IVisitor vtor) => vtor.Visit(this);
}

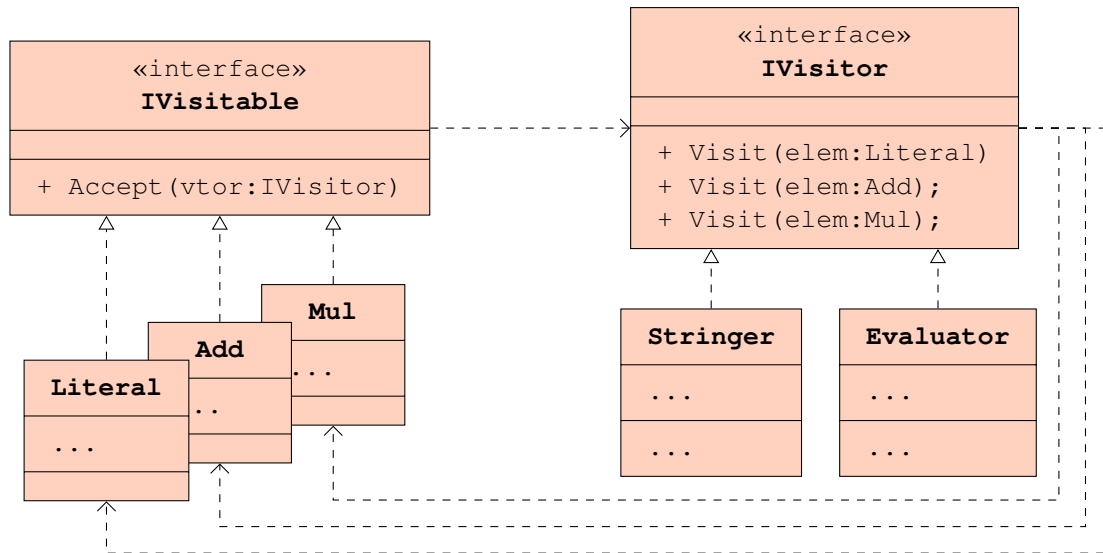
class Mul : IVisitable {
    public Mul(IVisitable Left, IVisitable Right) {
        this.Left = Left;
        this.Right = Right;
    }
    public void Accept(IVisitor vtor) => vtor.Visit(this);
    public readonly IVisitable Left, Right;
}

class Add : IVisitable {
    public Add(IVisitable Left, IVisitable Right) {
        this.Left = Left;
        this.Right = Right;
    }
    public void Accept(IVisitor vtor) => vtor.Visit(this);
    public readonly IVisitable Left, Right;
}

class Program {
    static void Main() {
        IVisitable t = new Mul(new Literal(7),
                               new Add(new Literal(3), new Literal(4)));
        Console.WriteLine(new Stringer(t)+"="+new Evaluator(t));
    }
}
```

This led to the definition of two interfaces: `IVisitor` and `IVisitable`. The first one forces any class implementing a visit algorithm to define a method that can process each type of node. The

second one is required to be implemented by each type of node to bind/dispatch the right method at runtime when the visit takes place. Here is a sketch of the class diagram representing how the visitor pattern has been applied in the previous example:



As final note, mind that adding a new type of element will require to modify all existing visitors to support that new element. This can become cumbersome in systems with many element types, as each visitor will need to be updated to handle every new element introduced.

## 9.4 Builder

*creational design pattern*

This is a creational design pattern aiming to construct objects consisting of multiple parts and having different configurations. Example:

```

public interface IHabitationBuilder {
    IHabitationBuilder AddFoundation();
    IHabitationBuilder AddWalls();
    IHabitationBuilder AddFixtures();
    IHabitationBuilder AddFurniture();
}

public class CastleBuilder : IHabitationBuilder {
    public IHabitationBuilder AddFoundation() {
        castle.Append("castle foundation: done\n");
        return this;
    }
    public IHabitationBuilder AddWalls() {

```

```
        castle.Append("brick-walls and archer-towers: done\n");
        return this;
    }
    public IHabitationBuilder AddFixtures() {
        castle.Append("gates and cross-windows: done\n");
        return this;
    }
    public IHabitationBuilder AddFurniture() {
        castle.Append("torture chamber: done\n");
        return this;
    }
    public override string ToString() => castle.ToString();
    StringBuilder castle = new StringBuilder();
}

public class CastleDirector : CastleBuilder {
    public CastleDirector() {
        this.AddFoundation();
        this.AddWalls();
        this.AddFixtures();
        this.AddFurniture();
    }
}

class Program {
    static void Main() {
        var c1 = new CastleBuilder()
            .AddWalls()
            .AddFoundation()
            .AddFurniture();
        Console.WriteLine("c1::\n"+c1);
        var c2 = new CastleDirector();
        Console.WriteLine("c2::\n"+c2);
    }
}
```

The example above shows the interface `IHabitationBuilder` which is required to build a generic habitation and how the concrete class builder `CastleBuilder` possibly implements it. The pattern can be extended with a director class (in this case `CastleDirector`) which checks the building steps and enforces their invocation order (e.g., `c1` got walls before foundation and it has no furniture).

## 9.5 Adapter

*structural design pattern*

It is a structural design pattern that makes it possible the cooperation between two objects that, otherwise, are not compatible (e.g., to adapt some legacy code and to work into a more recent system as well as to change the format of some data to make them readable for some other classes). An adapter object wraps some specific method calls of the adaptee object and makes them usable by another one. Example:

```
class FahrenheitThermometer {
    public double GetTemperature() {
        return rnd.NextDouble() * 400.0 - 60.0;
    }
    Random rnd = new Random();
}

public interface ITemperatureAdaptor {
    double GetTemperature();
}

class FahrenheitAdapter : ITemperatureAdaptor {
    public FahrenheitAdapter(FahrenheitThermometer adaptee) {
        Adaptee = adaptee;
    }
    public double GetTemperature() {
        return (Adaptee.GetTemperature() - 32.0) / 1.8;
    }
    FahrenheitThermometer Adaptee;
}

class Program {
    static void Main() {
        var a = new FahrenheitAdapter(new FahrenheitThermometer());
        double t = a.GetTemperature();
        if (t >= 100)
            Console.WriteLine("{0:0.00} DegC : water is boiling\n", t);
        else
            Console.WriteLine("{0:0.00} DegC : water is not boiling\n", t);
    }
}
```

The Adaptee attribute implements some useful behavior but its interface or format is not compatible with the target code, i.e., the thermometer returns a temperature in the Fahrenheit scale while the

Main function expects a temperature in the Celsius scale. The class `FahrenheitAdapter` fixes the issue.

## 9.6 Factory Method

*creational design pattern*

This pattern allows to encapsulate the creation process of different types of object based on specific parameters (referred as *products*) into a class (referred as *creator*). All products implement a common interface (or are derived from a common abstract class) which defines the methods that any product is required to implement. The creator class defines a creation method which invokes the constructor of the desired product with a specific configuration and then returns a reference of the products' interface. In the following example, we apply this pattern for creating different shapes where `IShape` represents the products' interface and the method `FactoryMethod` is responsible for creating a shape according to the type `t` required:

```
public interface IShape {
    double Area();
}

public class Square : IShape {
    public Square(int size) {
        Size = size;
    }
    public double Area() {
        return Size*Size;
    }
    int Size;
}

public class UnitCircle : IShape {
    public double Area() {
        return Math.PI;
    }
}

public class Creator {
    public IShape FactoryMethod(ProductType t) {
        if(t==ProductType.UnitSquare) return new Square(1);
        if(t==ProductType.Square4x4) return new Square(4);
        if(t==ProductType.UnitCircle) return new UnitCircle();
        throw new Exception("Unknown Product Type");
    }
}
```



```
public enum ProductType { UnitSquare, Square4x4, UnitCircle };  
}  
  
class Program {  
    static void Main(string[] args) {  
        IShape p;  
        var c = new Creator();  
        p = c.FactoryMethod(Creator.ProductType.Square4x4);  
        Console.WriteLine(p.Area());  
        p = c.FactoryMethod(Creator.ProductType.UnitCircle);  
        Console.WriteLine(p.Area());  
    }  
}
```

The previous implementation, however, can violate the open/closed principle (see Section 8) in case the set of products grows or if it is required to create a product with a new setting since the method `FactoryMethod` must be updated to accept the new products. The following example tries to overcome such an issue at the expense of defining a further interface, `IShapeCreator`, that unites the set of possible concrete constructors, i.e., one for each product that we need:

```
public interface IShape {  
    double Area();  
}  
  
public class Square : IShape {  
    public Square(int size) {  
        Size = size;  
    }  
    public double Area() {  
        return Size*Size;  
    }  
    int Size;  
}  
  
public class UnitCircle : IShape {  
    public double Area() {  
        return Math.PI;  
    }  
}  
  
public interface IShapeCreator {  
    IShape FactoryMethod();  
}
```

```

}

public class UnitSquareCreator : IShapeCreator {
    public IShape FactoryMethod() {return new Square(1); }
}

public class Square4x4Creator : IShapeCreator {
    public IShape FactoryMethod() { return new Square(4); }
}

public class UnitCircleCreator : IShapeCreator {
    public IShape FactoryMethod() { return new UnitCircle(); }
}

public class Creator {
    public IShape FactoryMethod(IShapeCreator creator) {
        return creator.FactoryMethod();
    }
}

class Program {
    static void Main(string[] args) {
        IShape p;
        var c = new Creator();
        p = c.FactoryMethod(new Square4x4Creator());
        Console.WriteLine(p.Area());
        p = c.FactoryMethod(new UnitCircleCreator());
        Console.WriteLine(p.Area());
    }
}

```

## 9.7 Strategy

*behavioral design pattern*

This design pattern turns a set of behaviors into objects and makes them interchangeable inside original hosting object (similarly to delegates, see Section 4.4). A given object (so-called context) has a reference to an object (so-called strategy) which defines how a certain action should be performed. In this way, the set of strategies can be extended as well as the strategy to be applied can be set at runtime without any modification to the context implementation. Example:

```

public interface IStrategy {
    void Sort(int[] array);
}

```



```
}

public class Context {
    public Context(int n) {
        System.Random rnd = new System.Random();
        data = new int[n];
        for (int i=0; i<n; ++i)
            data[i] = rnd.Next(0,100);
    }
    public void SetStrategy(IStrategy a) {
        strategy = a;
    }
    public void Sort() {
        strategy.Sort(data);
    }
    IStrategy strategy;
    int[] data;
}

public class HeapSort : IStrategy { // unstable, O(n log n)
    public void Sort(int[] arr) {
        for (int i = arr.Length/2-1; i>=0; --i)
            Heapify(arr, arr.Length, i);
        for (int i = arr.Length-1; i>=0; --i) {
            Swap(arr, i, 0);
            Heapify(arr, i, 0);
        }
    }
    void Heapify(int[] arr, int n, int i) {
        int max = i, left = 2*i+1, right = 2*i+2;
        if (left<n && arr[left]>arr[max]) max = left;
        if (right<n && arr[right]>arr[max]) max = right;
        if (max!=i) {
            Swap(arr, i, max);
            Heapify(arr, n, max);
        }
    }
    void Swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

```

public class InsertionSort : IStrategy{ // stable, O(n^2)
    public void Sort(int[] arr) {
        for (int i=1; i<arr.Length; ++i) {
            int j=i-1, key = arr[i];
            while(j>=0 && arr[j]>key) arr[j+1] = arr[j--];
            arr[j+1] = key;
        }
    }
}

public class Program {
    static void Main() {
        var context = new Context(20);
        context.SetStrategy(new HeapSort());
        context.Sort();
    }
}

```

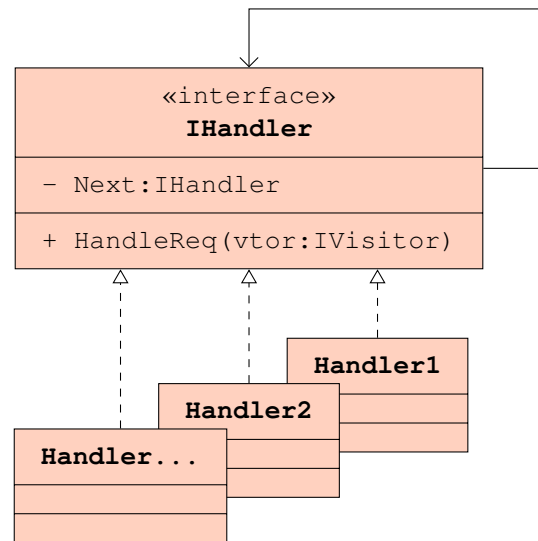
In the example above, `Context` is a class with an array to be sorted and `IStrategy` is the interface that links the context to any strategy, i.e., it is required that a strategy implements such an interface to be used. Sorting is a very wide topic, and numerous algorithms are available in literature, each with different properties<sup>3</sup>. The example offers two strategies: a stable sorting algorithm with quadratic time complexity and an unstable one with log-linear time complexity<sup>3</sup>... And more strategies can be added on demand. Sometimes, delegates can implement this pattern more elegantly, e.g., if all strategies consist of just one function, but in this case `HeapSort` has three methods and defining a class for each strategy is required.

## 9.8 Chain of Responsibility

*behavioral design pattern*

This design pattern gives us the possibility to dynamically split the processing of a request into a sequence of distinct steps performed by a set of objects (so-called *handlers*) like the stages of a pipeline. After processing a request, each handler may decide to pass it to the next one or interrupt the processing. This approach is the same used for exception management, where an exception is forwarded to the next (and more generic) `catch` block until a matching one is found (see Section 5.6). The implementation uses recursive composition to chain whatever number of handlers is needed:

<sup>3</sup> A sorting algorithm is stable if any pair of input elements with equal keys keeps the same order after sorting. This is particularly useful for implementing multiple-keys sorting procedures.



## 9.9 Singleton

*creational design pattern*

Singleton design pattern ensures that a class can have at most only one instance providing a global reference which makes it accessible from anywhere in the program (i.e., once it has been instantiated, if try to you create a new one, you get instead a reference to the one already existing). This pattern is useful, for example, when different objects need to access the same shared resource as a file. Note that on several online guidelines this patterns is referred as antipattern because it may hide the dependencies in your code and increase the coupling of your solution.

The implementation can differ depending on the need of the application. For example, a class that fulfill the previous requirements can be coded as a static class. However, such a solution prevents the class from implementing an interface or be inherited from another class as well as to be passed as parameter in a method. Hence, we introduce the following alternative:

```

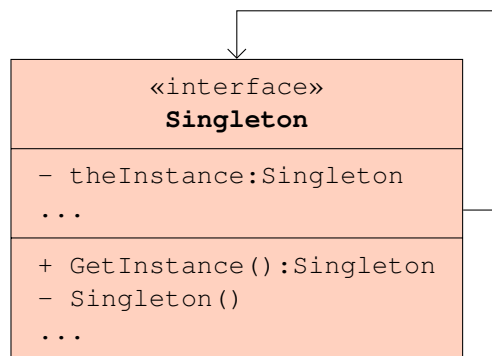
public class Singleton {
    Singleton() {}
    static Singleton theInstance;
    public static Singleton GetInstance() {
        if (theInstance==null)
            theInstance = new Singleton();
        return theInstance;
    }
    public int Counter {
        get { return ++counter; }
    }
    int counter = 0;
}
  
```

```

class Program {
    static void Main(string[] args) {
        Singleton singleton_one = Singleton.GetInstance();
        Singleton singleton_two = Singleton.GetInstance();
        Console.WriteLine(singleton_one.Counter);
        Console.WriteLine(singleton_two.Counter);
    }
}

```

The Singleton class declares the static method `GetInstance` that, on the first call when the reference `theInstance` is still null, creates a private static instance of the class then returns it any time the method is invoked. The constructor is declared private to prevent direct creation of an instance. The related class diagram can be schematized as it follows:



## 9.10 Prototype

*creational design pattern*

Prototype is a creational design pattern that allows cloning a given object even if the related class is unknown and part of its attributes are private (given that same-class objects can access them). This pattern is also important to highlight some aspects that must be considered when it is implemented or in situations like those shown in Section 5.2. Example:

```

public class Square {
    public Square(Point pos, double size) {
        Position = pos;
        Size = size;
    }
    public Square ShallowCopy() {
        return (Square) this.MemberwiseClone();
    }
    public Square DeepCopy() {

```



```
        Square clone = (Square) this.MemberwiseClone();
        clone.Position = new Point(Position.X, Position.Y);
        return clone;
    }
    public override string ToString() {
        return Size+"x"+Size+" @" +Position;
    }
    public void Scale(double f) {
        Size*=f;
    }
    public void Shift(int dX, int dY) {
        Position.X += dX;
        Position.Y += dY;
    }
    double Size;
    Point Position;
}

public class Point {
    public Point(int X, int Y) {
        this.X = X;
        this.Y = Y;
    }
    public override string ToString() => " (" +X+" , "+Y+" ) ";
    public int X, Y;
}

class Program {
    static void Main(string[] args) {
        Square s1 = new Square(new Point(1,1), 2);
        var s2 = s1.ShallowCopy();
        var s3 = s1.DeepCopy();
        // change s1 and display all objects
        s1.Scale(2);
        s1.Shift(-1,-1);
        Console.WriteLine(s1);
        Console.WriteLine(s2);
        Console.WriteLine(s3);
    }
}
```

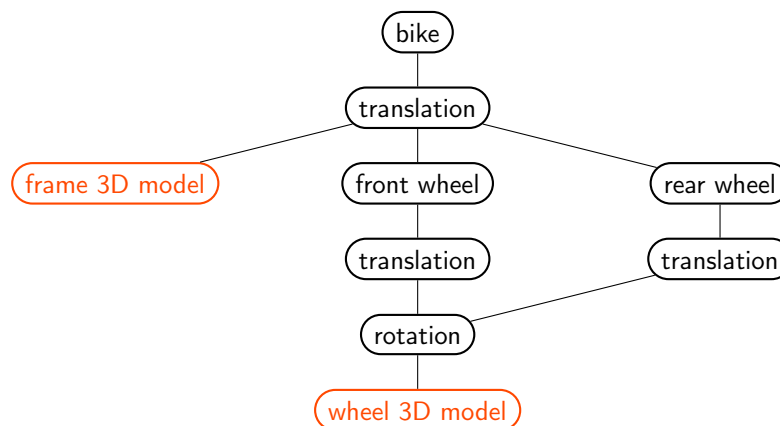
The `MemberwiseClone` method creates a shallow copy by instantiating a new object then copying the non-static attributes of the current object into the new one. It is worth nothing that if an attribute

is a value-type, e.g., `Size`, any change to one of them does not affect the other. However, if the attribute is a reference-type (e.g., `Position`), just the reference is copied so that the original object and its clone refer to the same object. Therefore, we created two cloning methods, i.e., `DeepCopy` and `ShallowCopy`, having different behaviors.

## 9.11 Flyweight

*structural design pattern*

The Flyweight pattern is applied in those cases where separate objects containing plenty of common data and you intend to optimize the memory usage and improve performance by allowing objects to share the common parts of their state, rather than storing all the data within each individual object. The same approach is used in graphics game development where a *scene graph* is used to replicate the same “heavy” object (like a complex 3D model) in different part of the scene. To manage all the copies efficiently and minimize memory usage, complex 3D models are represented only once as leaves of a tree-like data structure (it’s actually a graph, from which the name) while the unique information normally used to control animations are stored separately in the internal nodes. Finally, the tree is traversed for rendering the scene. For example, to render the animation of a moving bicycle, the wheels are the same 3D model performing the same rotation in each frame, but they are placed in different positions:



The Flyweight pattern is usually combined to a factory (see Section 9.6) that manages the shared objects and ensures that if one of them already exists, it is reused:

```

public struct Point3D {
    public Point3D(int x, int y, int z) { X=x; Y=y; Z=z; }
    public override string ToString() => "(" + X + ", " + Y + ", " + Z + ") ";
    public readonly int X, Y, Z;
}
  
```



```
public class Model3D {
    public Model3D(string URI) {
        Console.WriteLine($"Load the 3D model from {URI} into Mesh");
        this.URI = URI;
    }
    public void Draw(Point3D Position) {
        Console.WriteLine($"Render the 3D model from {URI} at {Position}");
    }
    Point3D[] Mesh;
    string URI;
}

public class FlyweightModel3DFactory {
    public FlyweightModel3DFactory() {
        ModelCache = new Dictionary<string, Model3D> ();
    }
    public Model3D GetModel3D(string URI) {
        if(!ModelCache.ContainsKey(URI))
            ModelCache[URI] = new Model3D(URI);
        return ModelCache[URI];
    }
    Dictionary<string, Model3D> ModelCache;
}

class Program {
    static void Main(string[] args) {
        var factory = new FlyweightModel3DFactory();
        Model3D boulder1 = factory.GetModel3D("/models/boulder.3ds");
        Model3D boulder2 = factory.GetModel3D("/models/boulder.3ds");
        Model3D tree1 = factory.GetModel3D("/models/pine.3ds");
        boulder1.Draw(new Point3D(0,23,14));
        boulder2.Draw(new Point3D(10,8,42));
        tree1.Draw(new Point3D(30,38,5));
    }
}
```

In the example above, the factory `FlyweightModel3DFactory` is used to get an instance of different 3D models. The factory recognizes a duplicate by means of a dictionary that uses the URI string associated to the 3D model source file as key. In case the 3D model does not exist in the cache then it is loaded from the source file into memory and a reference to the 3D model is finally returned. The client renders all the instances of `Model3D` at the position passed to the method `Draw`. As a final note, the consistency of the shared data should be guaranteed, e.g., making them immutable.

## References

- [1] Hansen, H. *"Connections between the Software Crisis and Object-Oriented Programming."* Cornell University, Science & Technology Studies, SIGCIS History of Computing Workshop in Memory of Michael S. Mahoney. <https://www.sigcis.org/files/Hsu%20--%20Software%20Crisis%20and%20OOP.pdf>
- [2] Arlow, J. and Neustadt, I. *"UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design"* Addison-Wesley Object Technology Series (2005).
- [3] Stevens, Wayne P., Glenford J. Myers, and Larry L. Constantine. *"Structured design"* IBM systems journal 13.2 (1974): pp. 115-139.
- [4] Martin, Robert C. *"Agile Software Development, Principles, Patterns, and Practices"* Prentice Hall, ISBN 978-0135974445 (2003): p. 95.