VNUHCM - UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

HOMEWORK REPORT
COURSE: OBJECT-ORIENTED PROGRAMMING

WEEK 09: ASSIGNMENT 02
SOLID PRINCIPLES

Instructor:
Nguyen Le Hoang Dung
Ho Tuan Thanh

Student:
Le Trung Kien 23127075 23CLC08

Ho Chi Minh City, 2024

Contents

List of Figures

1 SOLID PRINCIPLES

SOLID Principles are the five principles that help developers enhance their Object-oriented class designs, which are:

- Single Responsibility Principle (SRP)
- Open/Closed Principle
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

2 Liskov Substitution Principle (LSP)

2.1 Concept Explanation

2.1.1 Definition

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes without any unexpected behavior.

2.1.2 Importance of LSP in object-oriented design

LSP is essential in object-oriented design for multiple reasons:

- Code extensibility enhancement: Allow developers to extend their program by creating new derived classes without breaking the existing base classes' code.
- Code consistency assurance: Maintaining the substitutability of derived classes for base classes ensures the consistency of code's behavior.
- Polymorphism facilitation: Handle various derived classes uniformly, promoting flexibility and reusability.
- Bugs reduction: Ensuring that subclasses seamlessly substitute their base classes helps prevent unexpected issues.

2.1.3 Diagram

Let's get shapes rectangle and square as the examples. A square can extend from a rectangle as its size is the rectangle's width and height with the same value. However, in case we need to modify both the width and height of a rectangle, which can be different from each other, if we use a square (derived class) instead of a rectangle (base class), an issue occurs as we don't know to change the size of the square to which value: height or width?

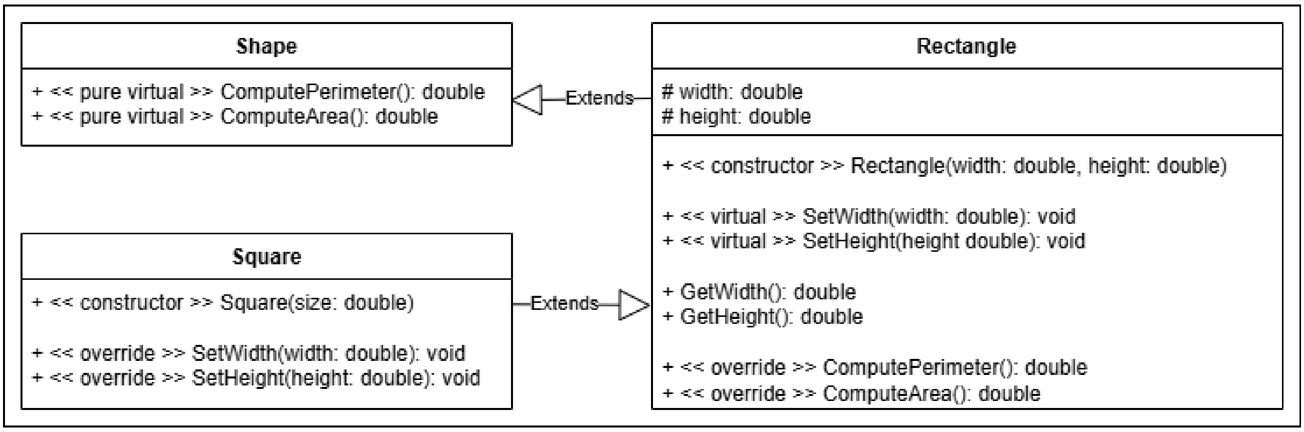
Shape
+ << pure virtual >> ComputePerimeter(): double
+ << pure virtual >> ComputeArea(): double
Extends
Rectangle
# width: double
# height: double
+ << constructor >> Rectangle(width: double, height: double)
+ << virtual >> SetWidth(width: double): void
+ << virtual >> SetHeight(height double): void
+ GetWidth(): double
+ GetHeight(): double
+ << override >> ComputePerimeter(): double
+ << override >> ComputeArea(): double
Square
+ << constructor >> Square(size: double)
+ << override >> SetWidth(width: double): void
+ << override >> SetHeight(height: double): void
Extends

Figure 1: Violate LSP Class Diagram

To tackle this, in order to follow LSP, we should divide those shapes into two isolated derived classes.

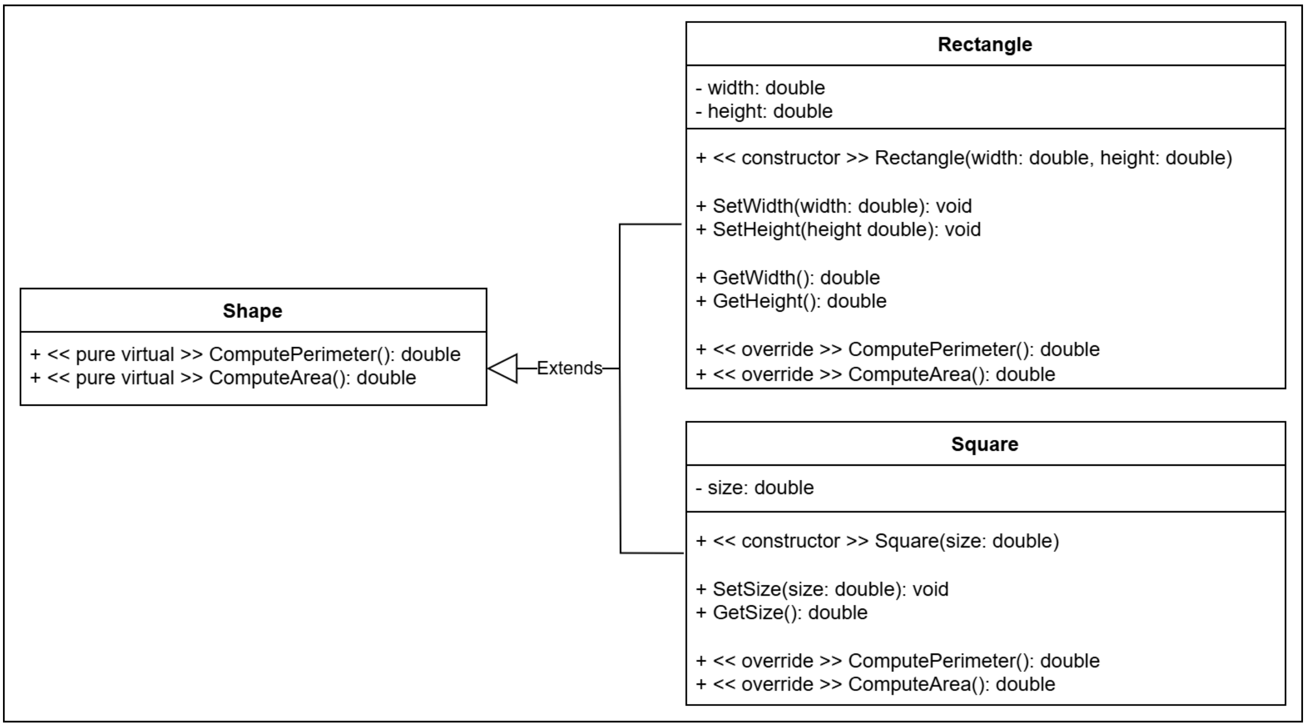
Rectangle
- width: double
- height: double
+ << constructor >> Rectangle(width: double, height: double)
+ SetWidth(width: double): void
+ SetHeight(height double): void
+ GetWidth(): double
+ GetHeight(): double
+ << override >> ComputePerimeter(): double
+ << override >> ComputeArea(): double
Shape
+ << pure virtual >> ComputePerimeter(): double
+ << pure virtual >> ComputeArea(): double
Extends
Square
- size: double
+ << constructor >> Square(size: double)
+ SetSize(size: double): void
+ GetSize(): double
+ << override >> ComputePerimeter(): double
+ << override >> ComputeArea(): double

Figure 2: Follow LSP Class Diagram

2.2 Application in Assignments or Projects

In assignment 03: Prince and Princess of week 07: Polymorphism, my code follows the Liskov Substitution Principle as the derived classes: BusinessGate, AcademicGate, and PowerGate, can be completely used in place of the base class: Gate with three pure virtual functions.

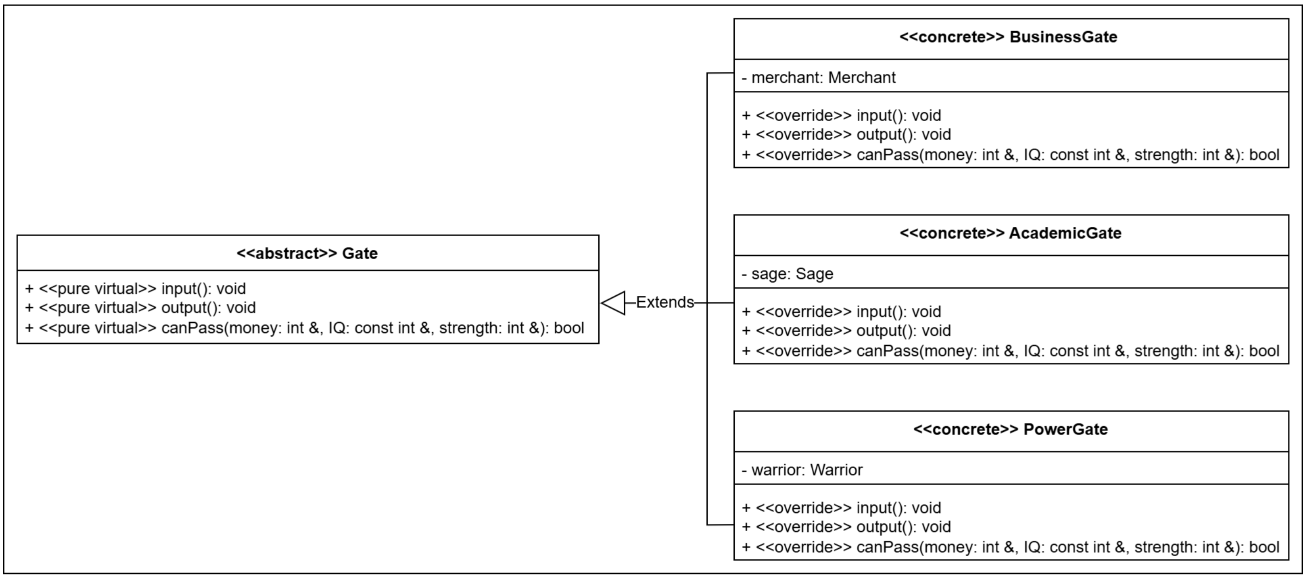
<<concrete>> BusinessGate
- merchant: Merchant
+ <<override>> input(): void
+ <<override>> output(): void
+ <<override>> canPass(money: int &, IQ: const int &, strength: int &): bool
<<concrete>> AcademicGate
- sage: Sage
+ <<override>> input(): void
+ <<override>> output(): void
+ <<override>> canPass(money: int &, IQ: const int &, strength: int &): bool
<<abstract>> Gate
+ <<pure virtual>> input(): void
+ <<pure virtual>> output(): void
+ <<pure virtual>> canPass(money: int &, IQ: const int &, strength: int &): bool
Extends
<<concrete>> PowerGate
- warrior: Warrior
+ <<override>> input(): void
+ <<override>> output(): void
+ <<override>> canPass(money: int &, IQ: const int &, strength: int &): bool

Figure 3: Gate diagrams

It provided me with several benefits:

- Code extensibility and consistency: I can extend my system by adding new Gate types easily.
- Polymorphism facilitation: I can input and output the data of the gates and check if the prince can pass the gates or note without knowing the gates' types.
- Bugs reduction: Three gate types are separated from each other so it is quite simple to modify the code without creating bugs.

I assigned design pattern **Strategy** to the mentioned assignment:

2.3 Connection to the Seminar

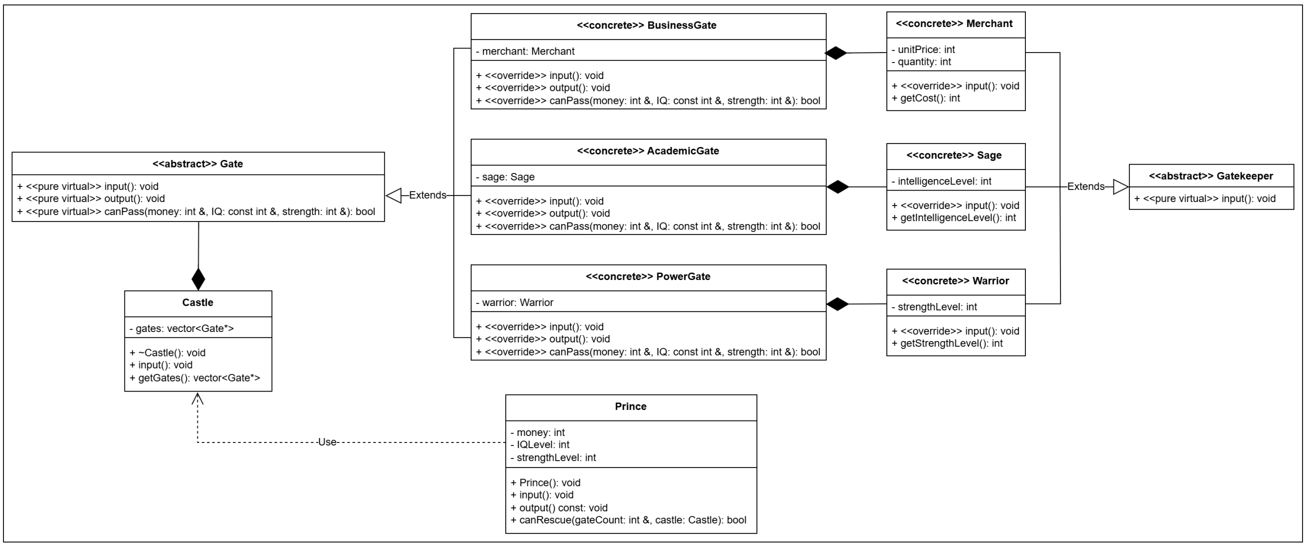
<<concrete>> BusinessGate
- merchant: Merchant
+ <<override>> input(): void
+ <<override>> output(): void
+ <<override>> canPass(money: int &, IQ: const int &, strength: int &): bool
<<concrete>> Merchant
- unitPrice: int
- quantity: int
+ <<override>> input(): void
+ getCost(): int
<<abstract>> Gate
+ <<pure virtual>> input(): void
+ <<pure virtual>> output(): void
+ <<pure virtual>> canPass(money: int &, IQ: const int &, strength: int &): bool
Extends
<<concrete>> AcademicGate
- sage: Sage
+ <<override>> input(): void
+ <<override>> output(): void
+ <<override>> canPass(money: int &, IQ: const int &, strength: int &): bool
<<concrete>> Sage
- intelligenceLevel: int
+ <<override>> input(): void
+ getIntelligenceLevel(): int
Extends
<<abstract>> Gatekeeper
+ <<pure virtual>> input(): void
Castle
- gates: vector<Gate*>
+ ~Castle(): void
+ input(): void
+ getGates(): vector<Gate*>
<<concrete>> PowerGate
- warrior: Warrior
+ <<override>> input(): void
+ <<override>> output(): void
+ <<override>> canPass(money: int &, IQ: const int &, strength: int &): bool
<<concrete>> Warrior
- strengthLevel: int
+ <<override>> input(): void
+ getStrengthLevel(): int
Use
Prince
- money: int
- IQLevel: int
- strengthLevel: int
+ Prince(): void
+ input(): void
+ output() const: void
+ canRescue(gateCount: int &, castle: Castle): bool

Figure 4: Prince and Princess assignment's class diagram

Strategy pattern's components:

- Context class: Castle.
- Interface (base class): Gate.
- Concrete classes: BusinessGate, AcademicGate, and PowerGate.