**Nursultan Serikov**

**Zhalgas Srymbetov**

**Anuar Duisenov**

**Daulet Tulegenov**

**Instructor: Almas Baisultan**

**Project report**

**SE-2224**

**Gadget Store Marketplace**

# **Outline**

1. **[Introduction.](#Introduction)**
2. Team composition…………………………………………………….3
3. Responsibilities of participants……………………………………….3
4. [**OOP.**](#OOP)
5. The meaning of OOP…………………………………………………3
6. General concepts of OOP.…………………………………………….4
7. OOP features..…………………………………………...……………5
8. [**Team project**](#Team_project)
9. Basics of project..…………………………………………………….5
10. Short plan of development…………………………………………...5
11. [**Documentation.**](#Documentation)
12. Hierarchy……………………………………………………………..6
13. S.O.L.I.D of classes…………………………………………………..7
14. SQL connections……………………………………………………..8
15. Design patterns………………………………………………………10
16. Console……………………………………………………………...12
17. Final Output…………………………………………………………15
18. [**Conclusion.**](#Conclusion)

# **Introduction**

## **1. Team composition.**

Our team consists of four people:

* Nursultan
* Zhalgas
* Anuar
* Daulet

## **2. Responsibilities of participants.**

Sharing work is an important part of any teamwork. The main coding part was entrusted to **Nursultan**. Production, the hierarchy was created by **Zhalgas**. **Daulet and Anuar** oversaw finding references, through which the backbone, meaning, ideas and thoughts for the project were built.

In this way, work on the project was balanced, with each member doing their part. However, during the development process, we consulted each other and pointed out our mistakes.

# **OOP**

## **The meaning of OOP.**

Object-oriented programming generates an enormous amount of controversy around itself. Many people see it as an asset in the programming world, while others perceive OOP as something fatal.

Object-oriented programming generates an enormous amount of controversy around itself. Many people see it as an asset in the programming world, while others perceive OOP as something fatal.

We believe that OOP is a theory. It is a style somewhat similar to the world around us. “**Object-oriented programming** has been preferred for modeling and to provide solutions for complex tasks with use of objects interactions” (Singh et. al., 2021, p. 2).

Everything in our world is an object. That's what the OOP is all about.

## **General concepts of OOP.**

According to **Wegner (1990)**, general concepts of OOP is: Encapsulation, Inheritance, Polymorphism, Abstraction.

**Encapsulation:**

Encapsulation is the process of exposing only a public interface for interacting with an object and keeping the implementation details hidden from the outside world. This protects against inadvertent or unauthorized modifications by guaranteeing that the internal state of an object can only be changed in a regulated manner.

**Inheritance:**

A class can take methods and attributes from another class through inheritance. The class being inherited from is referred to as the base or parent class, while the class being inherited from is referred to as the derived or child class. Code maintenance and code duplication can both be improved by inheritance.

**Polymorphism:**

The term "polymorphism" describes an object's capacity to assume various forms. Method overriding, which enables a derived class to offer its own implementation of a method that is already defined in the base class, is a common OOP technique for achieving polymorphism. Coding that is adaptable to various use cases and versatile is made simpler by polymorphism.

**Abstraction:**

The representation of complicated systems using basic, high-level ideas is known as abstraction. In object-oriented programming (OOP), abstraction is frequently achieved through the use of abstract classes and interfaces, which define a set of characteristics and methods necessary for a particular type of object while omitting implementation details. By separating implementation details from the rest of the code, abstraction can help make code more modular and simpler to maintain.

## **OOP features.**

As Brookes et. al. (1994) noted, the main feature of OOP is the programming style itself.

The main feature of OOP is the programming **style itself**. OOP allows development in such a way that the code is essentially divided into modules, which makes solving complex problems much easier than in procedural programming. The code can be constantly evolved and added to, depending on the development needs. The division into classes, methods makes the code easier to understand.

In general, OOP can open up new programming niches, completely new ways to solve problems.

# **Team project**

## **Basics of project.**

The basis for our project will be the object-oriented language **Java**, according to the course. Various internet marketplaces will be used as references for development, based on which our own site will be created. **PostgreSQL** database management system will be used for data storage and processing.

According to the terms of reference, front-end is not needed for the project, so the basis for management will be a console interface.

## **Short plan of development.**

According to the short plan, the development should take place in three phases:

1. Drawing up the foundations for the project.

2. Connecting to the database.

3. Development of the console, final stages.

At the beginning, we will create a class hierarchy, which will be the basis of our project. It will consist of the following classes: market - device - device varieties - gadget varieties.

The most important prerequisite for the initial phase is to lay down the S.O.L.I.D principle, which is the basis of the terms of reference.

The second stage will be the most complex, because of the variety of methods and classes to be created.

A PostgreSQL driver will be introduced into the project, and then the database itself will be created in which we will store the data.

After that, we will create a class called market, which will inherit a series of CRUD methods, with the help of which we will operate with objects. The complexity of this stage lies in the work that needs to be done. We need to create quite a lot of different methods, classes, interfaces. It is necessary to think through the logic of SQL queries, in order to optimize them.

In the final stage, there will be a console through which we will operate the marketplace. The core of the console will be methods; we will call them by selecting a certain service or function. Also, during this period, we will be fine-tuning what has already been created, literally bringing the project "up to date".

# **Documentation**

## **Hierarchy.**

The hierarchy is structured as follows. There is a market class, below which is the device class. From a device, classes are inherited, subtypes of devices are inherited, and from subtypes, devices are already varieties of these very devices.

public abstract class Device implements DeviceSpecification {  
 private String brand;  
 private String model;  
 private double price;

public class Tablet extends Device {  
 private boolean stylusSup;  
 private boolean keyboardSup;  
 private String screenRatio;

public class Phone extends Device {  
 private int ram;  
 private int memory;  
 private String screenSize;  
 private String camera;

public class Gadget extends Device {  
 private String type;

These are followed by varieties like smartphones, mobile phones, gaming tablets, headphones, etc.

public class Smartphone extends Phone {  
 private String OS;  
 private boolean faceID;  
 private boolean fingerprintSensor;

}

public class GamingTablet extends Tablet {  
 private String gpu;  
 private boolean vrSupport;  
 private int memory;  
 private int ram;

}

public class Headphones extends Gadget {  
 private String audioQuality;  
 private boolean noiseCancel;

}

## **S.O.L.I.D of classes.**

Building the S.O.L.I.D. principle is the main objective of this project. To comply with the S.O.L.I.D. principle, the DeviceSpecification interface was created so that the superclass can always be supplemented with something new.

public interface DeviceSpecification {  
  
 String getBrand();  
 String getModel();  
 double getPrice();  
}

Now, every class inherited from the Device is closed to change, but open to addition. Each class has only its own task, and is only responsible for its own type of object, as shown earlier. Also, the Liskov principle is respected, because deleting the Device class does not change anything, because its subclasses can continue to work without it, and the subclasses of these classes can also work without their superclasses.

Speaking of the rule of interfaces, we will touch on this topic in the pattern part, talking about the abstract factory.

About dependency inversion, using interfaces, classes, and their methods are all built in such a way that essentially classes using an interface can do the same action in different ways.

public interface GFactory {  
 Headphones createHeadphones(String brand, String model, double price, String type, String audioQuality, boolean noiseCancel);  
 Smartwatch createSmartwatch(String brand, String model, double price, String type, String operatingSystem, boolean fitnessTracker, boolean heartRate);  
}

In this case, thanks to this interface, a class can generate two different objects, and their generation can also be done using the constructors of these very classes. Simply put, the top-level modules are independent of the lower-level modules. The parts depend on the abstractions.

## **SQL connections.**

Continuing with the S.O.L.I.D. theme, consider connecting to a database. The basis for the connection was PostgreSQL, in which all tables were generated. Let's look at the classes.

To comply with S.O.L.I.D. principles, the Market class was assembled from different pieces of CRUD methods. That is, each class is only responsible for its own part.

public class MarketCreate {

public void initDevice(String login, int choice) {  
 Scanner scanner = new Scanner(System.*in*);  
 switch (choice) {  
 case 1 -> {  
 System.*out*.println("Enter brand: ");  
 String brand = scanner.nextLine();  
 System.*out*.println("Enter model: ");  
 String model = scanner.nextLine();  
 System.*out*.println("Enter price: ");  
 double price = Double.*parseDouble*(scanner.nextLine());  
 System.*out*.println("Enter type: ");  
 String type = scanner.nextLine();  
 System.*out*.println("Enter Audio Quality");  
 String audioQuality = scanner.nextLine();  
 System.*out*.println("Enter Noise Cancel (true/false): ");  
 boolean noiseCancel = Boolean.*parseBoolean*(scanner.nextLine());  
 Headphones headphones = *gadgetsFactory*.createHeadphones(brand, model, price, type, audioQuality, noiseCancel);  
 try {  
 Class.*forName*("org.postgresql.Driver");  
 Connection con = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/as3help", "postgres", "password");  
 String sql = "INSERT INTO public.\"Headphones\" (brand, model, price, type, audio\_quality, noise\_cancel, seller) VALUES (?, ?, ?, ?, ?, ?, ?);";  
 PreparedStatement st = con.prepareStatement(sql);  
 st.setString(1, headphones.getBrand());  
 st.setString(2, headphones.getModel());  
 st.setDouble(3, headphones.getPrice());  
 st.setString(4, headphones.getType());  
 st.setString(5, headphones.getAudioQuality());  
 st.setBoolean(6, headphones.isNoiseCancel());  
 st.setString(7, login);  
 st.executeUpdate();  
 con.close();  
 } catch (Exception e) {  
 System.*out*.println("exception: " + e.getMessage());  
 }  
 }

public class MarketRead extends MarketCreate{

public void readDevices(String tableName) {  
 try {  
 Class.*forName*("org.postgresql.Driver");  
 Connection con = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/as3help", "postgres", "password");  
 Statement st = con.createStatement();  
 ResultSet rs = st.executeQuery("SELECT \* FROM public.\"" + tableName + "\"");  
 while (rs.next()) {  
 System.*out*.println("ID: "+rs.getInt("id"));  
 System.*out*.println("Model: "+rs.getString("model"));  
 System.*out*.println("----------------------");  
 }  
 con.close();  
 } catch (Exception e) {  
 System.*out*.println("exception: " + e.getMessage());  
 }  
}

public class MarketUpdate extends MarketRead{  
 public void updateProduct(int id, String category) {  
 Scanner scanner = new Scanner(System.*in*);  
 try {  
 Class.*forName*("org.postgresql.Driver");  
 Connection con = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/as3help", "postgres", "password");  
 PreparedStatement preparedStatement;  
 switch (category) {  
 case "Headphones" -> {  
 System.*out*.println("Enter the new brand: ");  
 String newBrand = scanner.nextLine();  
 System.*out*.println("Enter the new model: ");  
 String newModel = scanner.nextLine();  
 System.*out*.println("Enter the new price: ");  
 double newPrice = scanner.nextDouble();  
 scanner.nextLine(); // consume the line separator  
 System.*out*.println("Enter the new type: ");  
 String type = scanner.nextLine();  
 System.*out*.println("Enter the new audio quality: ");  
 String newAudio\_quality = scanner.nextLine();  
 System.*out*.println("Enter the new noise cancelling support (true/false): ");  
 boolean newNoise\_Cancel = scanner.nextBoolean();  
 preparedStatement = con.prepareStatement("UPDATE public.\"Headphones\" SET brand = ?, model = ?, price = ?, type = ?, audio\_quality = ?, noise\_cancel = ? WHERE id = ?");  
 preparedStatement.setString(1, newBrand);  
 preparedStatement.setString(2, newModel);  
 preparedStatement.setDouble(3, newPrice);  
 preparedStatement.setString(4, type);  
 preparedStatement.setString(5, newAudio\_quality);  
 preparedStatement.setBoolean(6, newNoise\_Cancel);  
 preparedStatement.setInt(7, id);

public class MarketDelete extends MarketUpdate{  
 public void deleteDevice(String category, int id) {  
 try {  
 Class.*forName*("org.postgresql.Driver");  
 Connection con = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/as3help", "postgres", "password");  
 con.setAutoCommit(false);  
 PreparedStatement st = con.prepareStatement("DELETE FROM public.\""+category+"\" WHERE id = ?");  
 st.setInt(1, id);  
 int rowsAffected = st.executeUpdate();  
 if (rowsAffected > 0) {  
 con.commit();  
 } else {  
 con.rollback();  
 System.*out*.println("No record was deleted. Check if the ID and category are correct.");  
 }  
 st.close();  
 con.close();  
 } catch (Exception e) {  
 System.*out*.println("An error occurred while deleting the device: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }

public class Market extends MarketSort {  
 private static Market *instance* = null;

Market Sort has been omitted for the purposes of readability of the text. We will parse this class later.

Each class has its own methods responsible for creating, reading, updating and deleting tables. The variety of these methods is large, but the point remains that each class is only responsible for its own part. At the end, we have created a class Market, through which we will operate with all SQL queries.

## **Design patterns.**

Our project uses two design patterns. Let's look at the simplest one.

To make it easier to work with the Market class, we have implemented Singleton for this class. That is, we need only one object of this class to exist, and this object must be reachable anywhere.

private static Market *instance* = null;  
  
private Market() {}  
  
public static Market getInstance() {  
 if (*instance* == null) {  
 *instance* = new Market();  
 }  
 return *instance*;  
}

public static Market *market* = Market.*getInstance*();

In this way, we ensure that there is only one object of a given class.

The next design is an abstract factory. The basics were already established in the hierarchy, and also, all of the S.O.L.I.D. principles of the project were prepared for this particular pattern.

There are three different interfaces for different factories. Phones, gadgets and tablets.

public interface GFactory {  
 Headphones createHeadphones(String brand, String model, double price, String type, String audioQuality, boolean noiseCancel);  
 Smartwatch createSmartwatch(String brand, String model, double price, String type, String operatingSystem, boolean fitnessTracker, boolean heartRate);  
}

public interface MFactory {  
 MobilePhone createMobilePhone(String brand, String model, double price, int ram, int memory, String screenSize, String camera, boolean dualSim);  
 Smartphone createSmartphone(String brand, String model, double price, int ram, int memory, String screenSize, String camera, String OS, boolean faceID, boolean fingerprintSensor);  
}

public interface TFactory {  
 BusinessTablet createBusinessTablet(String brand, String model, double price, boolean stylusSup, boolean keyboardSup, String screenRatio, boolean fingerprintScanner, boolean facialRecognition);  
 DrawingTablet createDrawingTablet(String brand, String model, double price, boolean stylusSup, boolean keyboardSup, String screenRatio, int pressureSensitivity, boolean eraser);  
 GamingTablet createGamingTablet(String brand, String model, double price, boolean stylusSup, boolean keyboardSup, String screenRatio, int memory, int ram, String gpu, boolean vrSupport);  
}

Then, classes were created to implement these interfaces.

public class GadgetsFactory implements GFactory {  
 @Override  
 public Headphones createHeadphones(String brand, String model, double price, String type, String audioQuality, boolean noiseCancel) {  
 return new Headphones(brand, model, price, type, audioQuality, noiseCancel);  
 }  
  
 @Override  
 public Smartwatch createSmartwatch(String brand, String model, double price, String type, String operatingSystem, boolean fitnessTracker, boolean heartRate) {  
 return new Smartwatch(brand, model, price, type, operatingSystem, fitnessTracker, heartRate);  
 }  
}

and other similar classes...

Now, we can generate objects of these classes using a single class, i.e. we can create related objects using abstractions.

Headphones headphones = *gadgetsFactory*.createHeadphones(brand, model, price, type, audioQuality, noiseCancel);

Smartwatch smartwatch = *gadgetsFactory*.createSmartwatch(brand, model, price, type, operatingSystem, fitnessTracker, heartRate);

## **Console interface.**

As previously stated, the basis of programme management is the console interface. It has been implemented with methods that we call depending on the functions we need. The usage is divided into two different branches - managing being a seller, and being a buyer.

public static void login() {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("Time to sign up! Are you buyer (1), or a seller? (2)");  
 int choice = scanner.nextInt();  
 if (choice == 1) {  
 *buyerPart*();  
 } else if (choice == 2) {  
 *sellerPart*();  
 } else {  
 System.*out*.println("Invalid choice, please try again.");  
 *login*();  
 }  
}

public static void sellerPart() {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("--- LIST OF SELLERS ---");  
 *market*.showSellers();  
 System.*out*.println("0. Get back to login. ");  
 System.*out*.println("1. Create a new seller. ");  
 System.*out*.println("2. Login. ");  
 int dilemma = scanner.nextInt();  
 if(dilemma==0){  
 *login*();  
 }  
 else if(dilemma==1){  
 *newSeller*();  
 }  
 else if(dilemma==2){  
 *getIn*();  
 }  
 else{  
 System.*out*.println("Invalid option. ");  
 *sellerPart*();  
 }  
  
}

public static void buyerPart() {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("--- BUYER PART ---");  
 System.*out*.println("Choose the product category you are interested in:");  
 System.*out*.println("1. Headphones");  
 System.*out*.println("2. Smartwatch");  
 System.*out*.println("3. Gaming tablet");  
 System.*out*.println("4. Drawing tablet");  
 System.*out*.println("5. Business tablet");  
 System.*out*.println("6. Mobile phone");  
 System.*out*.println("7. Smartphone");  
 System.*out*.println("8. Print basket: ");  
 System.*out*.println("0. Get back to login: ");  
 System.*out*.println("Enter the category:");  
 int choice = scanner.nextInt();  
 String category = scanner.nextLine();

In the seller part, we first have to log in to our account, there is a separate SQL query for this. If the data is incorrect, you will not be able to log in to the account. In this branch, we can manage items on behalf of the seller using CRUD methods.

On the buyer side, we can buy objects, view each object, and sort the objects based on our needs.

A shopping cart has also been implemented to clearly see what has already been added to the cart and to wait for confirmation of the purchase.

public static void printBasket(){  
 Scanner scanner = new Scanner(System.*in*);  
 double total = 0;  
 for (Double price : *prices*) {  
 total += price;  
 }  
 for (int i = 0; i < *devices*.size(); i++) {  
 System.*out*.println("Product "+ (i+1));  
 System.*out*.println("Model "+*devices*.get(i));  
 System.*out*.println("ID "+ *productNumbs*.get(i));  
 System.*out*.println("Price "+*prices*.get(i));  
 System.*out*.println("--------------------");  
 }  
 System.*out*.println("Total price: "+total);  
 System.*out*.println("0. Get Back");  
 System.*out*.println("1. Make a purchase.");  
 System.*out*.println("2. Delete products.");  
 int des = scanner.nextInt();  
 if(des==1){  
 *confirm*();  
 }  
 else if(des==2){  
 *remove*();  
 }  
}

public static void confirm() {  
 Scanner scanner = new Scanner(System.*in*);  
 double total = 0;  
 for (Double price : *prices*) {  
 total += price;  
 }  
 System.*out*.println("Total price: " + total);  
 System.*out*.println("Confirm purchase (Y/N): ");  
 String confirm = scanner.nextLine();  
 if (confirm.equals("Y")) {  
 System.*out*.println("The purchase was successful. Have a nice day!");  
 try {  
 for (int i = 0; i < *devices*.size(); i++) {  
 String category = *categories*.get(i);  
 int id = *productNumbs*.get(i);  
 *market*.deleteDevice(category, id);  
 }  
 *devices*.clear();  
 *categories*.clear();  
 *productNumbs*.clear();  
 *prices*.clear();  
 } catch (IndexOutOfBoundsException e) {  
 *buyerPart*();  
 }  
 } else {  
 *printBasket*();  
 }  
}

Many other methods will be shown on the defence, in order to save space in the report.

## **Final Output.**

When we start a project, we can manage the marketplace, depending on our objectives.



Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

# **Conclusion**

It was a long process and we had a lot of challenges during the development phase, but it was a really good experience that we will use for future projects. In our opinion, we have implemented everything we need for the marketplace. At the time of writing the report, the management of the marketplace is very simple and straightforward. We are grateful for the university's OOP course, and will definitely use this theory for future developments.

**Reference list**

Brookes, S., Main, M., Melton, A., Mislove, M., & Schmidt, D. (1994). *Mathematical Foundations of Programming Semantics: 9th International Conference, New Orleans, LA, USA, April 7 - 10, 1993. Proceedings*. Springer Science & Business Media.

Singh, N., Chouhan, S. S., & Verma, K. (2021). Object Oriented Programming: Concepts, Limitations and Application Trends. *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*. https://doi.org/10.1109/iscon52037.2021.9702463

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *OOPS Messenger*, *1*(1), 7–87. https://doi.org/10.1145/382192.383004