BSc (Hons) in Information Technology
Year 2

Lab 06 – Design Patterns

IT2020 - Software Engineering Semester 1, 2025

In this Lab, we will look into the Strategy pattern under Design Patterns.

You will use the NetBeans IDE.

Strategy Pattern

In software development, we often face situations where an object needs to perform a task, but the way it performs that task may change depending on circumstances. Instead of hard-coding all the possible ways inside one class, the **Strategy Pattern** allows us to define a *family of algorithms* (or behaviors), put each into its own class, and make them interchangeable at runtime.

What It Does

The Strategy Pattern enables you to:

- Define a set of related algorithms or behaviors (called **strategies**).
- Encapsulate each behavior in a separate class.
- Let the main object (called the **context**) use these behaviors without knowing their details.
- Change the behavior dynamically while the program is running.

When to Use

Use the Strategy Pattern when:

- You have multiple ways to perform a task (e.g., payment by card, PayPal, or cash).
- You want to avoid long *if-else* or *switch* statements to choose which algorithm to run.
- You expect the behavior to change at runtime (e.g., switching sorting algorithms, discount methods, or user roles).

Structure

1. **Strategy Interface** – Defines the common behavior (e.g., TravelStrategy with travel() method).

2. **Concrete Strategies** – Different implementations (e.g., CarStrategy, TrainStrategy, FlightStrategy).

3. **Context Class** – Holds a reference to a Strategy and delegates the work to it.

Practice Activity

The university is developing a Student Activity Management System to manage both academic programs and competition participation for students, supporting different student types such as Undergraduates and Postgraduates. Each student can enroll in an academic program like BSc, MSc, or Doctoral, and at the same time take part in extra-curricular activities such as CodeFest, RoboFest, or GameFest.

The system should add points for extra-curricular activities
In CodeFest, points are awarded as follows:
100 points for 1st place,
70 points for 2nd
50 points for 3rd
15 points for participation

In RoboFest, points are awarded as follows:
110 points for 1st place
80 points for 2nd place
60 points for 3rd place
20 for participation

In GameFest, points are awarded as follows:
90 points for 1st place
65 points for 2ndplace
45 points for 3rd place
25 for participation

Each festival awards points differently for 1st, 2nd, 3rd places and participation. Students can switch both their academic program and festival dynamically at runtime.

Your Task:
Implement the above scenario using the **Strategy Pattern,** following the step-by-step guide already provided in the lab sheet:

- Define interfaces for program and competition strategies.
- Implement concrete classes for BSc, MSc, Doctoral, CodeFest, RoboFest, and GameFest.
- Create Student as a context class using composition.
- Demonstrate switching strategies (program/festival) dynamically.

Follow the steps below

Step 1
First, you should identify what the university wants to achieve. The system must handle two types of students (Undergraduates and Postgraduates), support academic programs (BSc, MSc, Doctoral), and manage competitions (CodeFest, RoboFest, GameFest) where points are awarded based on performance. Additionally, students should be able to switch their program and festival while the system is running, and their total points should be tracked.

Step 2
Next, determine what parts of the system may vary and need flexibility. In this case, academic programs can change (BSc → MSc → Doctoral) and competition rules for awarding points differ by festival. Since both of these behaviors can change independently at runtime, they are good candidates for the Strategy Pattern, where we separate the "what varies" from the rest of the student logic.

Step 3
Now, define common contracts for these behaviors. You create one interface for Program Strategy (to represent academic program enrollment) and one for Competition Strategy (to represent competition scoring). Each interface should define the key behavior (e.g., getting the program name, or calculating points for a given rank). This ensures that all future programs and competition can follow the same structure.

Step 4
Once the interfaces are defined, you build the actual behaviors. For academic programs, create strategies like BScProgram, MScProgram, and DoctoralProgram. For festivals,

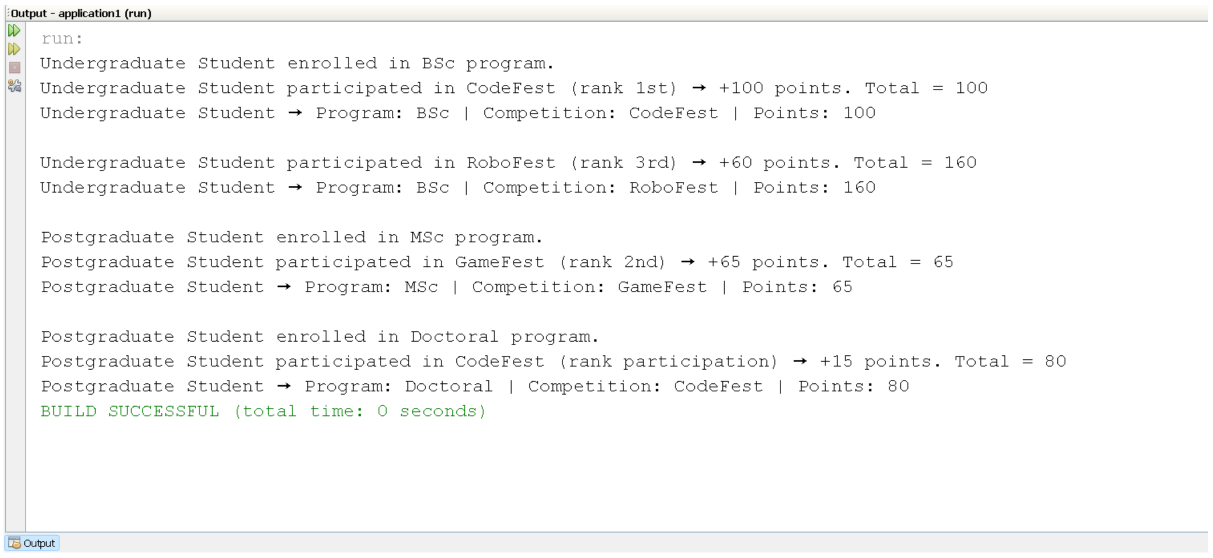
implement CodeFest, RoboFest, and GameFest, each with its specific point-awarding rules. These concrete classes encapsulate the unique logic, but they all adhere to their strategy interface.

Step 5
Create an abstract Student class that acts as the "context" in the Strategy Pattern. This class should have references to both a program strategy and a festival strategy. Instead of implementing enrollment and scoring logic directly, it delegates to the strategies. The Student class also keeps track of the total points earned. It provides methods for enrolling, participating in a competition, and showing the current state.

Step 6
Then, add concrete student types by extending the abstract Student class. Define UndergraduateStudent and PostGraduateStudent, which represent the two categories. At this stage, they may simply call the parent class constructor, but having separate types makes the system extensible if you need different rules for undergraduates and postgraduates in the future.

Step 7
Because the design uses composition with strategies, you can give Student setter methods like setProgram and setFestival. This allows you to dynamically change the student's academic program or festival without altering the Student class or restarting the system. A student may switch from BSc to MSc or from CodeFest to RoboFest, and the system immediately applies the new behavior.

Step 8
Finally, in the application's main flow, demonstrate the process. Create students, assign them initial programs and festivals, and simulate participation in competitions. Show that points are awarded correctly and that switching strategies at runtime works as expected. For example, an undergraduate can join CodeFest and earn 100 points for first place, then later switch to RoboFest and earn 60 more points for third place. A postgraduate can enroll in MSc, compete in GameFest, and later upgrade to a Doctoral while switching festivals.



```
run:
Undergraduate Student enrolled in BSc program.
Undergraduate Student participated in CodeFest (rank 1st) → +100 points. Total = 100
Undergraduate Student → Program: BSc | Competition: CodeFest | Points: 100

Undergraduate Student participated in RoboFest (rank 3rd) → +60 points. Total = 160
Undergraduate Student → Program: BSc | Competition: RoboFest | Points: 160

Postgraduate Student enrolled in MSc program.
Postgraduate Student participated in GameFest (rank 2nd) → +65 points. Total = 65
Postgraduate Student → Program: MSc | Competition: GameFest | Points: 65

Postgraduate Student enrolled in Doctoral program.
Postgraduate Student participated in CodeFest (rank participation) → +15 points. Total = 80
Postgraduate Student → Program: Doctoral | Competition: CodeFest | Points: 80
BUILD SUCCESSFUL (total time: 0 seconds)
```

Group Project Task and SubmissionGroup Project Task

1. Form your pre-assigned 6-member groups.
2. Continue using the same project topic assigned in your previous lab.
3. Incorporate appropriate design patterns within your project implementation.
4. Each team must apply at least one design pattern that best fits your project requirements.

Submission Checklist

1. Report

Prepare a well-structured and clearly formatted report including the following sections:

- Group Details
  (List all members with student IDs)
- Design Pattern(s) Used
  (Mention which design pattern(s) your team applied)
- Justification for Each Pattern
  (Explain why the selected design pattern(s) are suitable for your project and how they improve your design)

- Screenshots of Implementation
  (Include relevant code snippets or output screenshots that demonstrate the applied design pattern(s))

2. Code Files

Submit your source code with a clean and organized structure, ensuring the following:

- Proper use of Java classes (e.g., *Main Class, Context Class, Concrete Classes, Interfaces*, etc.)
- Clear naming conventions and comments for better readability
- Code must be functional and demonstrate the use of the identified design pattern(s)