Contents

Introduction to Object-Oriented Concepts

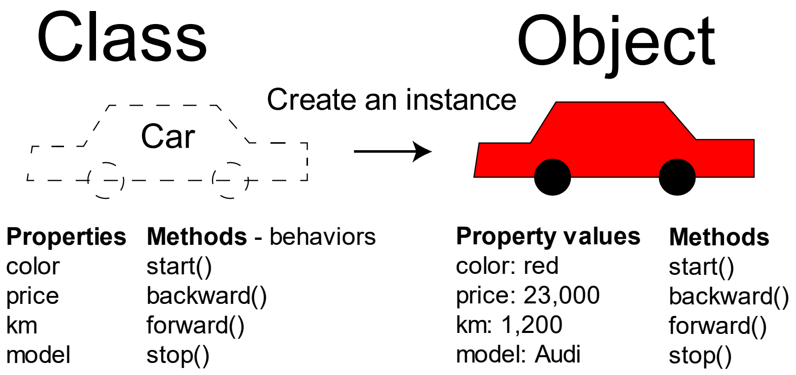
Object-Oriented Programming (OOP) is a programming paradigm that organizes and structures software design around **objects** rather than functions or logic.

Object-oriented concepts give you a solid foundation for making critical design decisions. I will explain the main concepts and how they are used in designing object-oriented systems and guide you through examples.

Class and Objects

A **class** is a blueprint, template or prototype that describes what an object will be. It defines the structure (attributes or properties) and behavior (methods) of an object. We must design a class before creating an object.

An **object** is an instance of a class with specific data and functionality. When we create an object, we create real-world entities such as cars, bicycles, or dogs with their own attributes and own behaviors.



Class vs Object

The following code shows how this Car template is translated into java.

```java
public class Car {

  private String color;
  private double price;
  private double km;
```

Polymorphism

- Ability to present the same interface for different underlying data types or classes.
- Achieved through method overriding and interfaces.

Example:

```
// Base class
class Shape {
  void draw() {
    System.out.println("Drawing a Shape");
  }
}

// Derived classes
class Circle extends Shape {
  @Override
  void draw() {
    System.out.println("Drawing a Circle");
  }
}

class Rectangle extends Shape {
  @Override
  void draw() {
    System.out.println("Drawing a Rectangle");
  }
}

public class Main {
  public static void main(String[] args) {
    Shape[] shapes = { new Circle(), new Rectangle() };

    for (Shape shape : shapes) {
      shape.draw();
    }
    // Output:
    // Drawing a Circle
    // Drawing a Rectangle
  }
}
```

Polymorphism means many shapes and is coupled to inheritance.

Abstraction

Abstraction is a process that focuses on the relevant characteristics of a situation, problem or object and ignores all of the non-essential details.

How is this applied in object-oriented software?:

- Hiding complex implementation details and showing only the necessary features of an object.
- Achieved using abstract classes and interfaces.

Example:

```
// Abstract base class
abstract class Vehicle {
  abstract void startEngine();  // Abstract method
}

// Concrete subclass
class Car extends Vehicle {
  @Override
  void startEngine() {
    System.out.println("Car engine started");
  }
}

public class Main {
  public static void main(String[] args) {
    Vehicle myCar = new Car();
    myCar.startEngine();  // Output: Car engine started
  }
}
```

Abstraction is considered *innate to human beings* because it is a cognitive skill that humans - *programmers* - naturally develop as part of their mental processes.

Here are some examples of abstraction in everyday life:

- When you drive a car, you don't need to know how the engine works in order to operate it. You just need to know how to turn the steering wheel, accelerate, and brake. Do you think, that knowing these minimal functionalities allows you to drive a car? In other words, does it solve your problem?

- When you use a computer, you don't need to know how the operating system works in order to use your applications. You just need to know how to open and close files, and how to use the various features of the applications.

- You can see the essential buttons on your TV remote, and when you press

Abstraction in Object-Oriented Software Development

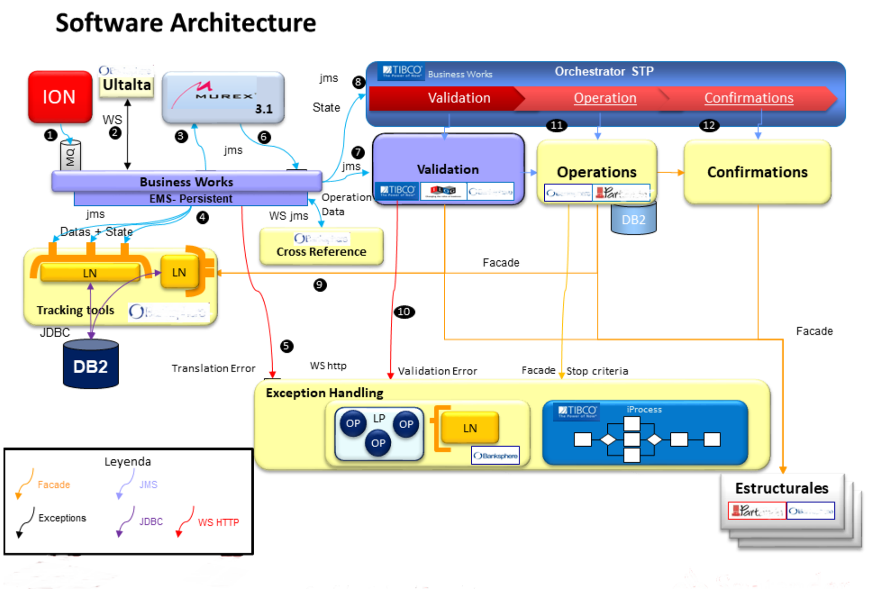
Abstraction allows an *object* telling to its users what an application does instead of how it does it.

Before becoming developers, we spend a relatively long time learning and developing our cognitive abilities, including abstraction.

We learn to recognize books, cars, streets, and clothes, and they are part of our daily life. And they usually **don't have drastic changes**.

While abstraction is a natural cognitive ability for software developers, abstracting concepts in a business domain can be challenging for several reasons:

- Domain Complexity: Business domains can be quite complex, involving intricate processes, workflows, and interactions between various stakeholders. This complexity can make it difficult for developers to fully grasp the essence of the domain and identify the core concepts that need to be represented in the software.



We use abstractions to model complex systems

- Lack of Domain Knowledge: Developers often specialize in software

Method overriding happens when a subclass has the same method as the parent class and provides a particular implementation when the method is called.

One way to implement polymorphism is via method overriding, which happens at runtime.

Relationships among Classes

Objects contribute to the behavior of a system by collaborating with one another. An object communicates with another object to use the results of operations provided by that object.

Association

Association is a relationship between two classes that are independent of one another.

Association can be of four types: one-to-one, one-to-many, many-to-one, and many-to-many.

For example, we have the Customer and Address classes.

```java
public class Customer {

  private String name;

  public Customer(String name) {
    this.name = name;
  }

  // getters and setters omitted for brevity
}
```

```java
public class Address {

  private String street;

  public Address(String street) {
    this.street = street;
  }

  // getters and setters omitted for brevity
}
```

We associate these two separate classes through their objects in the *main* method.

```
public class TestAssociation {
  public static void main(String[] args) {
    Customer customer = new Customer("Adam Fox");
    Address address = new Address("BornStrasse 12");
    System.out.println(customer.getName + " lives in " +
                       address.getStreet() + " street");
  }
}
```

Output

```
Adam Fox lives in BornStrasse 12 street
```

Aggregation

Aggregation is a particular case of association and is unidirectional.

The aggregated objects have their life cycle, but one of the objects is the owner of the HAS-A relationship.

For example, a *supplier* can provide products in a B2B food business.

```
public class Supplier {
  private String name;

  public Supplier(String name) {
    this.name = name;
  }

  // getters and setters omitted for brevity
}
```

And a *buyer* can order products from one specific *supplier*.

```
public class Buyer {
  private String name;
  private Supplier supplier;
  public Buyer(String name, Supplier supplier) {
    this.name = name;
    this.supplier = supplier;
  }
```

Create Classes by reusing Objects from other Classes.

The aggregation and composition provide a mechanism for building classes from other classes. In Java, we usually create a class with instance variables that references one or more objects of other classes.

The benefit of separating one class from another one is the Reuse.

For example, in a shopping context, we need a list of requested items and an *Address* class where to deliver an order.

```
public class Order {

  private int clientId;
  private List<Item> orderItems;
  private Address shippingAddress;

  //code omitted for brevity
}
```

We build an *Item* class including an *Article* class.

```
public class Item {

  private Article article;
  private double quantity;

  //code omitted for brevity
}
```

And the *Article* class includes enough attributes to support the shopping business.

```
public class Article {
  private int id;
  private String name;
  private double deliveryPrice;
  //code omitted for brevity
}
```

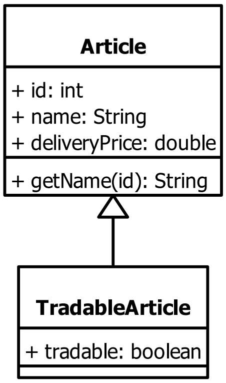
Even we can reuse the *Article* class to support a search request.

```
public class SearchResponse {
  private List<Article> articles;
  //code omitted for brevity
}
```

What happens if the business wants to introduce articles in a country where some articles are forbidden to trade?

We cannot add a new attribute called *tradable* to the Article class because we will never use it in normal countries.

Here, we can use the other technique to build new classes: inheritance.



Now, we can support the new requirement for the new country.

```
public class SearchResponse {

  private List<TradableArticle> articles;

  //code omitted for brevity
}
```

We can reuse our classes even out of context. For example, in a Bank context, we need an *Address* class to contact a customer.

```
public class Customer {

  private int customerId;
  private String lastName;
  private Address address;

  //code omitted for brevity
}
```

UML Diagrams for developers

The Unified Modeling Language is a graphical notation for modeling systems and conveying user software requirements. All developers must understand this notation before starting programming.

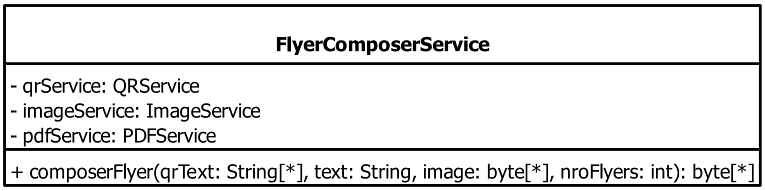
UML is not only pretty pictures. Instead, they communicate the software design decisions to programmers.

Why do we model?

- We build models to understand better the system we are developing.
- Models document the design decisions we have made.
- Models allow an open discussion in the development team before starting programming.
- It speeds up the implementation stage because potential technical issues are discussed during the design stage.
- Explain our software design proposal to external partners.

UML Class Notation

A class is a template for creating objects providing initial values for state (attributes) and behavior (operations). Each attribute has a type. Each operation has a signature.


FlyerComposerService
- qrService: QRService
- imageService: ImageService
- pdfService: PDFService
+ composerFlyer(qrText: String[*], text: String, image: byte[*], nroFlyers: int): byte[*]

From the figure above:

- The first compartment describes the class name.
- The second compartment describes the attributes with its visibility, private(-) or public(+), and their data types.
- The third compartment describes the operations and their return types.

The following section shows how these compartments are translated into code.

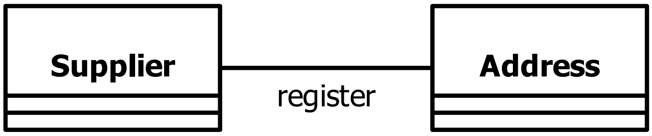
```
public class FlyerComposerService {

  private QRService qrService;
  private ImageService imageService;
  private PDFService pdfService;

  public byte[] composeFlyer(String[] qrText,
    String text,
    byte[] image,
    int nroFlyers) {
    //code omitted for brevity
  }
}
```

Relationships among classes

UML conveys how a class is related to other classes. Let's see the kind of relationships that matter to software design.

Association

An association draws a solid line connecting two classes. It could be named by a verb (using role names) that reflects the business problem domain we are modeling. The following diagram shows two classes that need to communicate with each other.



Aggregation

Aggregation is a particular association type representing a *has-a* relationship and is displayed as a solid line with an unfilled diamond at the association end.

A child class object can exist without the parent class object. In the following diagram, if you delete the Buyer class (parent), the Supplier class (child) still exists.

Coding Best Practices

Clean code can be read and enhanced by a developer other than its original author.

This kind of practice Robert C Martin introduced it.

If you want to be a better programmer, you must follow these recommendations.

Clean code has Intention-Revealing names

Names reveal intent. Someone who reads your code must understand the purpose of your variable, function, or class.

Real situation:

```
int sId; //supplier Id
int artDelPrice;
```

It must be refactored to this:

```
int supplierId;
int articleDeliveredPrice;
```

Even with external dependencies:

```
private Z_E2F_RS_Result e2fResult; //ingredients recordset
```

It must be refactored to this:

```
private Z_E2F_RS_Result ingredients;
```

Imagine that we don't have the *//ingredients* comment in the *e2fResult* variable. Then, further in any part of our code, when we try to process this variable, we have the following sentence:

```
e2f = e2fResult[i];
```

And we don't know what e2f means! Well, someone suggests asking the person responsible for this code. But that guy is not at the office. Well, send it an email, and he is on holiday!

But if we adopt names that reveal intent from the beginning, we could avoid these catastrophic scenarios.

```
ingredient = ingredients[i];
```

Clean code tells a story

When we try to fix bugs, when analyzing the sequence of actions (functions, methods), we realize the code does not communicate well the logical flow of these actions. It's a nightmare to decode the meaning of these actions.

This will always happen because our initial design based on the initial requirements changes over time. As developers, we are responsible for refactoring our code to make it a simple story that everybody can understand. For example, look at the following code.

```
public void performTask(String process) {

  ACMEWebServiceClient.login();
  if (process.equals("core") {
    ACMEWebServiceClient.transfer_buyersCoreData_to_ACME();
  }
  if (process.equals("status")) {
    ACMEWebServiceClient.transfer_buyersStatusChanges_to_ACME();
  }
  if (process.equals("events")) {
    ACMEWebServiceClient.transfer_events_to_ACME();
  }
  ACMEServiceClient.logout();

}
```

Functions should do one thing

Imagine we want to retrieve image objects from an external web service.

We receive image metadata in an array of Strings that informs different values to decide whether an image is valid. One of these values is an image identifier to retrieve the final image object.

```
private String retrieveImageId(String[] values) {

  if (!values[2].equals("Y") || !values[3].equals("Y"))
    return null;
  String imageId = null;
```

Avoid hard coding

Hard coding is embedding data directly into the source code instead of obtaining the data from external sources.

Sometimes we can’t avoid including conditional statements using hardcoded values because we need to implement them in a production environment immediately. There are hundreds of reasons why this happens because every company is different.

A company wants to implement in its code validation of customers who have the right to view images from certain providers.

The standard procedure in this company starts with a requirement to the DBA to implement a database function to retrieve a list of providers with this kind of restriction, create param classes for the developers, a period of implementation in a development environment, and its tests in a test environment, and deliver to the production environment.

But the company is facing problems with image author property rights and does not have the resources to implement the requirement, then decides to introduce hard-coded values.

```
boolean picIsRestricted =
 result.getProviderId() == "530636" || result.getProviderId() == "36507";
```

We usually forget the standard procedure to implement the solution because our code is already working. But these hard code values are required in other modules, packages, and classes and may need to validate more providers, etc., and the effort to maintain the code increase exponentially. And I think you know the rest of the history.

You can implement a little function to retrieve external data from a text file.

```
public interface DataService {

  public List<String> getRestrictedProviders() throws Exception;

}
```

Software Design Principles

Software design principles are guidelines and best practices that help software developers create high-quality, maintainable, and efficient software.

Let's see some commonly recognized software design principles.

In the realm of software engineering, few concepts have had as profound an impact on code quality, maintainability, and scalability as the SOLID principles. These five guiding tenets—**S**ingle Responsibility, **O**pen/Closed, **L**iskov Substitution, **I**nterface Segregation, and **D**ependency Inversion—represent a foundational framework upon which modern software design is built. They are the cornerstone of clean, robust, and adaptable code.

SOLID principles tell you how to arrange your functions into classes and how those classes should be interrelated. Robert C. Martin[1] introduced it.

Whether you are a seasoned programmer seeking to reinforce your expertise or a novice developer eager to grasp the essentials of SOLID design, this section provides a comprehensive roadmap to understanding and mastering these fundamental principles.

Let's look at each SOLID principle in detail and then move on to the other principles.

Single Responsibility Principle (SRP)

"A class should have only one reason to change."

This principle states that **a class should only have one responsibility**.

Following SRP ensures that a class is easier to understand, test, and maintain. When a class has multiple responsibilities, changes in one responsibility can affect the other, leading to bugs and code that is harder to maintain.

[1] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Example 1: File Handling (Violation of SRP)

In this example, a single class handles multiple responsibilities: reading a file, parsing the data, and displaying it.

```
// Violates SRP
public class FileManager {
  public void readFile(String filePath) {
    System.out.println("Reading file: " + filePath);
    // Code to read the file
  }

  public void parseData() {
    System.out.println("Parsing file data...");
    // Code to parse the file data
  }

  public void displayData() {
    System.out.println("Displaying data...");
    // Code to display the parsed data
  }
}
```

If the logic for reading a file, parsing, or displaying data changes, this single class will need to be modified multiple times, violating SRP.

Refactoring to Follow SRP

Let's split the responsibilities into separate classes.

```
// Class responsible for reading a file
public class FileReader {
  public String readFile(String filePath) {
    System.out.println("Reading file: " + filePath);
    // Simulated file content
    return "file content";
  }
}

// Class responsible for parsing data
public class DataParser {
  public String parseData(String fileContent) {
    System.out.println("Parsing file content...");
    // Simulated parsed data
    return "parsed data";
  }
}

// Class responsible for displaying data
public class DataDisplayer {
  public void displayData(String data) {
    System.out.println("Displaying data: " + data);
  }
}
```

Now, these classes can work together:

```
public class Main {
  public static void main(String[] args) {
    FileReader fileReader = new FileReader();
    String content = fileReader.readFile("data.txt");

    DataParser dataParser = new DataParser();
    String parsedData = dataParser.parseData(content);

    DataDisplayer dataDisplayer = new DataDisplayer();
    dataDisplayer.displayData(parsedData);
  }
}
```

Here:

- FileReader is responsible only for reading the file.
- DataParser is responsible only for parsing the content.
- DataDisplayer is responsible only for displaying the data.

Example 3: Payment and Card Teams (Violation of SRP)

For instance, imagine an online store that issues its cards for its customers, and from the beginning, the Payment and Card teams are in mutual agreement to apply for interest and to lock cards from customers who are in late payments for 14 days or more.

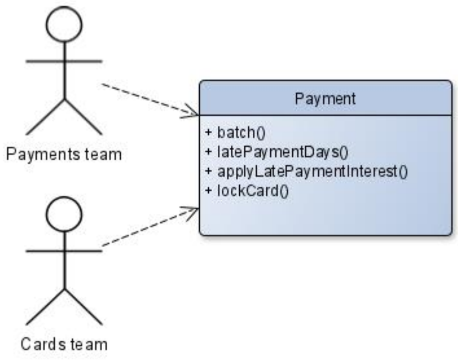
In the following Code, we have the first implementation of the Payment Class, which supports both requirements.

```
public class Payment {
  public static final int MAX_DAYS = 14;

  public void batch(List<Customer> customers) {
    for (Customer customer : customers) {
      int nDays = latePaymentDays(customer);
      if (nDays >= MAX_DAYS) {
        applyLatePaymentInterest(customer);
        lockCard(customer);
      }
    }
  }
}
```

The Problem: A Class has more than one responsibility

But suddenly, the Cards team wants to change the validation to 10 days. However, the Payments team manages other policies related to when interests by late payment are applied. As a result, the Payments team disagrees with the Cards team. Moreover, both teams are stuck on how to proceed.

This scenario is a clear example of how this Class design violates the Single Responsibility Principle. The Payment class has more than one reason to change and breaks the Payments team's business logic if they accept the Cards team's requirement.



Payment class

The previous figure shows the Payment Class with different responsibilities

The solution: Create Classes with only one responsibility

What do we need to do? In this scenario, we can apply the Single Responsibility Principle.

Refactored Version to Follow SRP

We move the *lockCard()* responsibility to a new Card Class. This technique is most known as refactoring.

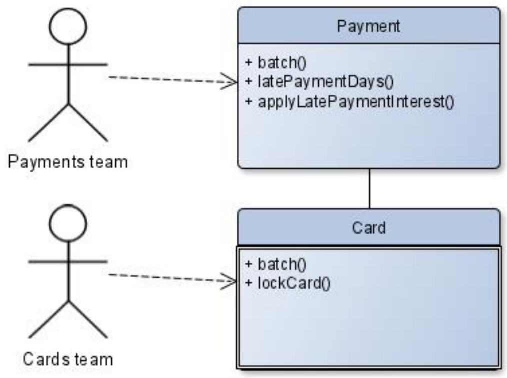
```
public class Card {
  public static final int MAX_DAYS = 10;
  public void batch(List customers) {
    for (Customer customer : customers) {
      int nDays = Payment.latePaymentDays(customer);
      if (nDays >= MAX_DAYS) {
        lockCard(customer);
      }
    }
  }
}
```

After that change and following clean code[2] principles, we can see how it looks the new Payment Class (refactored as well).

```
public class Payment {
  public static final int MAX_DAYS = 14;
  public void batch(List<Customer> customers) {
    for (Customer customer : customers) {
      int nDays = latePaymentDays(customer);
      if (nDays >= MAX_DAYS) {
        applyLatePaymentInterest(customer);
      }
    }
  }
}
```

Now, new changes to the MAX_DAYS variable will only depend on the requirements of every team separately. The following figure shows the Classes for different actors, without conflicts.

Therefore, the Payment Class is only responsible for supporting to the Payments team, and the Card Class is solely responsible for supporting the Cards team.



Classes for different actors

[2] https://codersite.dev/clean-code/

Also, when new features arrive, then we need to distinct in which class to include it. Moreover, this is related to the High Cohesion concept, which help us to group similar functions inside a Class, and that have the same purpose served by that class.

Use this principle as a tool when translating business software requirements into technical specifications.

Key Takeaways for SRP

- When you find that a class has many responsibilities, delegate unrelated responsibilities to new classes; in this way, each class will have only one responsibility and one reason to change.

- This principle is intended to separate behaviors into different classes so that if bugs arise due to your change, they don't affect other classes with unrelated behaviors.

- When you implement the single responsibility principle by abstracting your classes via interfaces, your code will be more adaptable to changing requirements in the way Agile frameworks demand.

- The advantages of having a single responsibility are that classes are easier to test, easier to understand their purpose, and easier to maintain.

- Enhanced Collaboration: Developers can work on different responsibilities simultaneously.

In conclusion, once you identify classes that have too many responsibilities, use this refactoring technique to create smaller classes with single responsibilities and focused only on one business actor.

Open-Closed Principle (OCP)

A class, module, or function should be open for extension but closed for modification.

This means you should design your classes and modules so that you can add

Don't Repeat Yourself (DRY)

The DRY principle states that code should not be repeated unnecessarily. Instead, developers should use abstractions, modularization, and other techniques to reduce repetition and make code more maintainable.

The DRY principle also state that every piece of knowledge must have a single, unambiguous, authoritative representation within a system. The principle was formulated in the book "The Pragmatic Programmer[3]".

The principle aims to avoid the creation of duplicate representations of knowledge, which means avoiding duplicated code in our software.

For example, if you have routines where you need to format specific data, move them to a Utility interface. Create a reference to the abstract interface from different parts of your application and use its standard methods.

Consider using frameworks and libraries that encapsulate common logic.

Example 1: Refactoring Duplicate Code into a Method

Suppose you have the following code that calculates the area of two different rectangles in different parts of the application:

```
public class AreaCalculator {
  public static void main(String[] args) {
    int width1 = 5;
    int height1 = 10;
    int area1 = width1 * height1;
    System.out.println("Area of rectangle 1: " + area1);
    int width2 = 7;
    int height2 = 3;
    int area2 = width2 * height2;
    System.out.println("Area of rectangle 2: " + area2);
  }
}
```

In this example, the formula for calculating the area (**width * height**) is duplicated. If this formula ever changes (e.g., you start considering a border area), you'd have to update it in multiple places, which could lead to errors.

Refactored Code:

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

```
public class AreaCalculator {
  public static void main(String[] args) {
    System.out.println("Area of rectangle 1: " + calculateArea(5, 10));
    System.out.println("Area of rectangle 2: " + calculateArea(7, 3));
  }
  private static int calculateArea(int width, int height) {
    return width * height;
  }
}
```

By moving the area calculation to a calculateArea method, we remove redundancy. Now if we need to change the calculation, we only do it in one place.

Example 2: Using Constants to Avoid Magic Numbers

Here's an example with "magic numbers," which are hardcoded values that might be used multiple times in a program.

```
public class CircleCalculator {
  public static void main(String[] args) {
    double radius1 = 5;
    double area1 = 3.14159 * radius1 * radius1;
    System.out.println("Area of circle 1: " + area1);
    double radius2 = 7;
    double area2 = 3.14159 * radius2 * radius2;
    System.out.println("Area of circle 2: " + area2);
  }
}
```

Here, `3.14159` is repeated. This not only violates DRY but also reduces code readability. Refactored Code with Constants:

```
public class CircleCalculator {
  private static final double PI = 3.14159;
  public static void main(String[] args) {
    System.out.println("Area of circle 1: " + calculateCircleArea(5));
    System.out.println("Area of circle 2: " + calculateCircleArea(7));
  }
  private static double calculateCircleArea(double radius) {
    return PI * radius * radius;
  }
}
```

By defining `PI` as a constant, the code is cleaner, and if you ever need to change the value of `PI` (e.g., to use `Math.PI`), you can do it in one place.

Soft Skills

Soft skills are the personal attributes that enable individuals to interact effectively and harmoniously with others. In software development, having strong soft skills is just as important as technical skills. Here are some examples of soft skills that are valuable in software development:

1. Communication: Good communication skills are essential for effective collaboration between developers, project managers, and other stakeholders. This includes the ability to communicate technical information clearly and concisely, as well as the ability to listen actively and ask questions.
2. Problem-solving: In software development, problems are inevitable. A developer with strong problem-solving skills can quickly identify issues and develop effective solutions.

3. Teamwork: Successful software development requires a collaborative effort. Team players can work effectively with others, understand their role within a team, and are willing to lend a hand when needed.

4. Adaptability: The ability to adapt to new technologies, processes, and challenges is crucial in software development, where change is a constant.

5. Time management: Meeting deadlines is crucial in software development. Strong time management skills enable developers to prioritize tasks effectively and deliver projects on time.

6. Attention to detail: Paying attention to details can help developers catch errors, bugs, and other issues before they become bigger problems.

7. Creativity: Creativity can help developers find innovative solutions to complex problems, and come up with new and better ways to approach development challenges.

These are just a few examples of soft skills that can be valuable in software development. Having a strong combination of technical and soft skills can help developers excel in their careers and contribute to successful software projects.

Case Studies

How to interpret a functional description

Imagine that you must design and implement an Order API from its documentation. For example:

Resources and their relationships

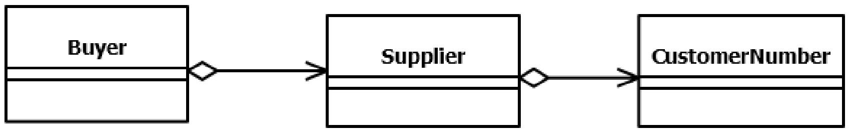
*A **buyer** represents an organization aiming to place orders at suppliers to get food and ingredients delivered. This organization can be a single kitchen, a central buying department or any other setup depending on your use-case.*

*A **supplier** represents an organization selling and delivering food and ingredients. These organizations range from local, small agriculture family businesses to large international corporations.*

*A **customer number** is used to identify a buyer towards a supplier. A buyer receives a customer number from a supplier after registration. A buyer will have different customer numbers for different suppliers and might have multiple customers numbers for the same supplier, for example to differentiate departments within the buyer's organization. While technically a customer number is optionally it is very common, and most suppliers work with customer numbers.*

Identify the entities and their relationships

The following figure shows the relationships between these entities.



We design different functionalities (in services classes) for every entity that fit the global API requirements, such as creating, updating, and deleting a buyer, and a list of buyers.

We create abstractions about these concepts by creating interfaces.

To retrieve a list of suppliers we already know that we need a buyerId and additional parameters.

Remember to start with classes and methods that are simple and manageable.

```
public interface SuppliersService {
  public List<Supplier> listSuppliers (
    Integer buyerId,
    String supplierNumber,
    String sortBy,
    String sortOrder,
    Integer offset,
    Integer limit) throws Exception;
}
```

To retrieve a list of customers numbers we already know that we need a buyerId, a supplierId and additional parameters.

```
public interface CustomerNumbersService {
  public List<CustomerNumber> listCustomerNumbers(
    Integer buyerId,
    Integer supplierId,
    String customerNumber,
    String sortBy,
    String sortOrder,
    Integer offset,
    Integer limit) throws Exception;
}
```

By *separating concerns* into different specialized classes, reduces the effort of maintenance in future changes because we can localize faster the code where we need to implement, modify, or fix a bug. You will feel confident about changing the code.

Designing a RESTFul API

RESTful refers to implementing REST[4], an architectural style defining guidelines for creating web services.

Business requirement

A company wants to implement an API to allow Restaurants (represented as a buyer) to place orders at suppliers to get food articles delivered.

After thoroughly analyzing the requirements, the developer team identified the following *class diagram* showing the main entities and their relationships.

[4] https://codersite.dev/rest-api-overview/

Design a B2B Integration Project

Business-to-business (B2B) is a business conducted between one company and another.

One leading company implements an E-commerce portal to sell t-shirt products from several retail businesses and manufacturers. Uploading of products is done manually by retailers through a user interface.

Identification of a problem

All retailers want to sell their products on the portal because the leading company has a lot of traffic on the internet and is well recognized. But most of the retailers have their websites, so they ***duplicate the effort*** of the product uploading process.

The business requirement

A retailer sends its product data via a CSV file to the lead company, which uploads it into its database. The lead company then displays this data on its e-commerce portal for selling.

The retailer already has a public RESTful API to retrieve product data from a local database and display it on its own website. So the retailer asks the lead company to use the public API to retrieve thousands of products instead of retrieving this data from a CSV file. And eventually, the lead company can update the product images weekly by calling the API.

Your creative process in action

The company's IT department assigns you the task of **designing** the new software system to achieve the business requirement before sending it to a programming factory.

Well, you already know that software development is a creative process, similar to what an artist does. That means there are different solutions to achieve this automation process.

As a general rule, this is what we usually do when we develop software:

- Analyse carefully the current problem and the business requirements. Ask everything, even what looks obvious.

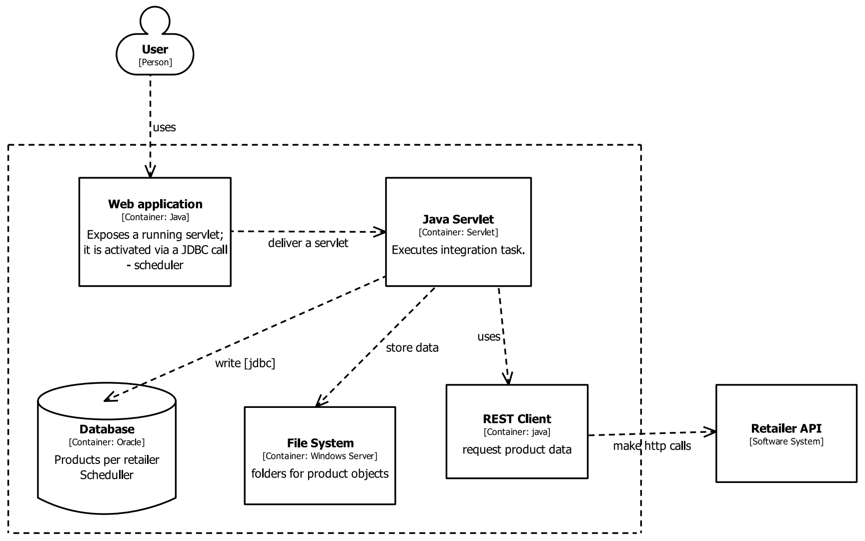
A software system comprises applications and data stores deployed in containers.

Container diagram

Container. A container is something that must be running for an application or data store to work.

Each container is a runtime environment that (not always) runs in its own process space. Inter-container communication takes the form of inter-process communication.

The container diagram shows how the responsibilities of a software system are distributed in different execution units - containers.

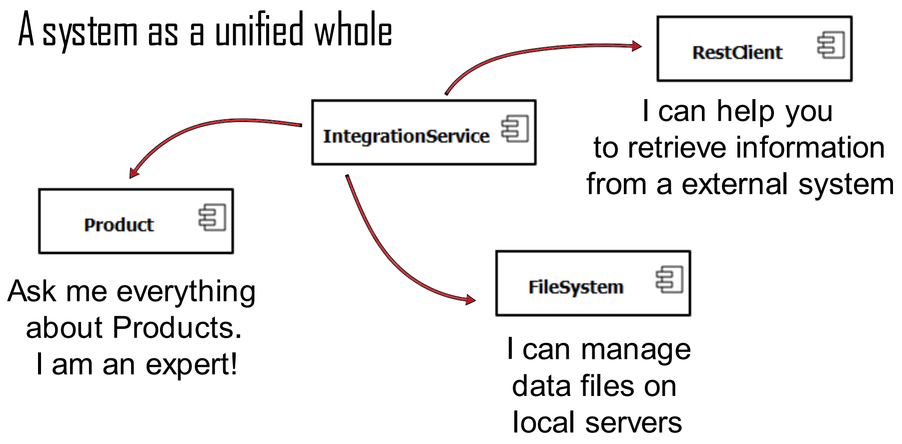


When the web application is deployed to an application server, a servlet waits for a request parameter to start the process, then delegates the task of retrieving data from an external API to a REST client.

The integration process must communicate with a database container such as Oracle to read/write data products and a file system container such as Windows Server to store image objects.

Component diagram

Component. A component is a group of related functionalities encapsulated


A system as a unified whole
IntegrationService
RestClient
I can help you to retrieve information from a external system
Product
Ask me everything about Products. I am an expert!
FileSystem
I can manage data files on local servers

Every module has one responsibility

When you are in charge of software design, remember to think about modularization.

How the new design handle complexity better

Change requirement is under control thanks to abstractions because we know where we need to touch the code in the software.

The cognitive load is minimal because developers do not need to know too much detail about how a method works described in an abstract interface. A developer can understand how the code works and what is needed to make a change thanks to design principles.

There is no duplicated business knowledge in the code because we have delegated specific responsibilities to specific reusable modules.

Dependencies between components are minimal and easy to recognize by using abstractions.

Conclusions

- Software or program design is a complex process limited only by our imagination, reasoning, opinions, and experience. To create software for commercial purposes, we must consider the abstractions and standard notations to understand and support the design process.