Software Design and Algorithms



Object-Oriented Programming and Object-Oriented Design Introduction

EPAM Systems Inc.

Learn & Development

Software Design and Algorithms

[www.epam.com](http://www.epam.com/)

Contents

[Object-Oriented Programming and Object-Oriented Design Introduction 1](#_Toc64995348)

[1. PROGRAMMING PARADIGM 3](#_Toc64995349)

[1.1. DECLARATIVE PROGRAMMING 3](#_Toc64995350)

[1.2. Imperative Programming 4](#_Toc64995351)

[1.3. OBJECT-ORIENTED PROGRAMMING 6](#_Toc64995352)

[1.4. FUNCTIONAL PROGRAMMING 6](#_Toc64995353)

[2. Unified modeling language 9](#_Toc64995354)

[2.1. What is UML 9](#_Toc64995355)

[2.2. Assosiasion 11](#_Toc64995356)

[2.3. Inheritance 12](#_Toc64995357)

[2.4. Composition and Aggregation 13](#_Toc64995358)

[3. Principles of Oop 15](#_Toc64995359)

[3.1. abstraction 15](#_Toc64995360)

[3.2. Encapsulation 19](#_Toc64995361)

[3.3. INHERITANCE 22](#_Toc64995362)

[3.4. POLYMORPHISM 24](#_Toc64995363)

[4. Object-oriented design Introduction 27](#_Toc64995364)

[4.1. Design smells 27](#_Toc64995365)

[4.2. What is design? 29](#_Toc64995366)

[4.3. WHY CHANGE IS HARD AND THE PROBLEM DESIGN SOLVES 30](#_Toc64995367)

[4.4. THE PURPOSE OF OOD 31](#_Toc64995368)

[4.5. THE TOOLS OF DESIGN 31](#_Toc64995369)

# PROGRAMMING PARADIGM

This lecture is going to be dedicated to object-oriented programming and object-oriented design. It is going to be an overview lecture of what you are going to learn in this course, but I recommend you pay close attention to it. The reason being, it is going to include a lot of basics and principles that will help you write better, cleaner, more maintainable code.

During development, you probably asked yourself questions like: “how do I write this class, to make it maintainable?”, “how do I design my system so that when requirements change, we don’t have to rewrite it from ground up?”. Principles of object-oriented programming might not give you an exact answer to those questions, but they will help you along the way of discovery of “good” design.

Before we dive deeper, let us go back in time for a little bit. In 1938 Alan Turing was the one to start it all. He established what, in future will be called “computer programming”. Since then, a lot of breakthroughs have happened. One of them was creation of different programming languages. First, came assembly language, in 1951 Grace Hopper created a first compiler. After that, languages were created one by one: COBOL, Lisp, C, C++, Java, Pascal etc. This explosion of languages has inevitably led to creation of programming paradigms.

A programming paradigm is sort of a programming style. It is not bound to a certain programming language. It provides us with means and structure for execution of a program. It is a complex of concepts, instruments, principles that define the fundamentals of programming style.

We are going to touch upon 4 of the most recognized paradigms. Although [some sources suggest](https://wiki.c2.com/?ThereAreExactlyThreeParadigms) that there are only 3 of them.

## DECLARATIVE PROGRAMMING

* In declarative programming, the programmer instructs the computer on what is to be computed.
* You do not know how it works, but you know what it does.
* It is a very good idea to decouple DOM manipulation from app logic. This improves the testability of code.
* Improves readability of a code

First programming paradigm we are going to talk about is **Declarative Programming**. In this paradigm we must define the specification for solving a task, i.e., we describe what the problem’s field and how what kind of a result we expect. The order of execution and the method of achieving results does not matter. A common example of declarative programming is HTML. It describes the contents of the page, but not the way it should be rendered.

select upper(name)

from people

where length(name) > 5

order by name

Listing 1.1 – Example of declarative code

On the example above, you can see an SQL query. It is also an example of declarative programming. Here, we describe the context of the problem, a list of people and what we want to get from that list: a list of names, uppercased, where the length of the name is larger than 5. The list of names should be ordered in ascending order. We do not care what type of database we are using. We do not describe how to retrieve the data or how to go through each entry of people list. Whether the application should use quick sort or merge sort. It is all up for the application or interpreter to decide.

Next paradigm is **Imperative Programming.**

## Imperative Programming

* The script is basically telling the computer how to do something.
* Imperative phrases which change the global state of a program"
* Not scalable

With imperative programming, we describe the system as a process of execution of instructions that change the state of the system. It is commonly considered to be less extensible than the others. Examples of languages with imperative programming support are C++, C, Go, JavaScript. You should always remember that you cannot always categorize a certain language to a single programming paradigm. Usually, the languages support 2 or even 3 paradigms at the same time. For example, JavaScript supports imperative, functional, and object-oriented paradigms at the same time.

    result = []

    i = 0

start:

    numPeople = length(people)

    if i >= numPeople goto finished

    p = people[i]

    nameLength = length(p.name)

    if nameLength <= 5 goto nextOne

    upperName = toUpper(p.name)

    addToList(result, upperName)

nextOne:

    i = i + 1

    goto start

finished:

    return sort(result)

Listing 1.2 – Example of imperative code

Above you can see an example of code written in imperative style. We precisely describe the sequence of actions to achieve the desired result. To get the expected result we must know how to achieve it. We know exactly in which order should those instructions be arranged in. Opposite to the previous example from declarative programming, here you can see that we have variables with state. One variable has an Integer type, the other is an array. To achieve the result, we must add conditions, loops, gotos.

With imperative programming you know exactly what is happening. You can dial into the execution of a program and easily debug it. While with declarative, the exact path of a program is not deterministic from the perspective of executed code.

Both approaches have their place in a programming with their strengths and weaknesses. It is also worth mentioning that all code is imperative in the end, when it is executed by the processor.

## OBJECT-ORIENTED PROGRAMMING

* Program is defined by object which combine state and behavior
* Good for structured and modular code
* Well suitable for big projects

Object-oriented paradigm describes the computer program as a set of specific objects, that are instances of a class. The objects communicate by sending, receiving, and processing the messages. The messages may include parameters. The objects have state, which they can change when processing a message. Objects may create other objects or may send messages when processing a message.

It is safe to say that object-oriented paradigm is well-suited for big projects that require having a state within the application. It is message-oriented approach provides a way for objects to be replaced by other objects of a same type. This means that the behavior of a program can be changed just by replacing an object. Also, same objects as building blocks could be reused in other parts of the same system.

result = []

for p in people {

    if p.name.length > 5 {

        result.add(p.name.toUpper);

    }

}

return result.sort;

Listing 1.3 – Example of object-oriented code

## FUNCTIONAL PROGRAMMING

* Treats computations as the evaluation of functions and avoids changing state and mutable data
* Eliminating side effects, i.e., changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program
* Emphasize using of immutable data

Last, but not least, **Functional paradigm** describes the program as a set of functions (not objects or procedures) that are used as building blocks to manipulate data. It forces us to use function in their mathematical sense, as they just declare a relationship between two entities. Functions do not change the state of a program (also known as *pure*), they simply pipe the data through them to produce a result. There are no variables, only constants.

sort(

    filter(λs. length s > 5,

        map(λp. to\_upper(name p),

            people)))

Listing 1.3 – Example of functional code

In the example we have a certain nesting of certain functions, where the result of the function can be expressed as a list of arguments that are passed to this function. However, arguments can also be functions. By the way, not all programming languages can pass a function as a parameter to another function, this is the so-called first-class citizen function. JavaScript has this capability, that is why it has some concepts of functional programming.

A program in functional programming is not represented by a specific state, but by a combination of function calls at a certain point in time.

Functional programming pushes us towards the idea that it would be nice to use immutable data. Since immutable data is faster, because we put them into memory once, and they do not change. There are disadvantages and advantages to this, but this is how it works in functional programming.

There are so-called pure functions in functional programming, that is, functions without side effects. When you have a list of parameters as function arguments, a certain logic for handling them and a certain result in return. The function does not change data and state outside of its scope. In fact, if you change a variable outside the scope of the function, this already indicates that the function is impure, that is, it has a side effect. This is neither bad nor good, these are just different approaches. Functional programming assumes that the functions are just pure and should follow the principle of single responsibility, that is, each function should have single responsibility. Because the essence of functional programming is in the combination of various functions with different responsibilities to achieve the result.

# Unified modeling language

## What is UML

UML is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

It is logical to assume that before starting to write code, we need to design our computer system. If the system (program) is small, for example some of your pet-project, then there is no need for visual design, because all the components of this system and their interconnections, in principle, you can keep in your head if you are writing this project alone. As soon as several people appear on the project and information needs to be somehow shared, as soon as the project grows, as soon as the number of components on the project becomes more than ten, you have a certain need for some plan, a representation of your system.

There is a language called UML or Unified Modeling Language for describing system components and their interrelationships. In fact, it gives us a list of terms, abstractions, concepts, and tools for high-level modeling of our system. In principle, the UML is to some extent design patterns in the programming world. When I say that in this part of the system, I need to apply a class that has only one instance to share its state between other components of the system - it will take too long, and we all know perfectly well what it is called a singleton. That is, we have some tools, some concepts, in terms of which we can discuss not some low-level details, but higher-level abstractions. The use of UML is not limited to programming, it is also used for modeling business processes, systems design, drawing up the organization structure, and so on.

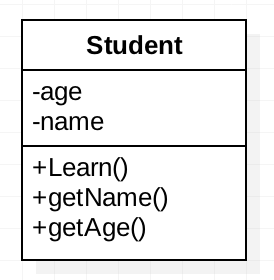


Figure 2.1 – UML diagram of Student class.

As you can see on the Figure 2.1, there is nothing complicated in the UML. Here is a diagram of the Student class, where at the top you see the name of the class, below is a list of properties that are inherent in this class, even below its behavior, that is, the methods that this class has.

Usually, the system is not limited to just one class, as we have already mentioned, there are tens, hundreds of classes, or even more. Naturally, all these classes somehow interact with each other, somehow communicate, send messages to each other, call each other's methods, send events, and so on. The visual representation of the classes and the relationship between these classes is called class diagrams. There are a lot of relationships, some of the most basic ones are shown in the figure 2.2.

Let us take a closer look at the relationship between the classes.



Figure 2.2 – UML Relations.

## ASSOCIATION

There are different types of relationship between two classes / objects. The most basic type of relationship is association, which means that the two classes are somehow related to each other, and we do not yet know exactly how this relationship is expressed and are going to clarify it in the future. This usually happens in the early stages of system design, when we know that there is a relationship, but what specific relationship - inheritance, composition, or something else is not yet clear to us. We are designing the system more globally. The association helps us a lot when we indicate that one class in some way interact with another class. At the initial stage, this is enough for us. Further, of course, this will be clarified. Why is this a directional association? – Because the arrow shows us that we have a component that uses another component. In this case the CustomService uses the CustomRepository component, and not vice versa.

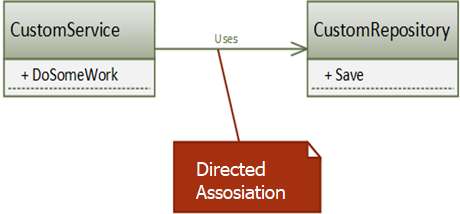


Figure 2.3 – Directed association.

An association is a relationship in which objects of one type are somehow related to objects of another type. For example, an object of one type contains or somehow uses an object of another type. The player plays in a team. We do not yet know what kind of relationship they have, or we are not interested in it at this stage of the design. But we know that there is a relationship.

## Inheritance

A more precise type of relationship is the public inheritance relationship (IS A Relationship), which says that everything that is true for the base class is true for its successor. With its help we can get polymorphic behavior, abstract from the concrete implementation of classes, dealing only with abstractions (interfaces or base classes) and do not pay attention to implementation details.

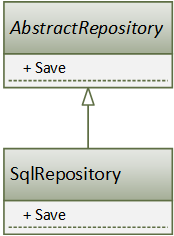


Figure 2.3 – Inheritance relationship.

Although inheritance is a great tool in the hands of any OOP programmer, it is clearly not enough for solving all types of problems. Firstly, not all relationships between classes are defined by the "is a" relationship, and secondly, inheritance is the strongest relationship between two classes that cannot be broken at runtime (this relationship is static and, in strongly typed languages, is determined at compile time).

That is, JavaScript, as we all know, has prototypal inheritance and indeed it can be changed for inherited classes, properties can be changed, you can just change prototypes. But this is more the exception than the rule. Because you cannot do this in classical inheritance: once inherited in the source code, at run time you will not break this connection and you will not change the base class. That is why it is believed that inheritance is the strongest relationship between objects. That is why architects and system designers recommend using inheritance only when it is necessary. I think you have heard the concept of preferring composition over inheritance, this suggests that composition can be broken at run time, and you can replace one object in the composition at runtime with another, change the behavior dynamically. You cannot do this with the inheritance.

## Composition and Aggregation

When relationships between components go beyond inheritance, relationships such as composition and aggregation come to our rescue. They both model a HAS-A Relationship and are usually expressed in that the class of a whole contains the fields (or properties) of its constituent parts. The line between the concepts is thing, but important, especially in the context of dependency management. We will also talk about dependency management a little later, when we touch on the topic of object-oriented design, because dependency management is one of the tools of object-oriented design.

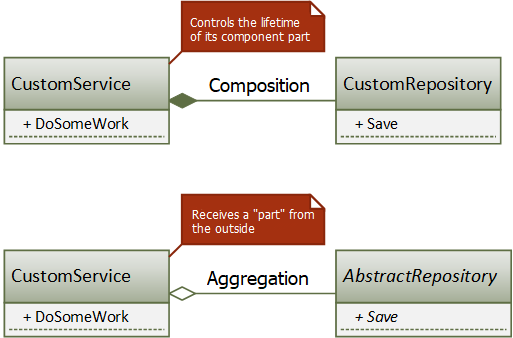


Figure 2.4 – Composition and Aggregation.

A couple of points to make it easier to remember the visual notation:

1. the diamond is always on the side of the whole, and the simple line is on the side of the component.
2. a filled rhombus indicates a stronger bond - composition, an open rhombus indicates a weaker bond - aggregation.

The difference between composition and aggregation is that in the case of composition, the whole explicitly controls the lifetime of its component part (the part does not exist without the whole), and in the case of aggregation, although the whole contains its component part, their lifetimes are not related (for example, the component part is passed via constructor parameters). Listing 2.1.

CompositeCustomService uses composition to manage its constituent parts, and AggregatedCustomService uses aggregation. In this case, explicit control of the lifetime usually leads to a higher coupling between the whole and the part, since a specific type is used that closely connects the participants with each other.

class CompositeCustomService {

  // Composition

  private readonly repository: CustomRepository = new CustomRepository();

  public doSomething() {

    // Usage of repository

  }

}

class AggregatedCustomService {

  // Aggregation

  private readonly repository: AbstractRepository;

  constructor(repository: AbstractRepository) {

    this.repository = repository;

  }

  public doSomething() {

     // Usage of repository

  }

}

Listing 2.1 – Example of Composition and Aggregation.

Another example of composition. Let us say a bicycle is a whole part and its components (shock absorbers, wheels, handlebars) are parts. In fact, a single shock absorber without a bike makes no sense. An example of aggregation. Suppose there is a university or a school as a whole and teachers, professors as parts, for a certain period they may be part of this university, in some period they may not be included. They can exist without this university after the university is destroyed, that is, its lifetime is over.

# Principles of Oop

We are done with the introductory part, let us move on to the principles of object-oriented programming. We will consider four basic principles of OOP on which the object-oriented concept, object-oriented programming is based: abstraction, polymorphism, encapsulation, and inheritance. In some sources, more concepts are highlighted, for example, Wikipedia still refers to the fundamental concepts of object-oriented programming such concepts as class and object. But these four concepts are the classics on which the concept of object-oriented programming is based.

## abstraction

Abstraction in object-oriented programming means the highlighting of some significant parts, meaningful information from a component, no matter whether it is a class or an architectural layer in the system, or a logical unit of our system. In general, the highlighting of significant parts or exclusion of insignificant parts from consideration. In OOP, only data abstraction is considered, usually it simply called as "abstraction" implying a set of the most significant characteristics of an object available for the program. Abstraction is essential when dealing with system complexity by hiding implementation details and highlighting essential aspects of behavior. The main idea of abstraction is to describe real life objects and how they interact in a software system. Abstraction can be implemented using interfaces and abstract classes.

new Promise((resolve, reject) => {

  // asynchronous code

})

  .then((result) => {

    // handling the result

  })

  .catch((error) => {

    // handling an error

  });

Listing 3.1 – Example of Promise object in JavaScript.

An example of abstraction is a JavaScript class Promise (Listing 3.1), which has a well-defined public interface (then, catch, all methods) while the client (the client is not a browser, but the calling program, the component that uses the Promise) works exclusively with meaningful behavior, public interface. It knows that by calling the then method he will receive the data, and by calling the catch method he will handle the error and so on. In this case, it does not matter at all how exactly the Promise object performs its work with asynchronous actions. Thus, all behavior that is insignificant to the client is hidden behind the abstraction. And if the logic of work inside the Promise object itself changes, the client will not change accordingly, because it does not depend on this behavior. The open part means the abstraction, that is, the interface should be the most stable, since a certain number of clients depend on it. Accordingly, the more the number of clients does depend on, the more stable it should be, because when the interface changes, there will be a cascading change in all clients.

The hidden part, that is, the implementation details, is the more unstable part of the abstraction, it can change while maintaining the public interface.

enum CoffeeSelection {

  FILTER\_COFFEE,

  ESPRESSO,

  CAPPUCCINO,

}

class CoffeeBean {

  // implementation of CoffeeBeen

}

class Coffee {

  constructor(selection: CoffeeSelection, volume: number) {

    // implementation of Coffee

  }

}

class Configuration {

  constructor(weight: number, volume: number) {

    // implementation of Configuration

  }

}

class CoffeeMachine {

  private configMap: Map<CoffeeSelection, Configuration>;

  private beans: Map<CoffeeSelection, CoffeeBean>;

  constructor(beans: Map<CoffeeSelection, CoffeeBean>) {

    this.beans = beans;

    // create coffee configuration

    this.configMap = new Map<CoffeeSelection, Configuration>();

    this.configMap.set(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

    this.configMap.set(

      CoffeeSelection.FILTER\_COFFEE,

      new Configuration(30, 480)

    );

  }

  public brewCoffee(selection: CoffeeSelection): Coffee {

    const coffee = new Coffee(selection, 100);

    console.log("Making coffee...");

    return coffee;

  }

}

Listing 3.3 – Abstraction example – Class CoffeeMachine with public method brewCoffee.

const main = () => {

  // create a |Map of available coffee beans

  const beans = new Map<CoffeeSelection, CoffeeBean>();

  beans.set(

    CoffeeSelection.ESPRESSO,

    new CoffeeBean("My favorite espresso bean", 1000)

  );

  beans.set(

    CoffeeSelection.FILTER\_COFFEE,

    new CoffeeBean("My favorite filter coffee bean", 1000)

  );

  // get a new CoffeeMachine object

  const machine = new CoffeeMachine(beans);

  // brew a fresh coffee

  const espresso: Coffee = machine.brewCoffee(CoffeeSelection.ESPRESSO);

};

Listing 3.3 – Abstraction example – Using class CoffeeMachine to brew coffee.

Well, you can see that we have a certain abstraction called a CoffeeMachine, it is presented on the Listing 3.3. A coffee machine is a kind of abstraction, in which there are some implementation details, that is, not significant behavior: configuration, configMap, working with different coffee beans, it knows how to brew espresso, it knows how to brew filtered coffee - all this is not important for the client. The client knows that the coffee machine has one and only public method (interface method) brewCoffee and the client depends only on this method. It is important for the client to know that he needs to create an instance of the CoffeeMachine class, pass coffee beans there and call the brewCoffee method.

The CoffeeMachine is an abstraction, and the method brewCoffee is a significant behavior that we have highlighted in this abstraction. Everything else, all the settings of the coffee machine, initialization in the constructor of some configuration of everything else - for the client this is not meaningful behavior, it should not depend on it, because it is unstable, it can change. The implementation details are changed systematically, the requirements are changed systematically, the main thing is not to change the public interface on which clients depend.

## Encapsulation

If abstraction allows us to highlight the essential behavior of our component, the essential aspects of its behavior, then encapsulation is the tool that helps to hide unimportant implementation details out of sight. In the design field, there are two concepts - encapsulation and data hiding, information hiding. Encapsulation is commonly used in the context of information hiding. Public mutable data violates encapsulation, because in this case any client of the class can change the internal state of the class object without the notification of the class. To achieve encapsulation in the design, two components are distinguished, the two parts of the class that were mentioned earlier are the public part, its public interface, and the private part, not meaningful to the client behavior, implementation details. At the same time, the class interface that has encapsulation should not just take and duplicate the property of this class through accessors (getters and setters), it should provide abstract interface, a higher-level one that is needed by the client. In other words, the public part should expose more about what the class does and hide unnecessary implementation details from clients. Abstraction and encapsulation complement each other and form some more general holistic picture of the object-oriented programming paradigm.

class Paystub {

  private readonly \_employees: Array<Employee>;

  public getEmployees(): Array<Employee> {

    return this.\_employees;

  }

  public computePayroll(): number {

    // using this.\_employees for calculation

    return 42;

  }

}

const p1 = new Paystub();

const employees = p1.getEmployees();

employees.push(new Employee());

employees.push(new Employee());

p1.computePayroll();

Listing 3.4 – Encapsulation is violated.

class Paystub2 {

  private readonly \_employees: Array<Employee>;

  public addEmployee(employee: Employee): void {

    this.\_employees.push(employee);

  }

  public computePayroll(): number {

    // using this.\_employees for calculation

    return 42;

  }

}

const p2 = new Paystub2();

const employee = p2.getEmployees();

p2.addEmployee(new Employee());

p2.addEmployee(new Employee());

p2.computePayroll();

Listing 3.5 – Encapsulation is not violated.

Example. Here we have two classes. The first class on the Listing 3.4 is Paystub, the other class is Paysbub2 (Listing 3.5). In both examples, we have the private data \_employees, which is an array that can contain a certain number of instances of the Employee class. In the example on the Listing 3.4, the getEmployees method returns this.\_employees array so the client can access and modify the data (add and remove items). In the second example, the designer of our system, in which this class appears, made a certain assumption that the client would be able to add Employee to Paysbub2, in what way - the client should not be interested. It should not depend on implementation details, how exactly Employees will be added to the Paysbub2 class, it should only depend on the abstract public interface, that is, on the addEmployees method. It should be enough for the client to call the addEmployees method with instances of the Employee class, it no longer depends on the data structure, an object or a linked list can be used instead of an array. In the second case, the client will not break, everything will work as before, only the implementation of the addEmployees method will change. In the first case, everything will break, because the client knows that this is an array, he works with this data as with an array, when we change it to an object and we cannot push data there, if you will need to add new records there - the client will break. This is bad, because there can be not one, but a thousand such clients, and the half of the system will collapse because this system depends on the internal implementation in which there is pseudo encapsulation.

As mentioned earlier, class members do not just have to be duplicated through setters and getters, and getEmployees is, in fact, a regular getter. And the abstraction should provide some high-level interface for working with itself.

Another example to show that hiding data is not just about creating private class members and working with them. Suppose we have a server. You are writing an application from scratch. Server is written on NodeJS and some configuration data is required on the backend. Suppose these are passwords for connecting to a database or some sort of sensitive information. All this data is stored in the config.json file located somewhere on your server, in some directory. And then we have 2 options:

* You can spread the logic of working with the configuration throughout the application in such a way that any component that needs a part of the configuration will just read this config.json file, parse it, and so on.
* The client is agnostic to the location and the configuration format. And access is provided through the ConfigurationProvider class or module, which encapsulated the logic for working with configuration, it would be able to parse it and return data. And ConfigurationProvider will be provided through dependency injection to the components where it is needed.

The second case is also an encapsulation, in which we hide the entire management of the configuration, and if something changes (for example, the data storage format changes from json to xml), it would not be necessary to rewrite all components that depend on the configuration. Clients of ConfigurationProvider interface will continue to work.

Abstraction and encapsulation also play a key role in fighting complexity, providing the ability to design at a higher level, abstracting from implementation details.

## INHERITANCE

Inheritance is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all the properties and behaviors of the parent object.

Here, I hope things should be much easier because you are most familiar with this concept. Inheritance mechanisms also play a key role in the object-oriented approach, in terms of extensibility, reusability of components in the system. This is an "is" or "is a" relationship, a relationship between a base class and descendants. This relationship is the strongest and in statically typed languages it cannot be broken, and this must be considered when assessing the need to use inheritance in this case. If inheritance were applied in a place where one could do without it, as a result, with poor support, this all leads to difficult to understand and difficult to maintain code, because the inheritance hierarchy can be 10 classes or more, and it is rather difficult to understand somewhere in the middle or how the last class will behave, it is hard to understand in what places which methods are being overwritten or overridden, and so on. Therefore, inheritance must be approached wisely.

class Person {

  protected name: string;

  constructor(name: string) {

    this.name = name;

  }

}

class Employee extends Person {

  private department: string;

  constructor(name: string, department: string) {

    super(name);

    this.department = department;

  }

  public getDetails() {

    return `Hello, my name is ${this.name} and I work in ${this.department}.`;

  }

}

const howard = new Employee("Howard", "Sales");

console.log(howard.getDetails()); // ok

console.log(howard.name); // error

console.log(howard.department) // error

Listing 3.6 – Inheritance example.

On the Listing 3.6 you can see an example of inheritance. We have a Person class and an Employee class. Employee inherits some methods and some properties from Person. There are 3 types of access modifiers:

1. Private – not accessible from the outside, only instances of this current class can work with these properties. The Employee class has a department property, and only objects of the Employee class can work with this property.
2. Protected – is a little wider than private, only instances of the current class and classes of descendants can work with them. From Employee, we can refer to name from Person. Moreover, they are also closed to the outside world.
3. Public - public properties and methods are those that are provided to clients in the form of a public interface, on which they will depend, which should be the most stable and the most unchangeable.

## POLYMORPHISM

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

In general, the word polymorphism consists of two parts: poly – many, morphs - forms, that is, many forms. The word polymorphism is used not only in programming, but also in other areas and it describes situations where something can exist in several different forms. In programming, polymorphism describes such a concept when objects of different types are sharing the same interface. Each type can provide its own implementation of this interface.

If we discuss the concept of polymorphism in its classical representation, that is, in statically typed languages, in Java or Typescript, there are two types of polymorphism - static polymorphism and dynamic polymorphism:

* Static polymorphism: allows you to implement multiple methods within the same class that use the same name but a different set of parameters. That is called method overloading. (Listing 3.7)
* Dynamic polymorphism: does not allow the compiler to determine the executed method. Within an inheritance hierarchy, a subclass can override a method of its superclass. That enables the developer of the subclass to customize or completely replace the behavior of that method. (Listing 3.8)

Listing 3.7 is an example of static polymorphism in TypeScript. There is a certain HeroService from the Angular documentation and it has a getHero method that behaves differently depending on the parameters that are passed to it.

Method overloading differs from one language to another because TypeScript makes it impossible to write different implementations for the same method. But it allows to write different signatures so that you can declare a method with different arguments and with different types of values obtained, while still using the same implementation.

interface Hero {

  name: string;

  skill: string;

  weakness: string;

}

class HeroService {

  protected heroes: Array<Hero> = [

    { name: "Superman", skill: "fly", weakness: "cryptonit" },

    { name: "Spiderman", skill: "spider-sense", weakness: "MJ" },

    { name: "Batman", skill: "superhuman power", weakness: "law" },

    { name: "Flash", skill: "run", weakness: "unknown" },

  ];

  public getHero(name: string);

  public getHero(name: string, skill: string);

  public getHero(name: string, skill?: string): Hero {

    if (!skill) {

      return this.heroes.find((hero) => hero.name === name);

    }

    return this.heroes.find(

      (hero) => hero.name === name && hero.skill === skill

    );

  }

}

const heroService = new HeroService();

const hero1 = heroService.getHero("Flash");

const hero2 = heroService.getHero("Superman", "fly");

Listing 3.7 – Static polymorphism example.

Dynamic polymorphism. Everything is simple here; this is a usual method overriding in the inherited classes. Dynamic, because at the compilation stage no binding is carried out and the actual implementation of the method can be defined only in Runtime, we do not know which of the class instances will be called. On the Listing 3.8 you can see that in AntiheroService method getHero is overridden to accept weakness instead of name or skill.

class HeroService {

  // implementation of HeroService

}

class AntiHeroService extends HeroService {

  public getHero(weakness: string): Hero {

    return this.heroes.find((hero) => hero.weakness === weakness);

  }

}

const antiHeroService = new AntiHeroService();

const hero = antiHeroService.getHero("law");

Listing 3.8 – Dynamic polymorphism example.

# Object-oriented design Introduction

Object-oriented design is the process of planning a system of **interacting objects** for the purpose of solving a **software problem**. It is one approach to software design.

Let us look at the words in bold: object interactions and no software problem solving. Imagine a situation, you are writing a project from scratch, you have a complete list of requirements, and all the necessary information. At the planning stage, we decided how everything would look, there were no questions about the design in general, everything was clear, the architecture was written. Also imagine that after the application is written it will never change. For such cases, the design is not needed, but usually this is quite rare and usually changes always happen and something always changes. For example, clients did not know what they exactly wanted, or they told something wrong. Even applications that are perfect at first glance are also unstable and may change. Because after the release, the customer may want to change it, expand it, add new features, captures other markets, and so on. Therefore, even the most ideal application will not save you from the need for design.

I think you get the idea: applications that are easy to change are always fun to write and extend. Such applications will be flexible in development, adaptable to new requirements, and so on. Conversely, applications that lack such qualities as flexibility and adaptability - each change will be more expensive and more difficult.

## Design smells

Let us look at the criteria for the so-called bad design or the so-called bad design smells, as there are code smells, there are also bad design smells:

* Rigidity
* Fragility
* Immobility
* Viscosity
* Needless Complexity

Rigidity indicates that a system is rigid if it is difficult to change it, or even a small change will entail high costs. This suggests that the system is no longer flexible and extensible.

Fragility indicates that a system is fragile if a change in some part breaks something in another part, while the part in which something breaks is not connected in any way and does not explicitly depend on the part in which something has changed.

Immobility means that a system is immobile if certain parts of it cannot be separated into separate components or into separate modules and reused in another part of this system, and in the best cases, in other systems in general. But this, of course, is even harder.

Viscosity. The system is viscous if basic operations are difficult or take too long to complete. So, they are simply ignored. An example of such operations that are difficult to modify and take a long time are slow tests. If they are hard to run, if they are hard to write, if they are slow to execute, then most likely no one will support them, no one will write them, no one will run them.

Needless Complexity or premature optimization - indicates that the system is unnecessarily complicated or prematurely optimized, has too much code that is not currently used, but was written with the intention that it may be needed in the future if the customer wants what -that functionality, it seems as we prepared for this. In fact, this generates dead code, that is, the code that is not used, first for days, then months, years, and then everyone forgets who wrote it, why it was written, and are simply afraid to delete it, because, it may be used somewhere, but nobody knows where.

In general, the main reason for all the above listed bad design smells is the lack of flexibility in the system. The system must be flexible, or, as per saying: software must be soft that is, it must be easy to change. And that is why we need a good design.

## What is design?



Figure 4.1 – Lifecycle of a construction project.

Let us imagine the process of creating something. It will have three phases: the actual design phase; the process of creation or construction and the final product.

Let us take construction industry as an example. What will be a design? - Well, obviously a drawing or blueprint. What will the creation process be? - In our case, this will be the construction process in accordance with the ready-made blueprint. And what the final product will be - a house or a building.

As you can see from the diagram, the creation or building process is much bigger than the design process, it is longer and more expensive. That is why there is a need for the blueprint to be well developed at the planning stage, because during the construction process, each change will be too expensive and complicated, and the cost of an error will be too high.

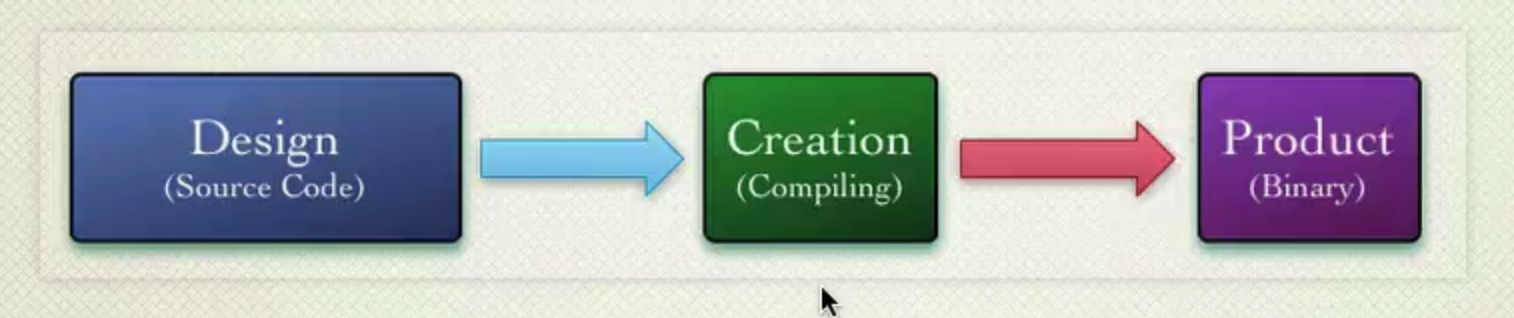


Figure 4.1 – Lifecycle of a software development project.

Now let us apply this constructing analogy to the architecture of a software system, just in reverse order. What will be the final product? – An application that solves some specific problems in its domain area. What will be the creation process? – The process of interpreting or compiling source code. And the design process, oddly enough, is the source code itself.

So, in fact, neither diagrams, nor the relationships between them, nor drawings, and so on - they are not design, they help us structure, organize our thoughts into a specific approach and implement it in the form of source code. In this case, the design phase is much longer and more expensive than the creation phase. That is why the design phase should be iterative, gradual, with constant feedback from both the product and the client. In general, this is the main difference between software and, for example, construction. In software, we have one unbreakable constant that will never change throughout the entire time of our development - these are changes, there have always been changes, they will always occur. And this is the reason why we need design as an ongoing interactive process.

## WHY CHANGE IS HARD AND THE PROBLEM DESIGN SOLVES

* Object-oriented application is made of parts – objects
* Interactions are embodied in the messages that pass between the objects
* Sender object – Target object creates dependencies between the two
* Object-oriented design is about managing dependencies

Requirements, product, ecosystem, environment, customer ¬ all this will change. The system needs a design that is ready for such changes. Such design would consist of certain parts that would interact with each other to create the behavior of something whole: a class, a component, an architectural layer, or an application. As you understand parts are objects, interactions between them are implemented using messages that are sent between these objects. At the same time, sending the correct message to the correct recipient-object requires knowledge of where this object is and how to interact with it. This knowledge creates a relationship between two or three objects, or generally a huge number of objects in the system. Cross dependencies, cycle-dependency and so on arise. Accordingly, all these dependencies complicate system change. Object oriented design is essentially dependency management. In the absence of design, unmanaged dependencies lead to chaos, because objects begin to know too much about each other, and at some point, it is easier to throw everything out and rewrite entire application than to add some next changes. Since these dependencies just become unmanageable. By changing something in one place, even if there are some tests, we do not exclude the possibility that something will not break in another place.

## THE PURPOSE OF OOD

Software must:

* Satisfy customer’s needs
* Be flexible for change and enhancement

Thus, the goal of object-oriented design is just two things. The first is, of course, to satisfy the needs of the customer, and the second is to be easy to change and be adaptable and ready for such changes and extensions.

## THE TOOLS OF DESIGN

* Design Principles: SOLID, DRY, KISS
* Design Patterns: Creational, Behavioral, Structural
* OOP Principles: Abstraction, Encapsulation, Polymorphism, Inheritance

In fact, there are a lot of approaches and tools, for example, these are the very principles of object-oriented programming: abstraction, encapsulation, etc., which were mentioned at the very beginning of the manual, they are basic, theoretical, but they fit perfectly into the practice of writing code, in the system design and are an essential tool in the fighting complexity, that is, a tool in the creation of object-oriented design. Also, such tools as design patterns: Solid, Dry, Kiss.

We will consider all these principles and tools in subsequent lectures.