CAT 1

Alejandro Pérez Bueno

Apr 02, 2024

Table of Contents

Question 1

a)

Here are the violated design principles:

- Single Responsibility Principle (SRP): The `Cart` class is responsible for both managing shopping cart items and for calculating the total cost of the cart. This violates SRP as the class changes to how items are managed and changes to how pricing is calculated.
- Open/Closed Principle (OCP): The `totalCost()` method in the `Cart` class won't work well if new product types are added or if the discount logic changes. If, for instance, the discount percentage for books were to change, the `totalCost()` method would need to be modified, which violates OCP.
- Liskov Substitution Principle (LSP): The `Product` class does not have a method for calculating the price which means subclasses (`Egg`, `Bread` and `Book`) would not be able to override such a method for their specific pricing logic.
- Dependency Inversion Principle (DIP): The `Cart` class is directly dependent on concrete classes (Book, Egg, Bread) rather than abstractions. This makes the system less flexible to changes.

The disadvantages of modeling with these violations are the following:

- Lack of flexibility: The system is less adaptable to changes.
- Increased risk of bugs: Changes in one part of the system could affect unrelated parts of the system (in this case cart item management, for example).
- High maintenance costs: The system becomes harder to understand and maintain.
- Poor reusability: The tightly coupled components make it hard to reuse them in different contexts or systems.

b)

Class Diagram

cls Cart

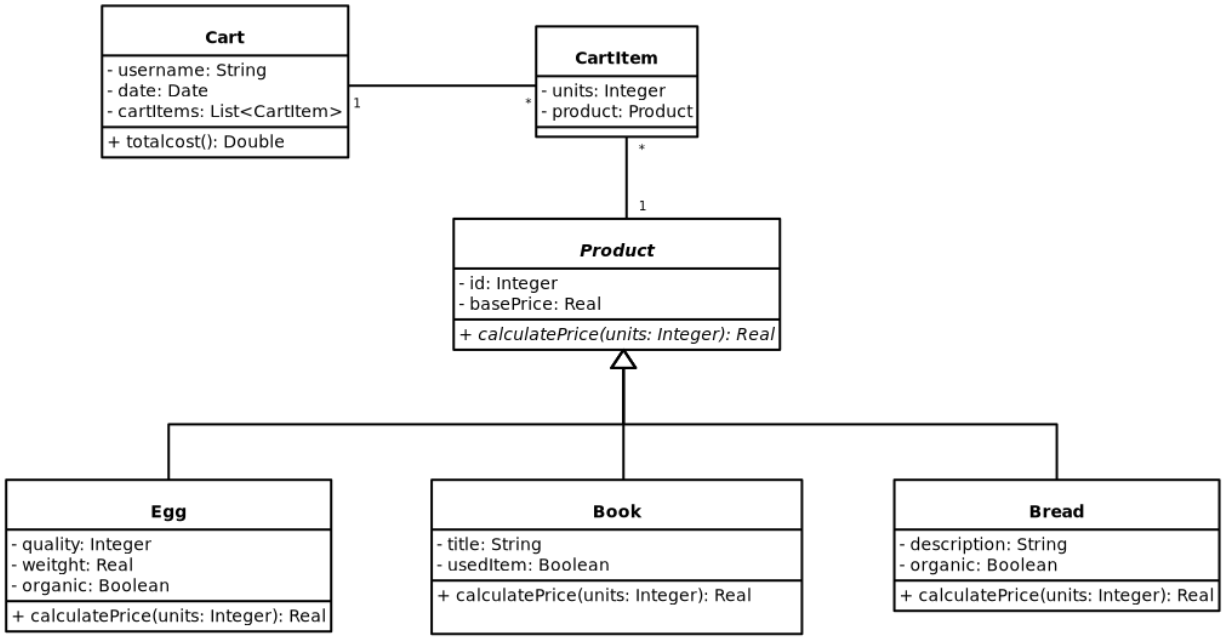


Figure 1: Class Diagram

Pseudocode

```
public abstract class Product {
    protected Integer id;
    protected Real basePrice;

    public abstract Real calculatePrice(Integer units);
}
public class Book extends Product {
    private String title;
    private Boolean usedItem;
```

```
    @Override
    public Real calculatePrice(Integer units) {
        Real price = basePrice * units;
        if (!usedItem) {
            price *= 0.9; // Apply 10% discount for new books
        }
        return price;
    }
}
public class Egg extends Product {
    private Integer quality;
    private Real weight;
    private Boolean organic;

    @Override
    public Real calculatePrice(Integer units) {
        return basePrice * units;
    }
}
public class Bread extends Product {
    private String description;
    private Boolean organic;

    @Override
    public Real calculatePrice(Integer units) {
        return basePrice * units;
    }
}
public class Cart {
    private String username;
    private Date date;
    private List<CartItem> cartItems;

    public Real totalCost() {
        Real total = 0.0;
        for (CartItem cartItem : cartItems) {
            total += cartItem.product.calculatePrice(cartItem.units);
        }
        return total;
    }
```

```
}
public class CartItem {
    private Product product;
    private Integer units;
}
```

Question 2

The Interface Segregation Principle (ISP) states that a class should be forced to depend on methods it does not use. This is often enforced in order to avoid or reduce the side effects caused by changes in the code by splitting the software into smaller interfaces.

Here is a very self-evident example of a violation of the ISP principle:

Pseudocode

```
interface IAnimalSounds {
    void fly();
    void bark();
    void meow();
    void chirp();
}
class Dog implements IAnimalSounds {
    public void bark()
    // Dogs don't meow or chirp but still need to implement the methods.
    public void meow()
    public void chirp()
    // Furthermore dogs can't fly and yet again have to implement this method.
    public void fly()
}
class Cat implements IAnimalSounds {
    public void meow()
    // Cats don't bark or chirp but still need to implement the methods.
    public void bark()
    public void chirp()
    // Furthermore cats can't fly and yet again they must implement this method.
    public void fly()
}
class Bird implements IAnimalSounds {
    public void fly()
    public void chirp()
```

```
    // Birds don't bark or meow but still need to implement the methods.
    public void bark()
    public void meow()
}
```

Class Diagram

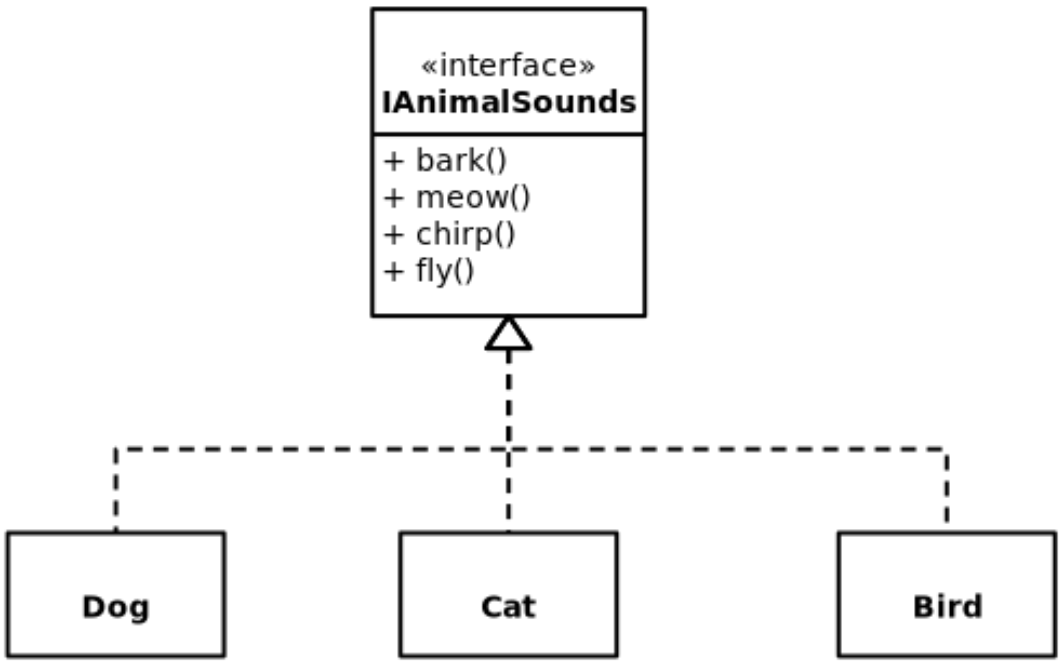


Figure 2: IAnimalSounds

In this example, the `IAnimalSounds` interface violates the ISP because it forces the Dog, Cat, and Bird classes to implement methods that they do not use, which can lead to empty implementations or exceptions.

The violation occurs because the interface is not client-specific, and has certain methods that not all clients need. Thus, the interface should be split into smaller interfaces to suit the needs of more specific clients, without having any client classes have to implement irrelevant methods.

Here is a brief list of some inconveniences of violating the ISP principle:

- Increased Complexity: Animals have to implement methods that they don't need, leading to bloated code.

- Difficulty in Maintainability: Changes to the interface may affect animals that don't use the changed methods, increasing the risk of bugs.
- Difficulty in Understanding: New developers may find it hard to understand why certain methods are implemented but not used.
- Inflexibility: It becomes harder to make changes or refactor the system since many classes are unnecessarily dependent on large interfaces.

Finally, here is a more suitable pseudocode that circumvents the ISP violation:

```
interface ICanFly {
    void fly();
}
interface ICanBark {
    void bark();
}
interface ICanMeow {
    void meow();
}
interface ICanChirp {
    void chirp();
}
class Dog implements ICanBark {
    public void bark()
}
class Cat implements ICanMeow {
    public void meow()
}
class Bird implements ICanChirp, ICanFly {
    public void chirp()
    public void fly()
}
```

Question 3

a)

Here are the analysis patterns we should take into account to design a system for tracking the mobility of employees within the Summer S.A. company:

- **Analysis Patterns**: This type of pattern will be crucial in representing the real-world concepts of employee mobility, location tracking, and time periods within the static analysis diagram. They will help in modeling the relationships and interactions between different entities such as employees, hotels,

and assignments.

- **Architectural Patterns**: Given the dynamic nature of employee assignments and the need for a system that can handle changes efficiently, an architectural pattern could be beneficial. This would allow for separation of concerns, where the presentation layer (user interface), the business logic layer (handling of assignments and tracking) and the data access layer (database operations) are clearly defined.
- **Design Patterns**: Specific design patterns might be used to solve particular problems that arise during the design phase.For example, this kind of pattern could be used to keep the system updated with the latest employee locations, or could manage the state transitions of an employee from one hotel to another.

b)

Classes

- **Employee**:An entity that represents a worker within the company.
- **Hotel**: Represents a hotel within the Summer S.A. network.
- **Assignment**: Relationship between an Employee and a Hotel for a given period of time.

Relationships

- An Employee is assigned to one or more Hotels through an Assignment.
- A Hotel can have multiple Employees assigned at any given time.
- An Assignment is associated with one Employee and one Hotel.

Class Diagram

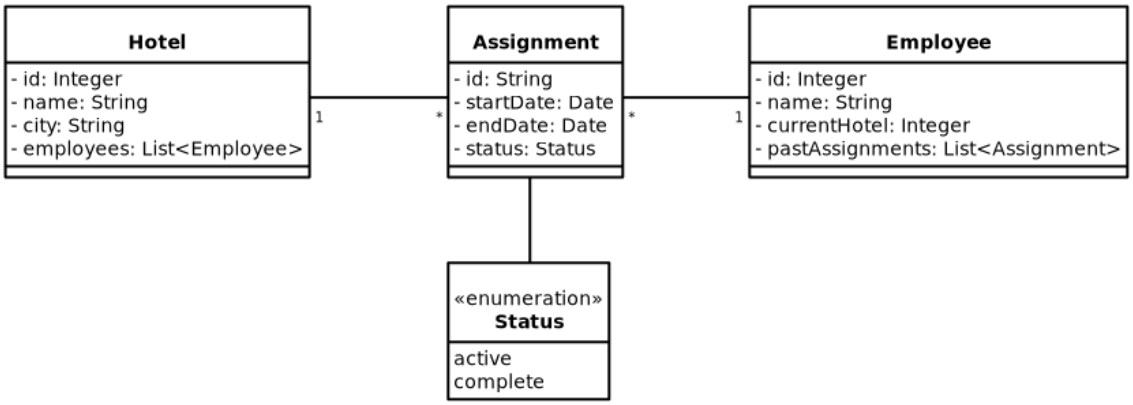


Figure 3: Hotel

Question 4

a)

Here are the analysis patterns we should take into account to design a system for storing information from different types of thermometers:

- **Analysis Patterns**: Again, these patterns may be necessary to represent the real-world concepts involved in the system. For instance, patterns related to the thermometer's characteristics like its name, description, measuring scale, minimum and maximum values it can measure, etc.
- **Architectural patterns**: An architectural pattern could be useful in this case. This is because the software system might be designed to have different layers such as the user interface layer (to enter thermometer details), the business logic layer (to process the entered data), and the data storage layer (to store the thermometer details).
- **Responsibility assignment patterns**: These patterns can help assign responsibilities to different classes. For instance, the Thermometer class might be responsible for storing and providing information about each thermometer.

b)

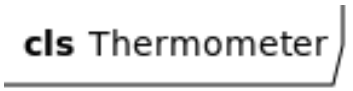
Classes and Relationships

- `ThermometerType`: represents different types of thermometers, each with a name, description, and associated measuring scale.
  - Attributes:
    - `name`: String
    - `description`: String
    - `measuringScale`: Scale
  - Relationships:
    - Has a `Scale` (association)
    - Has multiple `TemperatureRange` (composition)
- `Scale`: represents the scale on which the thermometer measures temperature (Celsius or Fahrenheit).
  - Attributes:
    - `scaleType`: String (e.g., Celsius, Fahrenheit)
  - Relationships:
    - Is part of `ThermometerType` (association)
- `TemperatureRange`: represents the range of temperatures that a thermometer can measure, with a minimum and maximum value.
  - Attributes:
    - `minValue`: Float
    - `maxValue`: Float
  - Relationships:
    - Is part of `ThermometerType` (composition)

Integrity Constraints:

- The `minValue` should always be less than the `maxValue` in `TemperatureRange`.
- The `scaleType` in `Scale` should be limited to "Celsius" or "Fahrenheit".

Class Diagram



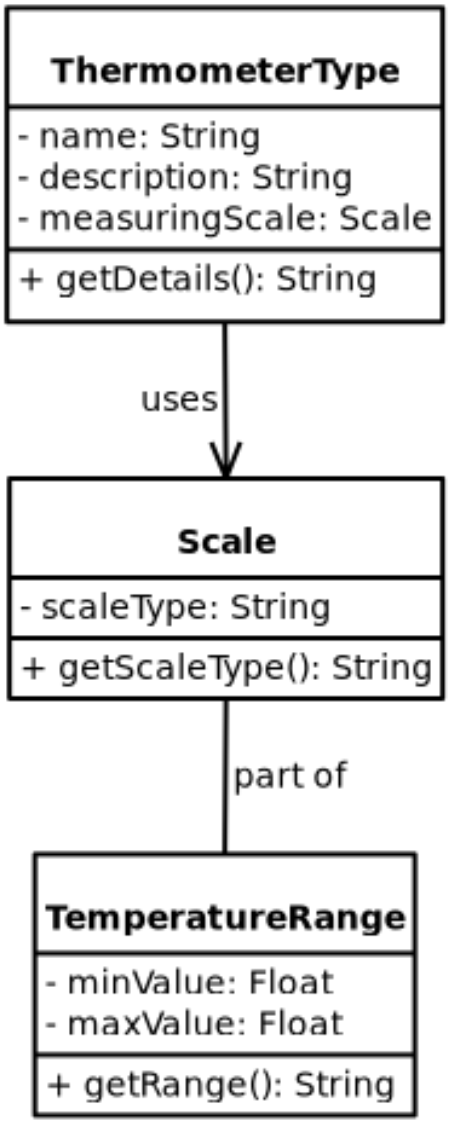


Figure 4: Thermometer