CSE 351 Design Patterns Term Project

Guitar Store

20200808004 – Tevfik KESİCİ
20190808062 – Hakan ZAVRAK

Statement of Work

In this project, we are going to operate a guitar store that operates a little differently than others. We have noticed that only 2% of people who try our guitars actually end up buying them, so we decided to use artificial intelligence employees instead of wasting the time of our real employees. The solution we found required solving software design and development problems because we now need a software to manage our store.

First, we started with a simple guitar store, but when we wanted to expand our store, some problems occurred. Most people in Istanbul wanted different models of the same brands, while people in Izmir wanted something else. For example, in our Izmir store, we sell new, trending guitars. In the Istanbul store, unlike Izmir, we sell old-school guitars from the same manufacturer. An example from the ESP brand, in Istanbul, the meaning of ESP was the ESP Eclipse II, but in Izmir, it was the ESP Snakebyte.

One other problem is the modifications you want to make when you want to buy the guitar. Adding accessories that we would like to buy with the guitar led to a class explosion. Every add-on we added and every price update we wanted to make was violating the open-closed principle.

One more problem, We have only one AIEmployee per store, so our operations should proceed on only one instance of an AIEmployee. we wanted to make sure there was at most one instance of our AIEmployee class because we needed to ensure that there were no problems in transactions between clients or inconsistent results.

The last problem is that our AIEmployee, which communicates with customers, gives an error when we don’t have a requested guitar.

Design Patterns

- **Factory Method**: The factory method pattern was the key to solving this problem because it is useful when we need a complicated construction process for constructing the object, which in our case is a guitar. While Istanbul demands different models, Izmir also demands different models. One other benefit of using the factory method pattern is that when one of the stores orders a guitar, that store does not need to know how the guitar is made. The guitar factory handles these things, which is why we chose the factory method pattern.

  We have a playGuitar() method, in which the customer passes the brand of the guitar as a parameter that he or she wants. Then, the chooseGuitar() method is called, which is defined in the abstract GuitarStore class and implemented in the IstanbulGuitarStore and IzmirGuitarStore classes. The selected guitar is then returned to the playGuitar() method, which has no idea what kind of guitar was created.

- **Decorator:** The reason why we chose this is that the decorator pattern is useful when adding additional functionality to an existing object, which is a guitar. In other words, it is instantiated as a class at runtime and dynamically changes the functionality of our guitar object at runtime without impacting the other existing functionality. After adding this pattern, we can easily change the prices of the modifications. And we don't need to calculate every time; we can simply wrap our guitar with these modifications.

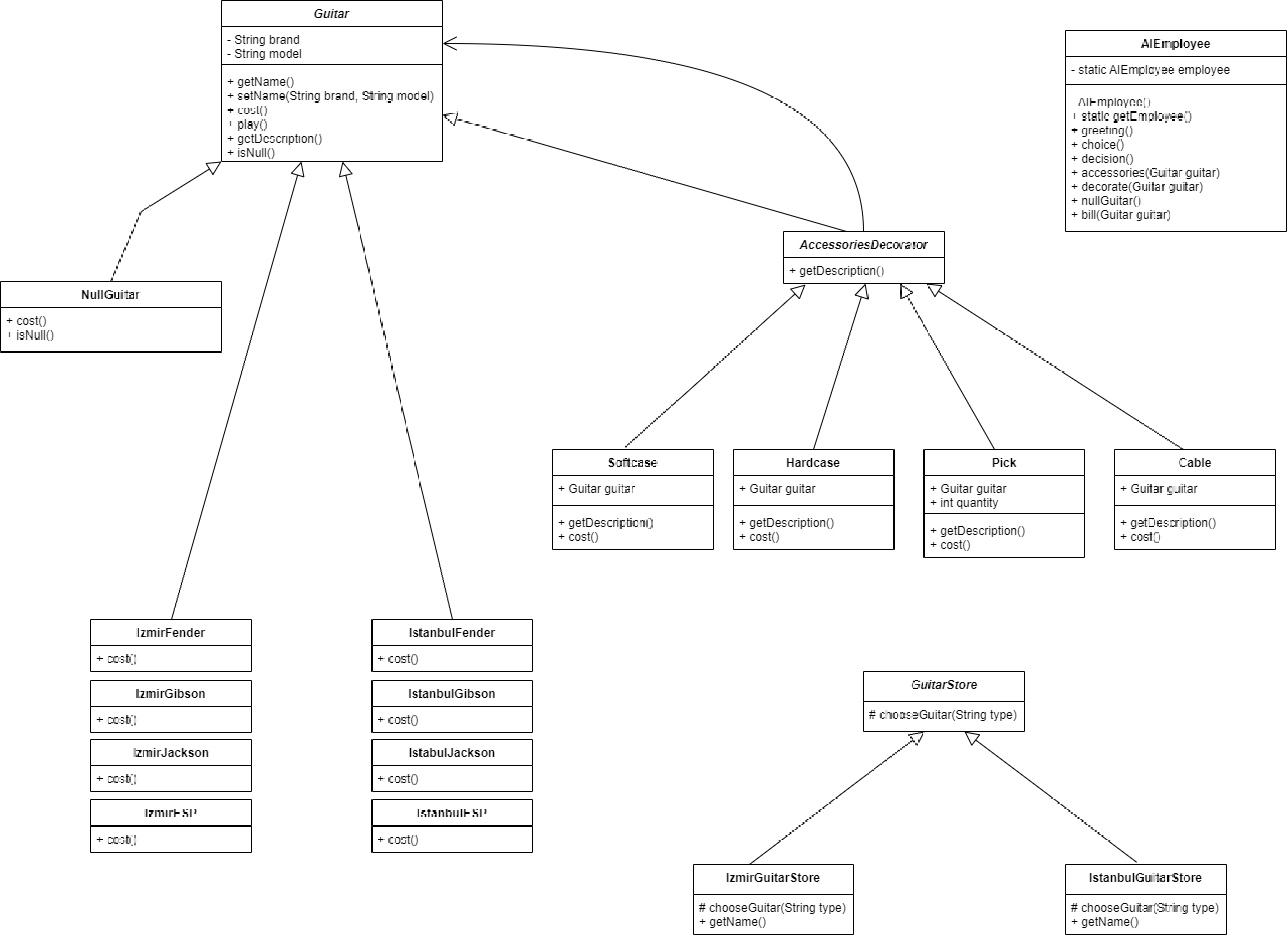
  We have an abstract AccessoriesDecorator class that extends our Guitar class for type matching. We also have concrete accessories classes that extend the AccessoriesDecorator class. We add new accessories by composing a decorator with our guitar object using composition. This allows us to dynamically change our guitar object at runtime. Additionally, we have made our guitars extend the Component class, which is the Guitar class. This allows us to wrap our component (the guitar) with any number of decorators, providing flexibility in customization.

- **Singleton:** We needed a design pattern that restricts a class from instantiating its multiple objects, so we chose the singleton pattern.

  In our public AIEmployee class, we have a private constructor and a static getEmployee() method. To deal with multithreading, we used synchronization because we don't have any performance issues due to the small number of AIEmployee instances.

- **Null Object:** If our store does not have requested guitar, we will handle this by our NullGuitar, it will reflect nothing and exit the system.

  We created a NullGuitar object that is inherited from the Guitar object, and we have overridden the isNull() method to make it return true. In this way, if our factories are unable to create any guitar with the given types, our factory will return a NullGuitar object. We will then handle non-existing guitar types by using the isNull() method for our guitars.


Guitar
- String brand
- String model
+ getName()
+ setName(String brand, String model)
+ cost()
+ play()
+ getDescription()
+ isNull()
NullGuitar
+ cost()
+ isNull()
AIEmployee
- static AIEmployee employee
- AIEmployee()
+ static getEmployee()
+ greeting()
+ choice()
+ decision()
+ accessories(Guitar guitar)
+ decorate(Guitar guitar)
+ nullGuitar()
+ bill(Guitar guitar)
AccessoriesDecorator
+ getDescription()
Softcase
+ Guitar guitar
+ getDescription()
+ cost()
Hardcase
+ Guitar guitar
+ getDescription()
+ cost()
Pick
+ Guitar guitar
+ int quantity
+ getDescription()
+ cost()
Cable
+ Guitar guitar
+ getDescription()
+ cost()
IzmirFender
+ cost()
IzmirGibson
+ cost()
IzmirJackson
+ cost()
IzmirESP
+ cost()
IstanbulFender
+ cost()
IstanbulGibson
+ cost()
IstabulJackson
+ cost()
IstanbulESP
+ cost()
GuitarStore
# chooseGuitar(String type)
IzmirGuitarStore
# chooseGuitar(String type)
+ getName()
IstanbulGuitarStore
# chooseGuitar(String type)
+ getName()

CSE 351 Design Patterns

Term Project Presentation

Tevfik Kesici & Hakan Zavrak

With AI

Guitar Store

Project Definition

We are are going to oparate a guitar store.

Istanbul Store

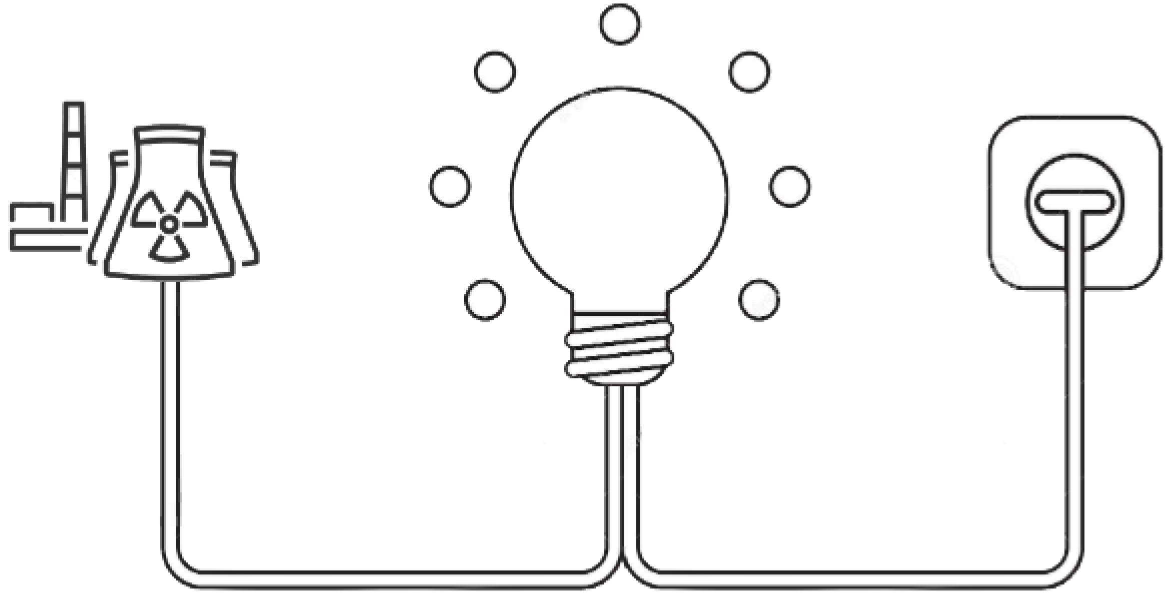
Old-school models of the same manufacturer

İzmir Store

Trending models of the same manufacturer

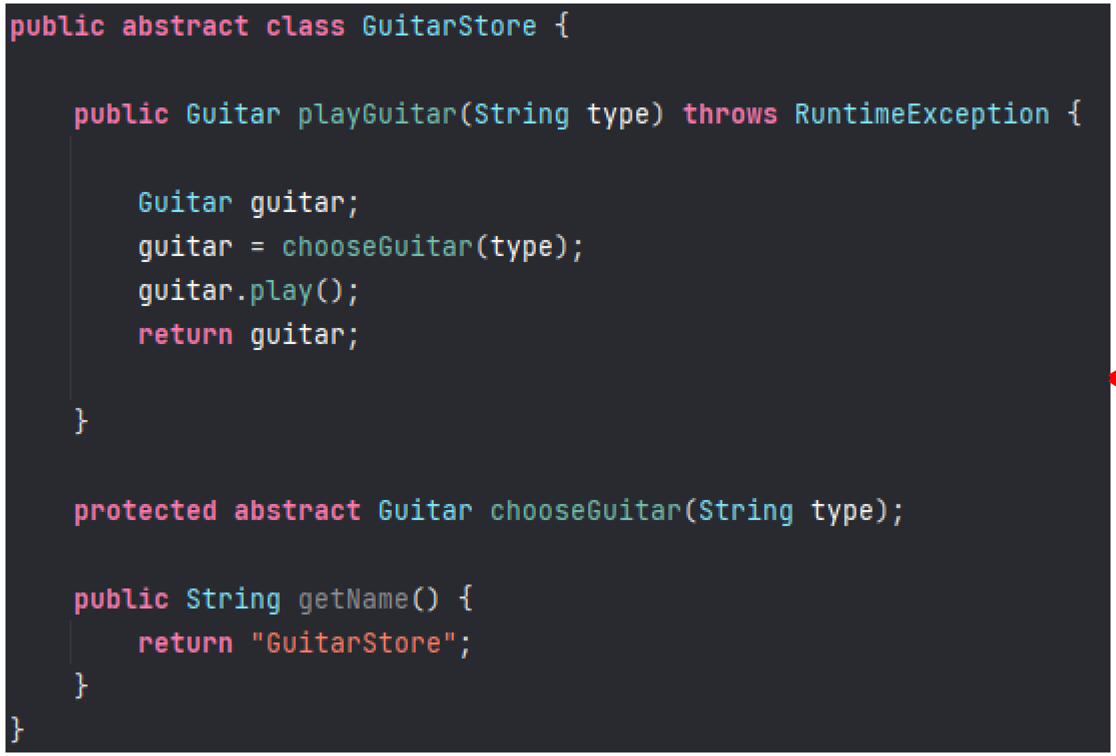
Problems

Tight Coupling

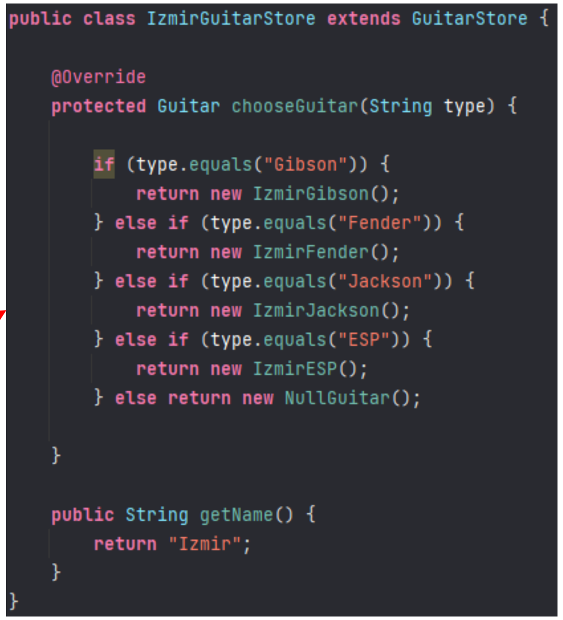


''You don't have to learn how to wire things up if you need to turn on the lights. You use a socket for that.''

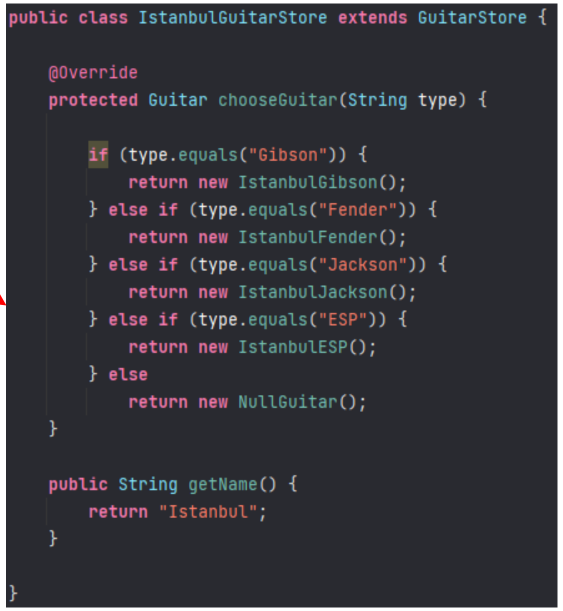
Solution



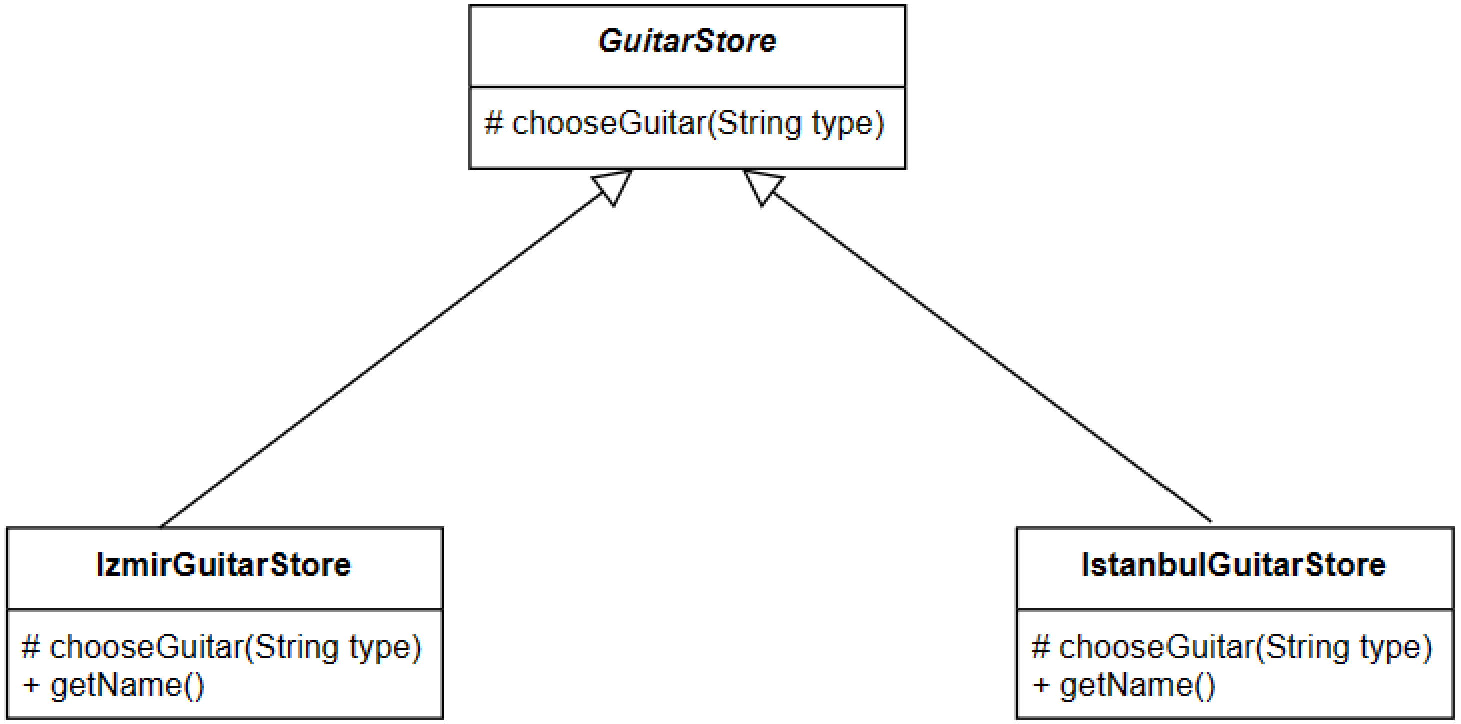
```java
public abstract class GuitarStore {

    public Guitar playGuitar(String type) throws RuntimeException {

        Guitar guitar;
        guitar = chooseGuitar(type);
        guitar.play();
        return guitar;

    }

    protected abstract Guitar chooseGuitar(String type);

    public String getName() {
        return "GuitarStore";
    }
}
```



```java
public class IzmirGuitarStore extends GuitarStore {

    @Override
    protected Guitar chooseGuitar(String type) {

        if (type.equals("Gibson")) {
            return new IzmirGibson();
        } else if (type.equals("Fender")) {
            return new IzmirFender();
        } else if (type.equals("Jackson")) {
            return new IzmirJackson();
        } else if (type.equals("ESP")) {
            return new IzmirESP();
        } else return new NullGuitar();

    }

    public String getName() {
        return "Izmir";
    }
}
```



```java
public class IstanbulGuitarStore extends GuitarStore {

    @Override
    protected Guitar chooseGuitar(String type) {

        if (type.equals("Gibson")) {
            return new IstanbulGibson();
        } else if (type.equals("Fender")) {
            return new IstanbulFender();
        } else if (type.equals("Jackson")) {
            return new IstanbulJackson();
        } else if (type.equals("ESP")) {
            return new IstanbulESP();
        } else
            return new NullGuitar();
    }

    public String getName() {
        return "Istanbul";
    }

}
```


GuitarStore
# chooseGuitar(String type)
IzmirGuitarStore
# chooseGuitar(String type)
+ getName()
IstanbulGuitarStore
# chooseGuitar(String type)
+ getName()

Conclusion

Factory method pattern was the key

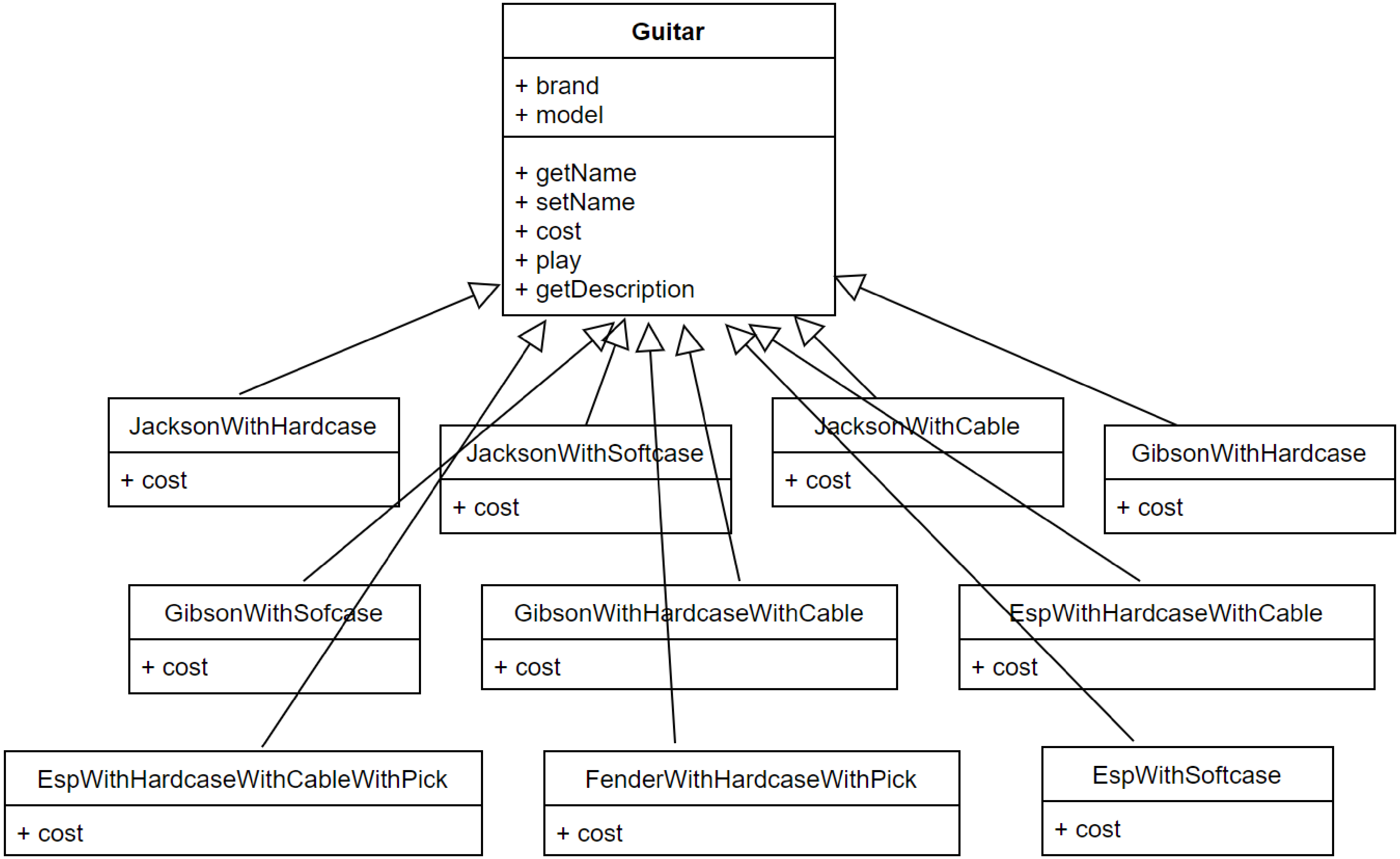
- Object creation is delegated to subclasses

- Subclasses implement the factory method to create objects

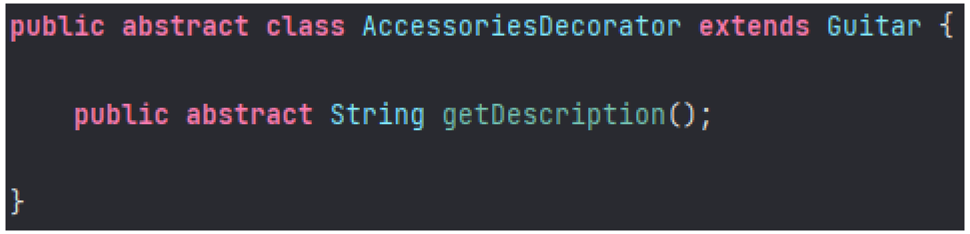
- Handles complicated construction process without impacting the other existing functionality

Problems

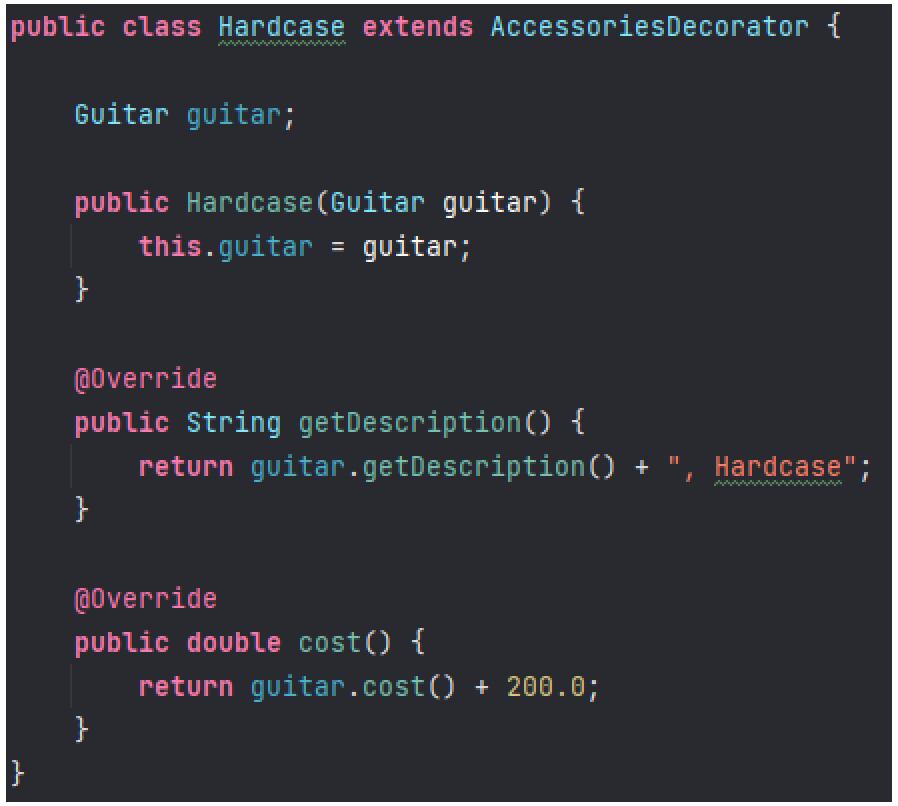
Adding Accessories


Guitar
+ brand
+ model
+ getName
+ setName
+ cost
+ play
+ getDescription
JacksonWithHardcase
+ cost
JacksonWithSoftcase
+ cost
JacksonWithCable
+ cost
GibsonWithHardcase
+ cost
GibsonWithSofcase
+ cost
GibsonWithHardcaseWithCable
+ cost
EspWithHardcaseWithCable
+ cost
EspWithHardcaseWithCableWithPick
+ cost
FenderWithHardcaseWithPick
+ cost
EspWithSoftcase
+ cost

Solution



```java
public abstract class AccessoriesDecorator extends Guitar {

    public abstract String getDescription();

}
```

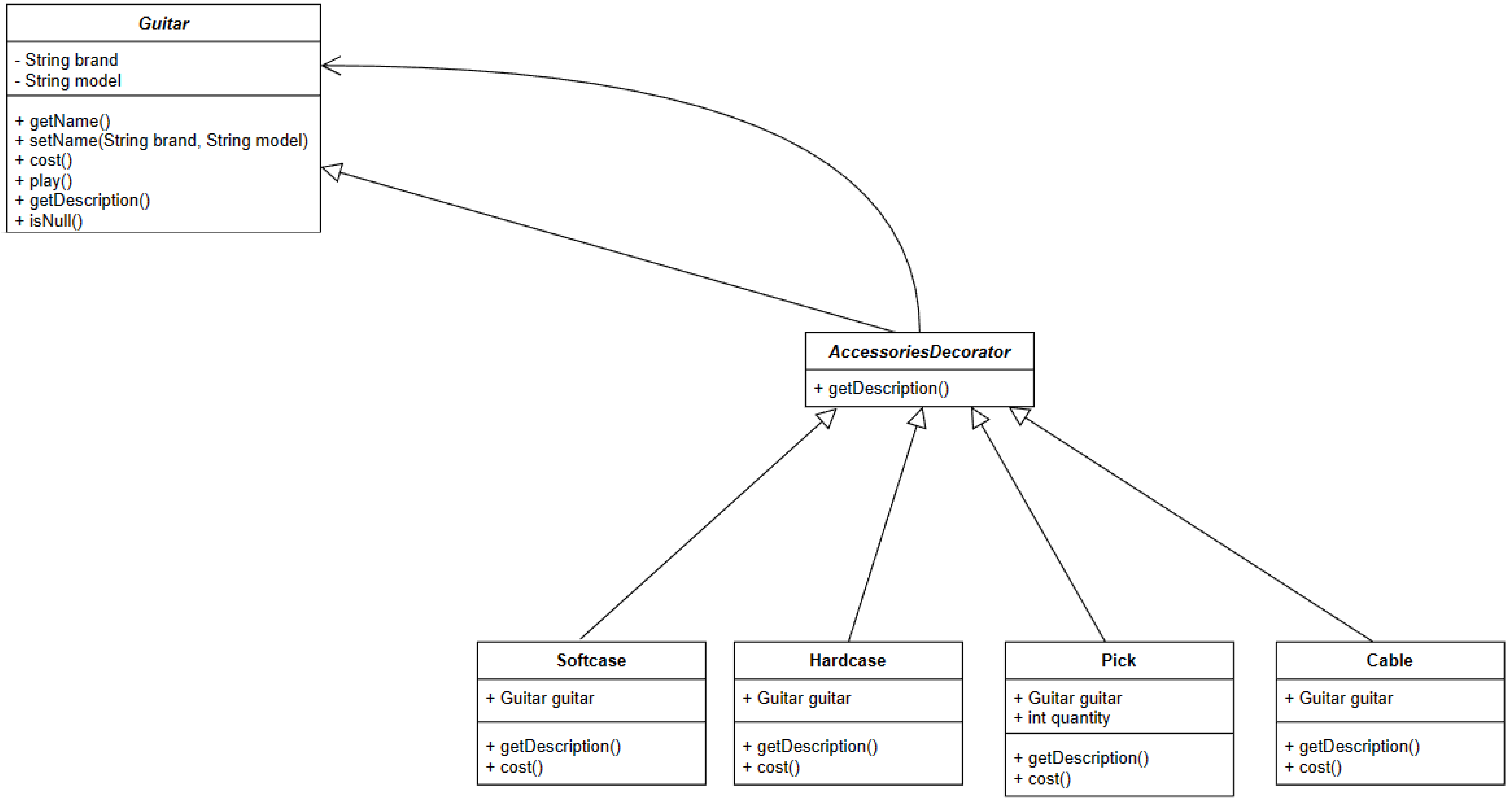


```java
public class Hardcase extends AccessoriesDecorator {

    Guitar guitar;

    public Hardcase(Guitar guitar) {
        this.guitar = guitar;
    }

    @Override
    public String getDescription() {
        return guitar.getDescription() + ", Hardcase";
    }

    @Override
    public double cost() {
        return guitar.cost() + 200.0;
    }
}
```

+

guitar.getDescription()

Hardcase


Guitar
- String brand
- String model
+ getName()
+ setName(String brand, String model)
+ cost()
+ play()
+ getDescription()
+ isNull()
AccessoriesDecorator
+ getDescription()
Softcase
+ Guitar guitar
+ getDescription()
+ cost()
Hardcase
+ Guitar guitar
+ getDescription()
+ cost()
Pick
+ Guitar guitar
+ int quantity
+ getDescription()
+ cost()
Cable
+ Guitar guitar
+ getDescription()
+ cost()

Conclusion

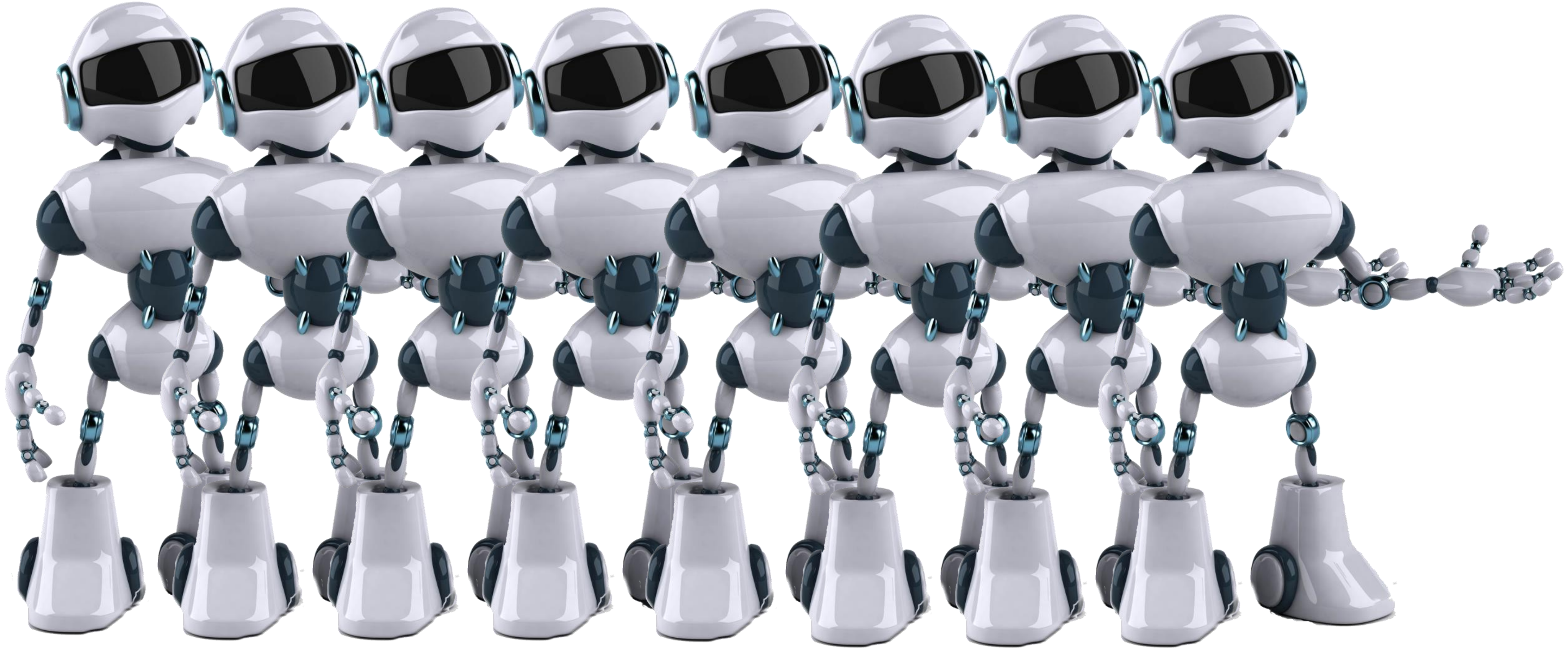
Decorator pattern was the key

- Composition is used to add new behaviors at runtime

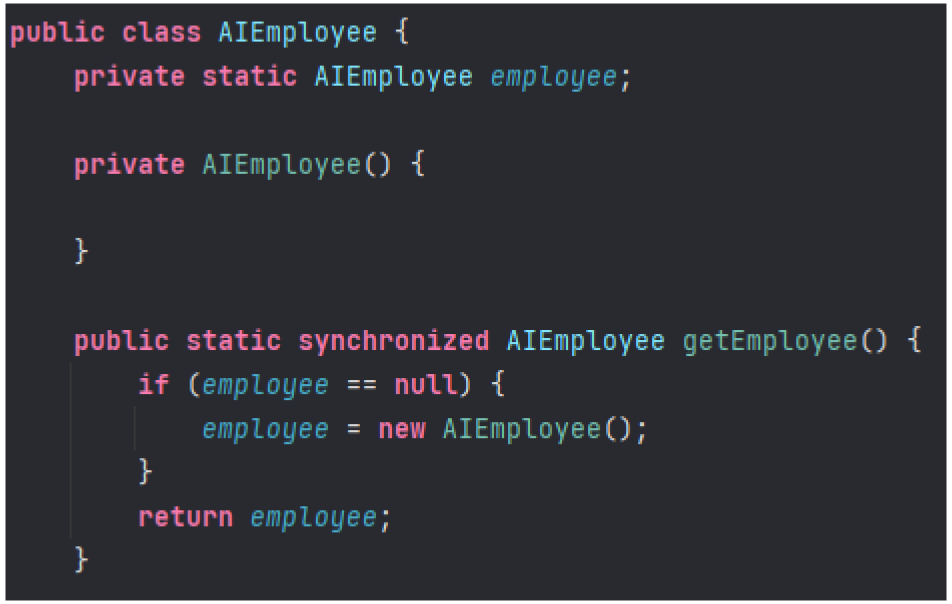
- Using inheritance for type matching allows our decorator classes to have the same type as our guitar

Problems

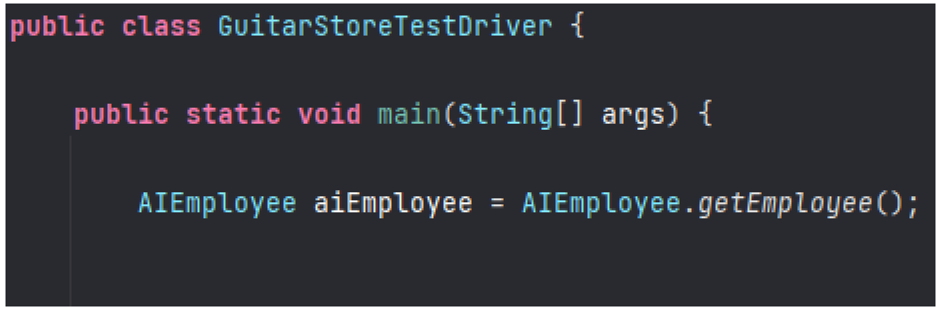
Possibly Multiple AIEmployees



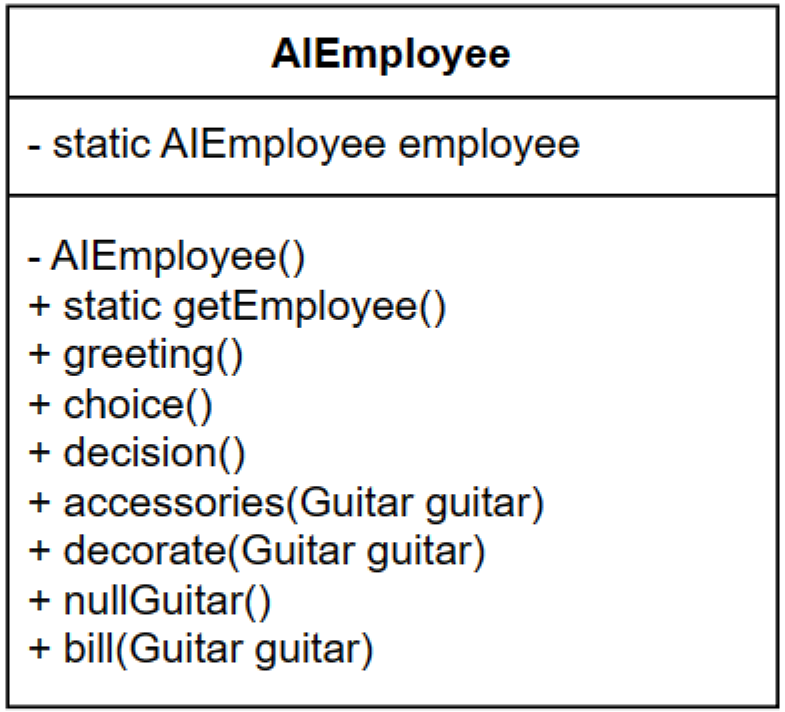
Solution



```java
public class AIEmployee {
    private static AIEmployee employee;

    private AIEmployee() {

    }

    public static synchronized AIEmployee getEmployee() {
        if (employee == null) {
            employee = new AIEmployee();
        }
        return employee;
    }
```



```java
public class GuitarStoreTestDriver {

    public static void main(String[] args) {

        AIEmployee aiEmployee = AIEmployee.getEmployee();

```


AIEmployee
- static AIEmployee employee
- AIEmployee()
+ static getEmployee()
+ greeting()
+ choice()
+ decision()
+ accessories(Guitar guitar)
+ decorate(Guitar guitar)
+ nullGuitar()
+ bill(Guitar guitar)

Conclusion

Singleton pattern was the key

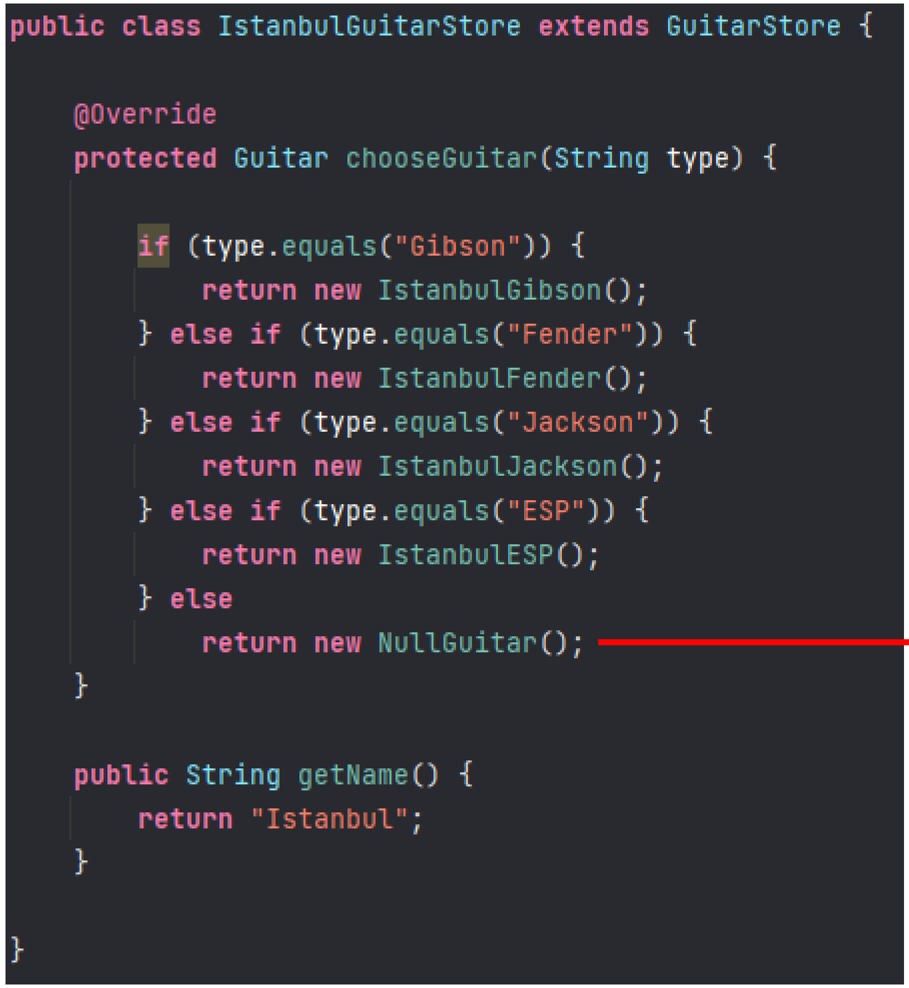
- We can be sure that we will have at most one instance of the AIEmployee class

- We don't have any performance issues with synchronizing the method.

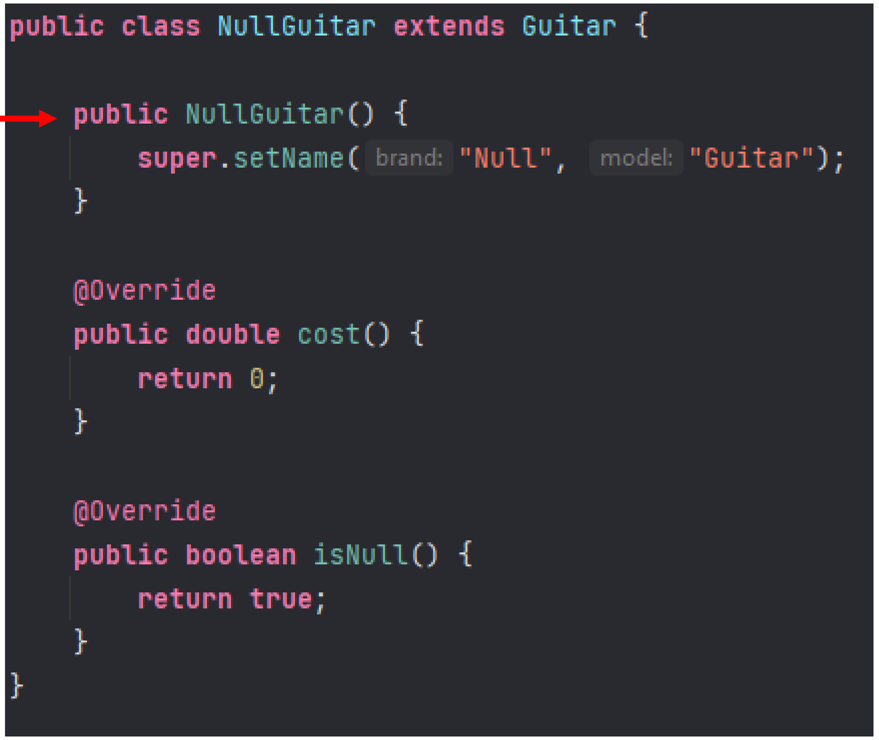
Problems

Unavailable Guitars

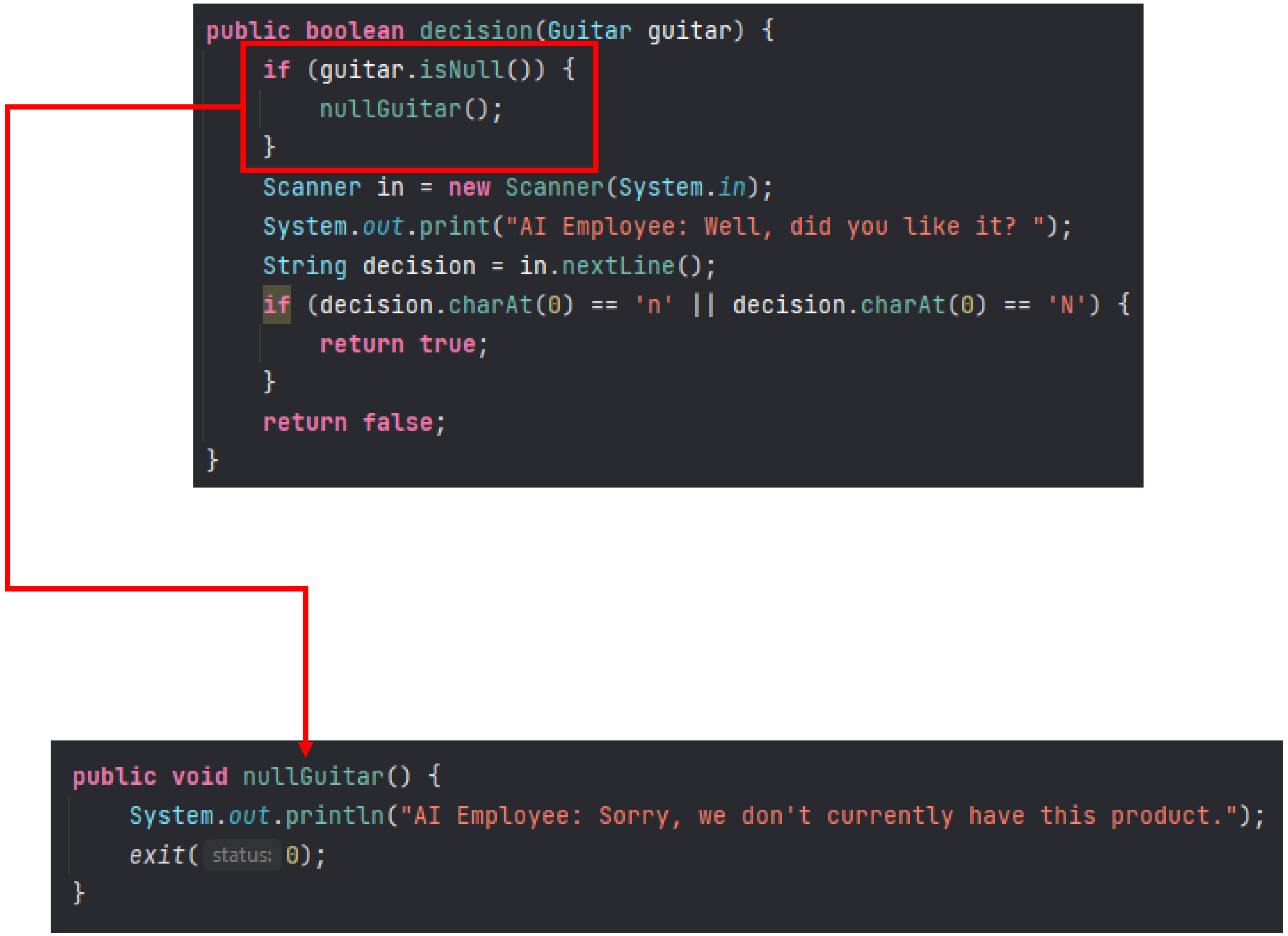
Solution



```java
public class IstanbulGuitarStore extends GuitarStore {

    @Override
    protected Guitar chooseGuitar(String type) {

        if (type.equals("Gibson")) {
            return new IstanbulGibson();
        } else if (type.equals("Fender")) {
            return new IstanbulFender();
        } else if (type.equals("Jackson")) {
            return new IstanbulJackson();
        } else if (type.equals("ESP")) {
            return new IstanbulESP();
        } else
            return new NullGuitar();
    }

    public String getName() {
        return "Istanbul";
    }

}
```

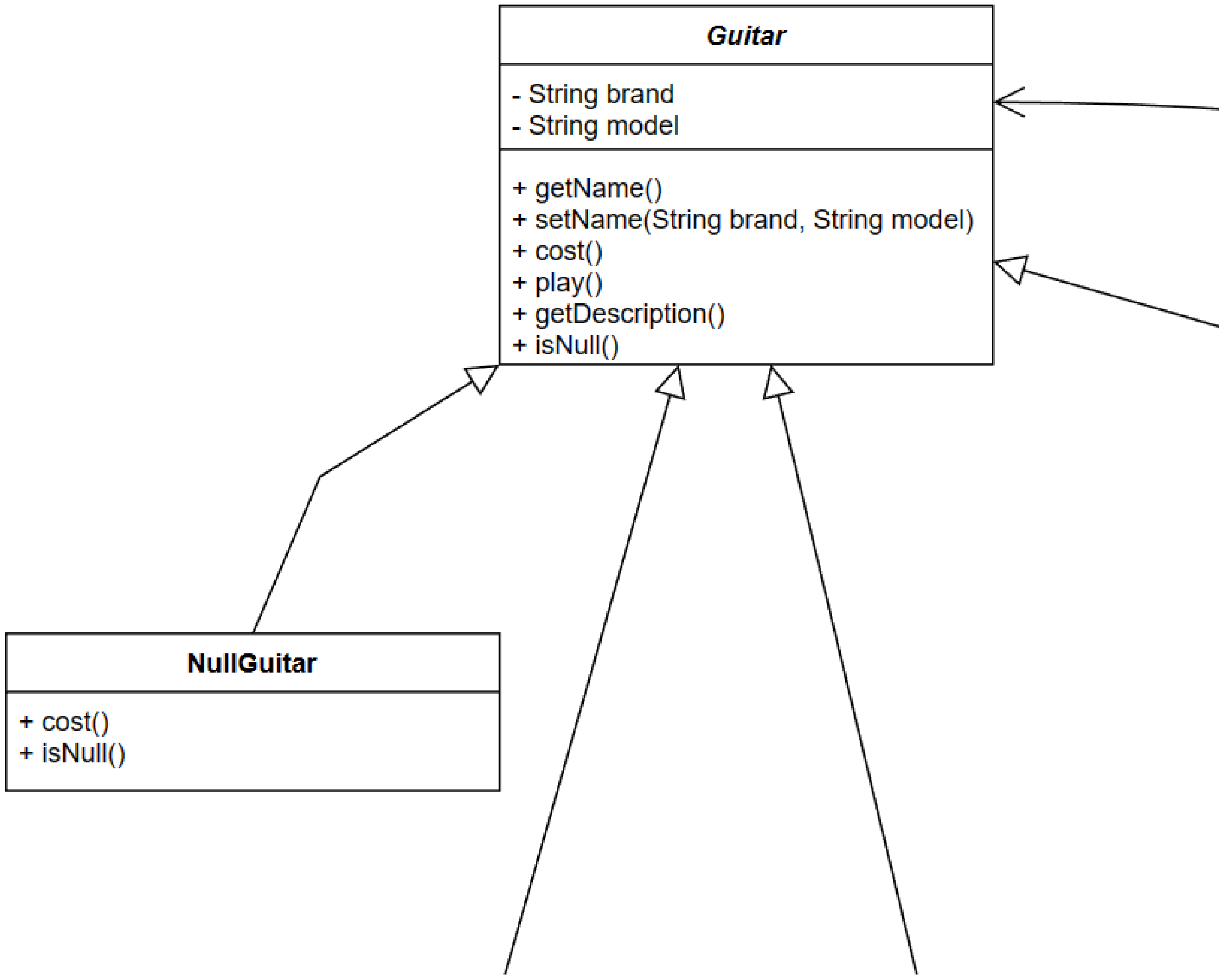


```java
public class NullGuitar extends Guitar {

    public NullGuitar() {
        super.setName( brand: "Null",  model: "Guitar");
    }

    @Override
    public double cost() {
        return 0;
    }

    @Override
    public boolean isNull() {
        return true;
    }
}
```



```java
public boolean decision(Guitar guitar) {
    if (guitar.isNull()) {
        nullGuitar();
    }
    Scanner in = new Scanner(System.in);
    System.out.print("AI Employee: Well, did you like it? ");
    String decision = in.nextLine();
    if (decision.charAt(0) == 'n' || decision.charAt(0) == 'N') {
        return true;
    }
    return false;
}
```

```java
public void nullGuitar() {
    System.out.println("AI Employee: Sorry, we don't currently have this product.");
    exit( status: 0);
}
```

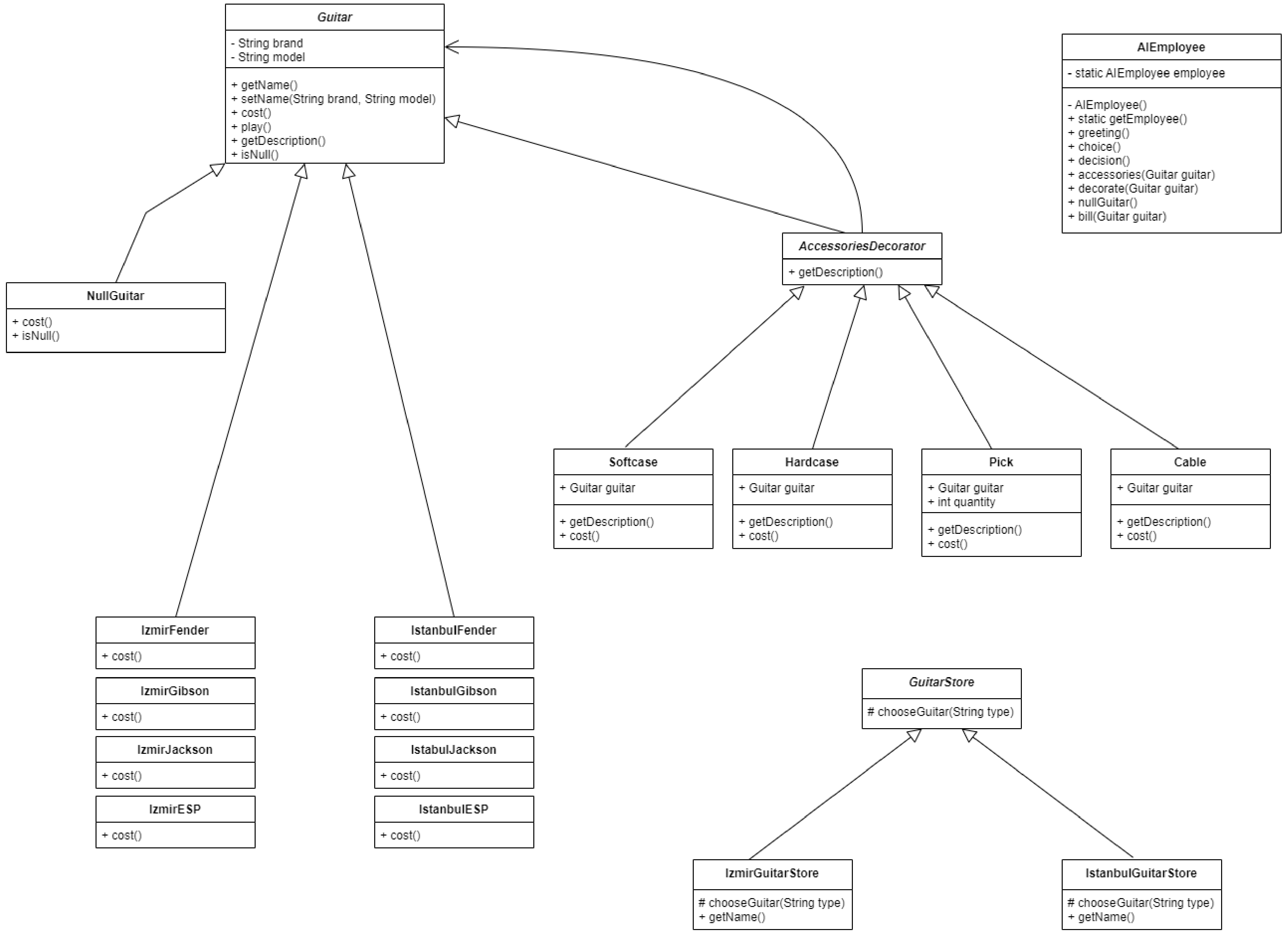

Guitar
- String brand
- String model
+ getName()
+ setName(String brand, String model)
+ cost()
+ play()
+ getDescription()
+ isNull()
NullGuitar
+ cost()
+ isNull()

Conclusion

Null Object pattern was the key

- If we don't have the guitar brand requested from the user in our store, it makes our job easier to create a null guitar and handle it within other methods.

Complete UML Diagram


Guitar
- String brand
- String model
+ getName()
+ setName(String brand, String model)
+ cost()
+ play()
+ getDescription()
+ isNull()
AIEmployee
- static AIEmployee employee
- AIEmployee()
+ static getEmployee()
+ greeting()
+ choice()
+ decision()
+ accessories(Guitar guitar)
+ decorate(Guitar guitar)
+ nullGuitar()
+ bill(Guitar guitar)
AccessoriesDecorator
+ getDescription()
NullGuitar
+ cost()
+ isNull()
Softcase
+ Guitar guitar
+ getDescription()
+ cost()
Hardcase
+ Guitar guitar
+ getDescription()
+ cost()
Pick
+ Guitar guitar
+ int quantity
+ getDescription()
+ cost()
Cable
+ Guitar guitar
+ getDescription()
+ cost()
IzmirFender
+ cost()
IzmirGibson
+ cost()
IzmirJackson
+ cost()
IzmirESP
+ cost()
IstanbulFender
+ cost()
IstanbulGibson
+ cost()
IstabulJackson
+ cost()
IstanbulESP
+ cost()
GuitarStore
# chooseGuitar(String type)
IzmirGuitarStore
# chooseGuitar(String type)
+ getName()
IstanbulGuitarStore
# chooseGuitar(String type)
+ getName()

Thank you for listening!