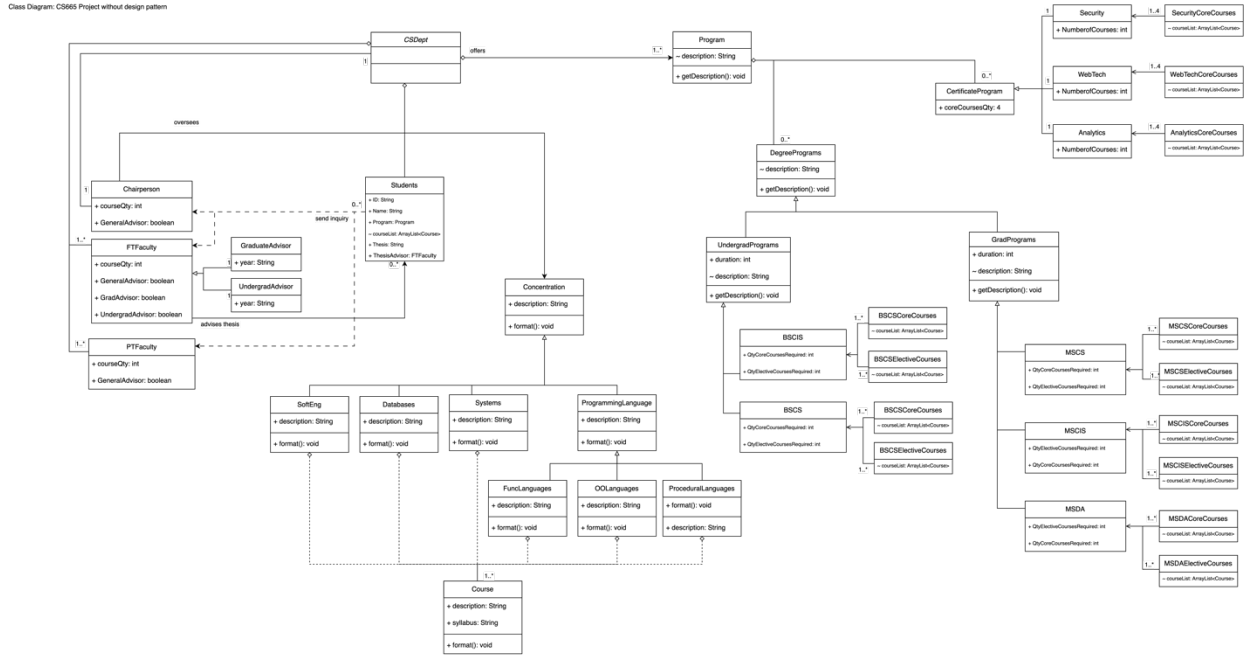
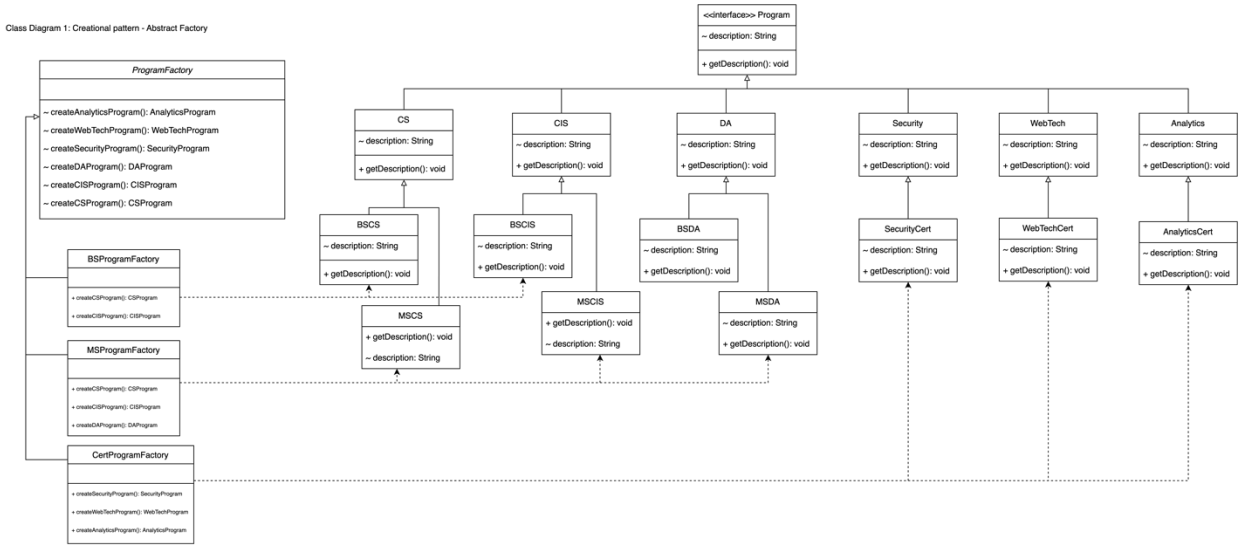
CS665_Project_LongWeiNee

Class Diagram: CS665 Project without design pattern



Creational Pattern 1: Abstract Family Pattern



Scenario:
The CS Department offers three different programs:

- Undergraduate program (BS) in CS, CIS
- Graduate program (MS) in CS, CIS, DA
- Certification program: Security, Web Technology, Analytics

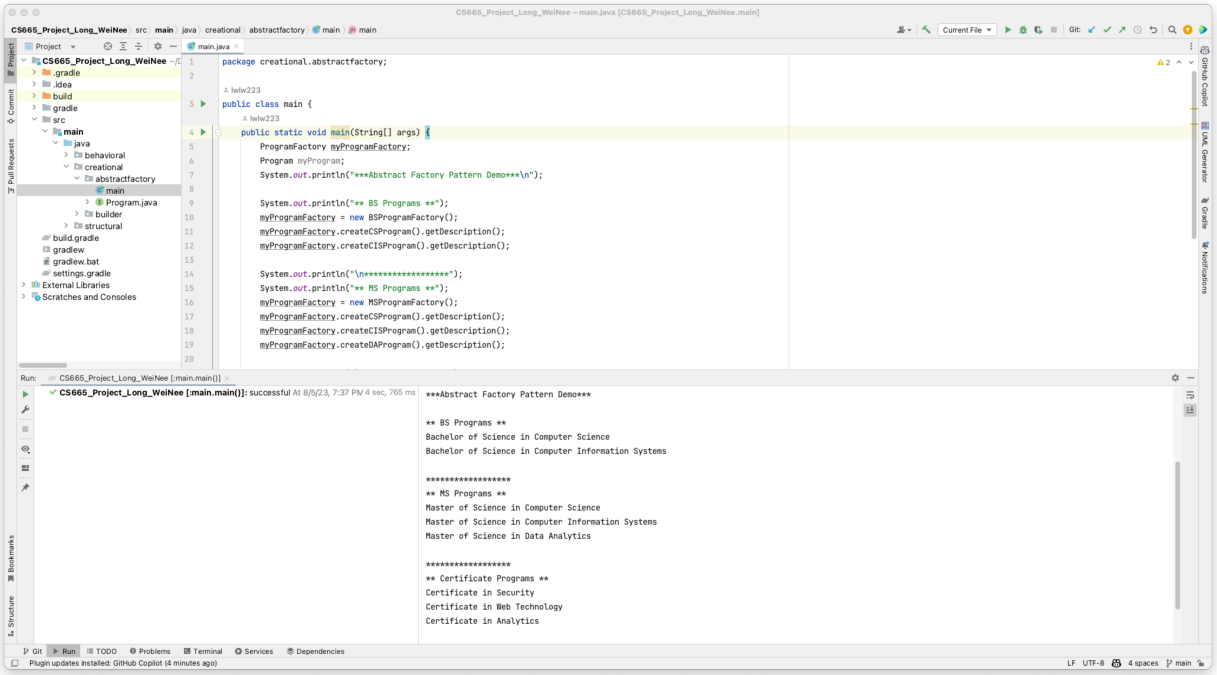
Each major has a fixed number of core and elective classes.

Justification:
The Abstract Family Pattern provides a way to create families of related objects without specifying their concrete classes. In this scenario, there are 3 different families of programs: “BSDegreeProgram”, “MSDegreeProgram”, and “CertificationProgram”.

Using this pattern, we can define an interface for the Program, which has methods to create objects for core and elective classes. Then, we can create concrete classes that implement this interface. Each concrete class is responsible for creating the appropriate core and elective classes for its respective program.

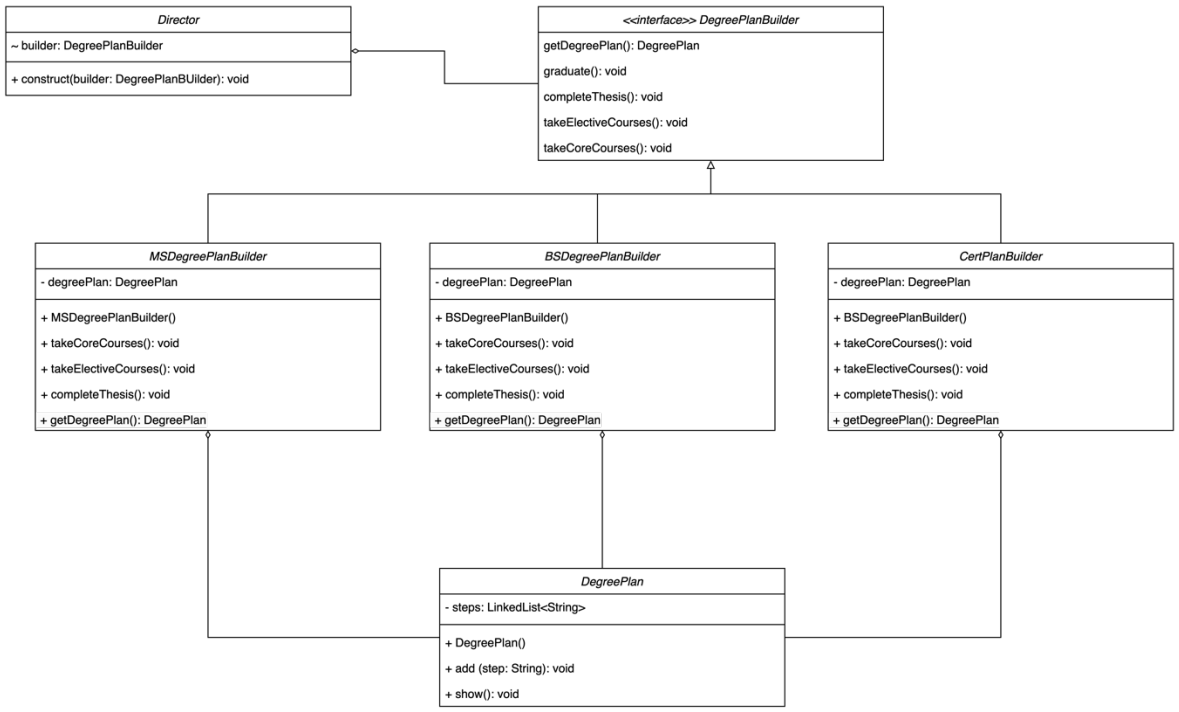
This pattern provides a way to organize the creation of families of related objects in a flexible and consistent manner, promoting code reusability. For example, if we need to add a new degree program in the future (e.g., Ph.D. program), we will only need to create a new concrete class implementing the ProgramFactory interface, without affecting the client code.

Output:


CS665_Project_Long_WeiNee – main.java [CS665_Project_Long_WeiNee.main]
CS665_Project_Long_WeiNee / src / main / java / creational / abstractfactory / main / main
Current File
Git:
Project
main.java
CS665_Project_Long_WeiNee ~/
.gradle
.idea
build
gradle
src
main
java
behavioral
creational
abstractfactory
main
Program.java
builder
structural
build.gradle
gradlew
gradlew.bat
settings.gradle
External Libraries
Scratches and Consoles
1 package creational.abstractfactory;
2
lwlw223
3 public class main {
lwlw223
4 public static void main(String[] args) {
5 ProgramFactory myProgramFactory;
6 Program myProgram;
7 System.out.println("***Abstract Factory Pattern Demo***\n");
8
9 System.out.println("** BS Programs **");
10 myProgramFactory = new BSProgramFactory();
11 myProgramFactory.createCSProgram().getDescription();
12 myProgramFactory.createCISProgram().getDescription();
13
14 System.out.println("\n*****************");
15 System.out.println("** MS Programs **");
16 myProgramFactory = new MSProgramFactory();
17 myProgramFactory.createCSProgram().getDescription();
18 myProgramFactory.createCISProgram().getDescription();
19 myProgramFactory.createDAProgram().getDescription();
20
Run:
CS665_Project_Long_WeiNee [:main.main()]
CS665_Project_Long_WeiNee [:main.main()]: successful At 8/5/23, 7:37 PM 4 sec, 765 ms
Abstract Factory Pattern Demo
** BS Programs **
Bachelor of Science in Computer Science
Bachelor of Science in Computer Information Systems
****
** MS Programs **
Master of Science in Computer Science
Master of Science in Computer Information Systems
Master of Science in Data Analytics
****
** Certificate Programs **
Certificate in Security
Certificate in Web Technology
Certificate in Analytics
Git
Run
TODO
Problems
Terminal
Services
Dependencies
Plugin updates installed: GitHub Copilot (4 minutes ago)
LF
UTF-8
4 spaces
main

Creational Pattern 2: Builder Pattern

Class Diagram 2: Creational pattern - Builder



Scenario:
For BS and MS degree program, the requirements for graduation are:

- A fixed number of core courses
- A fixed number of electives (in the last year of study)
- Thesis in the last semester

For the certification program, the requirements are:

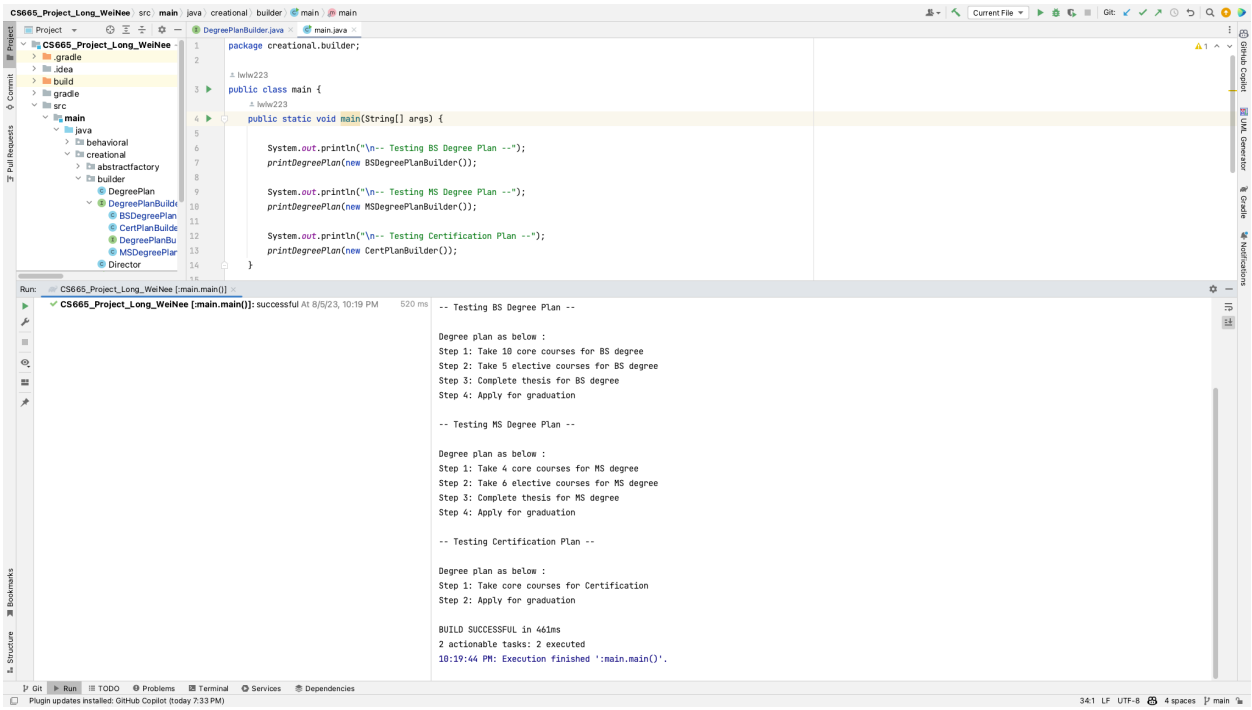
- 4 core courses

Justification:
The builder pattern allows us to construct complex objects step by step while keeping the construction process separate from the object's representation.

In this scenario, we have three different types of degree plans (for BS, MS, and Certification).

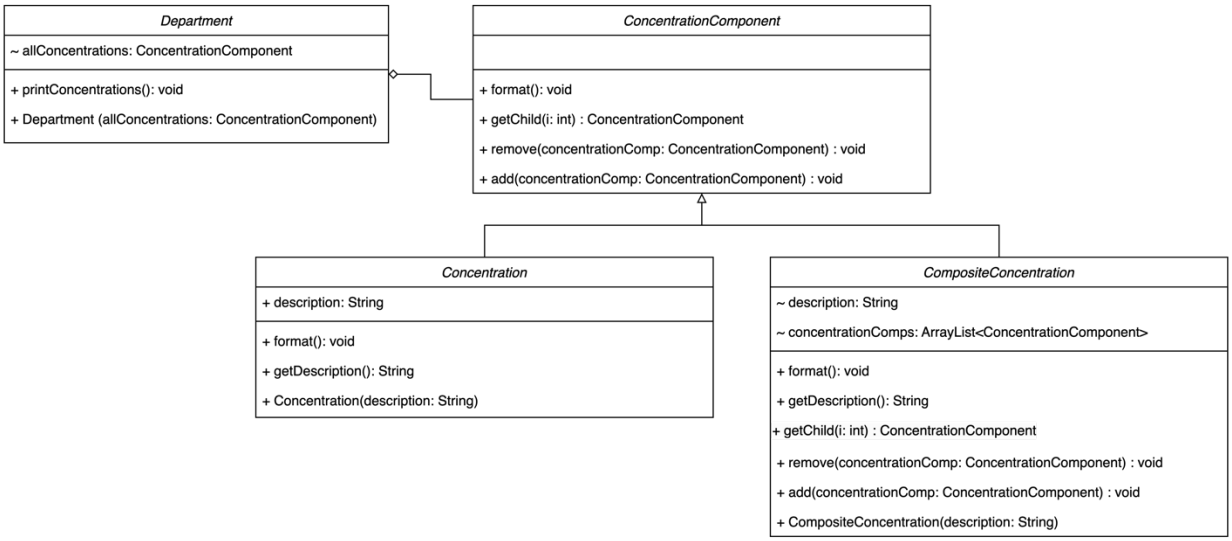
To manage this complexity effectively, we can define separate degree plan builder classes for BS, MS and Cert programs. These builders will handle the construction of the degree plan by adding the appropriate core courses, elective courses, and completing thesis as needed. Each builder will also handle the differentiation between requirements for core classes, electives, and thesis. This allows us to create different degree plans while hiding the implementation details, hence providing a consistent interface to work with the client code.

Output:


CS665_Project_Long_WeiNee
src
main
java
creational
builder
main
Current File
Git:
Project
DegreePlanBuilder.java
main.java
CS665_Project_Long_WeiNee
.gradle
.idea
build
gradle
src
main
java
behavioral
creational
abstractfactory
builder
DegreePlan
DegreePlanBuilde
BSDegreePlan
CertPlanBuilde
DegreePlanBu
MSDegreePlar
Director
package creational.builder;
lwlw223
public class main {
lwlw223
public static void main(String[] args) {
System.out.println("\n-- Testing BS Degree Plan --");
printDegreePlan(new BSDegreePlanBuilder());
System.out.println("\n-- Testing MS Degree Plan --");
printDegreePlan(new MSDegreePlanBuilder());
System.out.println("\n-- Testing Certification Plan --");
printDegreePlan(new CertPlanBuilder());
}
Run:
CS665_Project_Long_WeiNee [:main.main()]
CS665_Project_Long_WeiNee [:main.main()]: successful At 8/5/23, 10:19 PM
520 ms
-- Testing BS Degree Plan --
Degree plan as below :
Step 1: Take 10 core courses for BS degree
Step 2: Take 5 elective courses for BS degree
Step 3: Complete thesis for BS degree
Step 4: Apply for graduation
-- Testing MS Degree Plan --
Degree plan as below :
Step 1: Take 4 core courses for MS degree
Step 2: Take 6 elective courses for MS degree
Step 3: Complete thesis for MS degree
Step 4: Apply for graduation
-- Testing Certification Plan --
Degree plan as below :
Step 1: Take core courses for Certification
Step 2: Apply for graduation
BUILD SUCCESSFUL in 461ms
2 actionable tasks: 2 executed
10:19:44 PM: Execution finished ':main.main()'.
Git
Run
TODO
Problems
Terminal
Services
Dependencies
Plugin updates installed: GitHub Copilot (today 7:33 PM)
34:1
LF
UTF-8
4 spaces
main

Structural Pattern 1: Composite

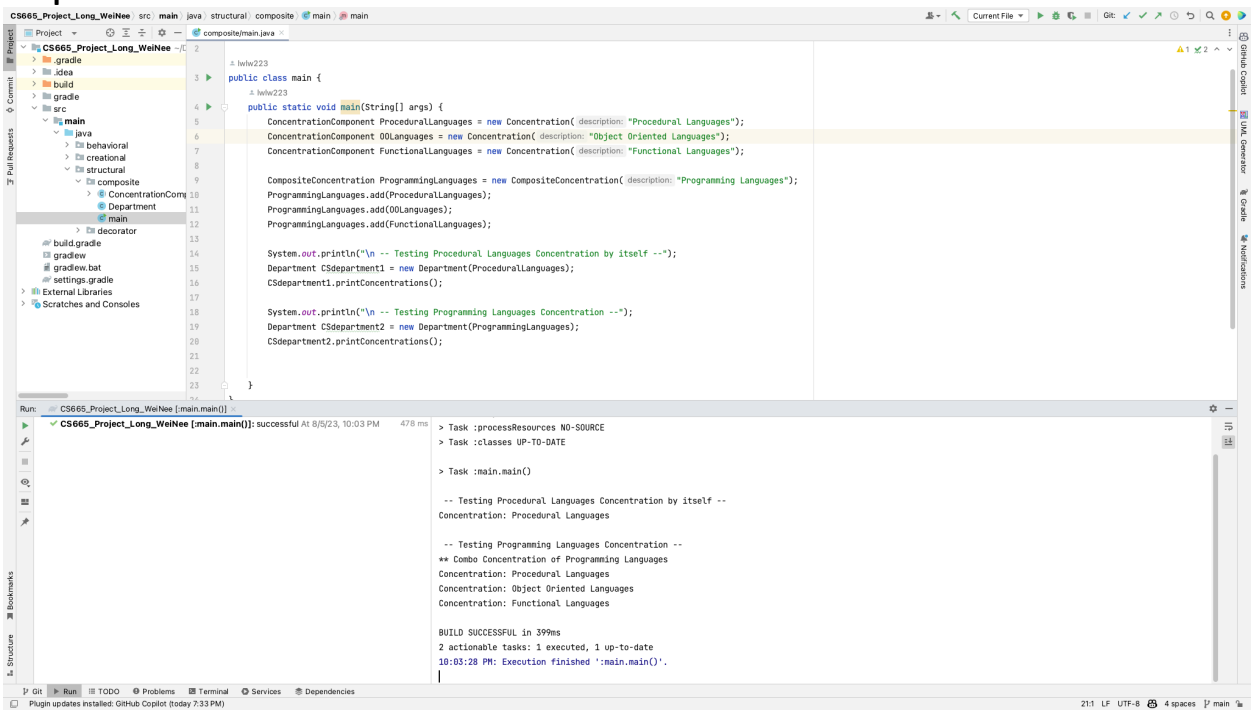
Class Diagram 3: Structural pattern: Composite



Scenario:
Organizing the concentrations in the CS Department. Each concentration can contain a sub-concentration or a list of courses in that concentration.

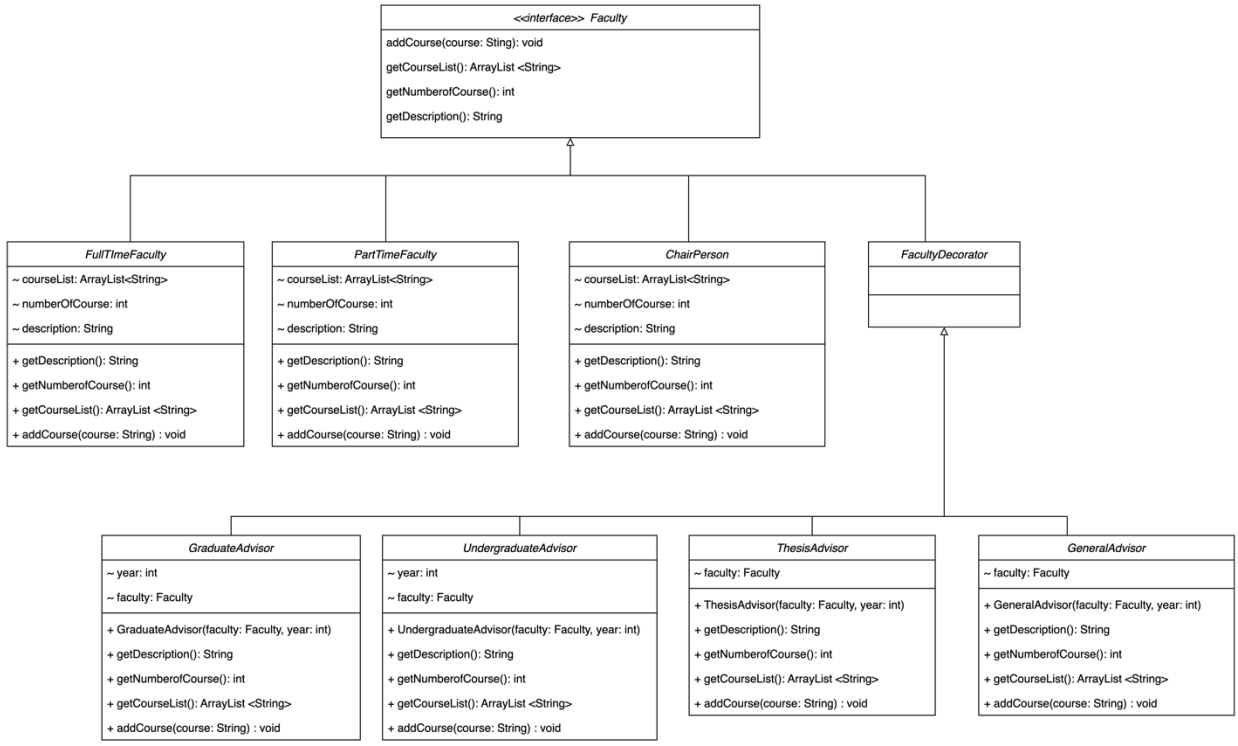
Justification:
The composite pattern is appropriate in the scenario because it allows us to treat each individual concentration and composite concentration uniformly. For example, the Programming Languages concentration has sub-concentrations of Procedural Languages, Object Oriented Languages, and Functional Languages. By using the composite pattern, we can create a common interface that work uniformly with both the top concentration and the low-level concentration.

Output:


CS665_Project_Long_WeiNee
src
main
java
structural
composite
main
main
Current File
Git:
Project
composite/main.java
lwlw223
public class main {
lwlw223
public static void main(String[] args) {
ConcentrationComponent ProceduralLanguages = new Concentration(description: "Procedural Languages");
ConcentrationComponent OOLanguages = new Concentration(description: "Object Oriented Languages");
ConcentrationComponent FunctionalLanguages = new Concentration(description: "Functional Languages");
CompositeConcentration ProgrammingLanguages = new CompositeConcentration(description: "Programming Languages");
ProgrammingLanguages.add(ProceduralLanguages);
ProgrammingLanguages.add(OOLanguages);
ProgrammingLanguages.add(FunctionalLanguages);
System.out.println("\n -- Testing Procedural Languages Concentration by itself --");
Department CSdepartment1 = new Department(ProceduralLanguages);
CSdepartment1.printConcentrations();
System.out.println("\n -- Testing Programming Languages Concentration --");
Department CSdepartment2 = new Department(ProgrammingLanguages);
CSdepartment2.printConcentrations();
}
.gradle
.idea
build
gradle
src
main
java
behavioral
creational
structural
composite
ConcentrationComp
Department
main
decorator
build.gradle
gradlew
gradlew.bat
settings.gradle
External Libraries
Scratches and Consoles
Run:
CS665_Project_Long_WeiNee [:main.main()]
CS665_Project_Long_WeiNee [:main.main()]: successful At 8/5/23, 10:03 PM
478 ms
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :main.main()
-- Testing Procedural Languages Concentration by itself --
Concentration: Procedural Languages
-- Testing Programming Languages Concentration --
** Combo Concentration of Programming Languages
Concentration: Procedural Languages
Concentration: Object Oriented Languages
Concentration: Functional Languages
BUILD SUCCESSFUL in 399ms
2 actionable tasks: 1 executed, 1 up-to-date
10:03:28 PM: Execution finished ':main.main()'.
Git
Run
TODO
Problems
Terminal
Services
Dependencies
Plugin updates installed: GitHub Copilot (today 7:33 PM)
21:1
LF
UTF-8
4 spaces
main

Structural Pattern 2: Decorator

Class Diagram 4: Structural pattern: Decorator

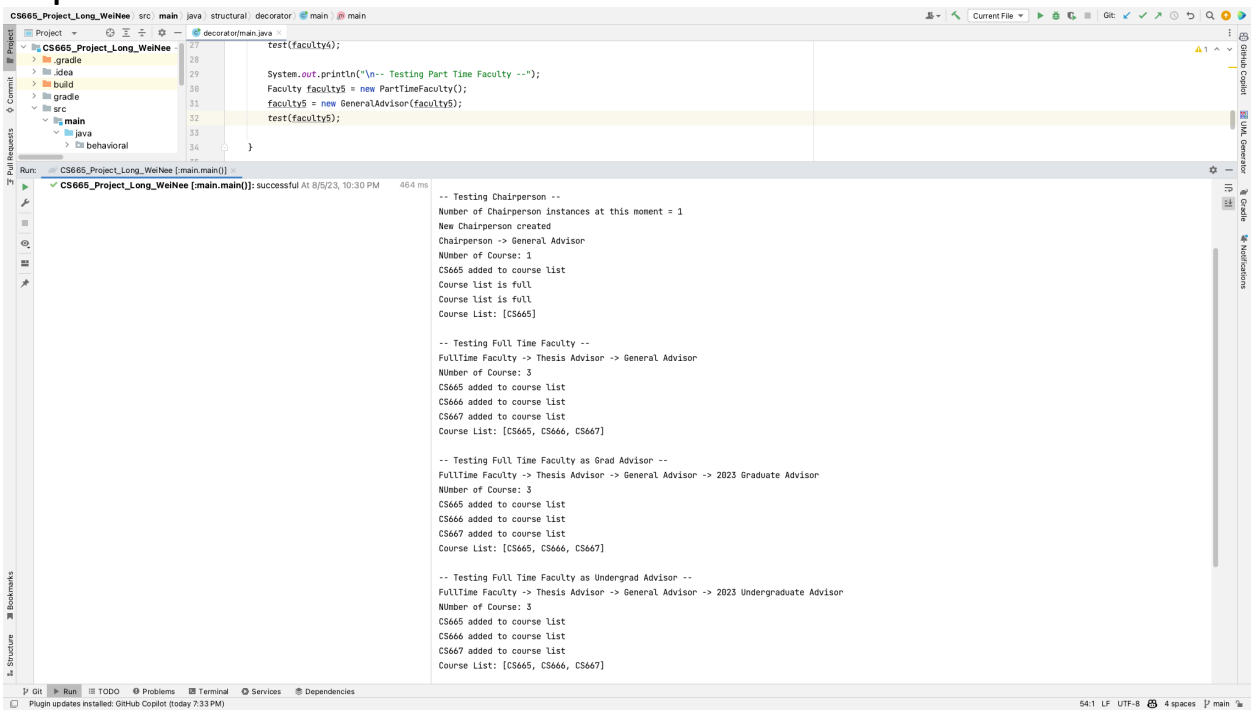


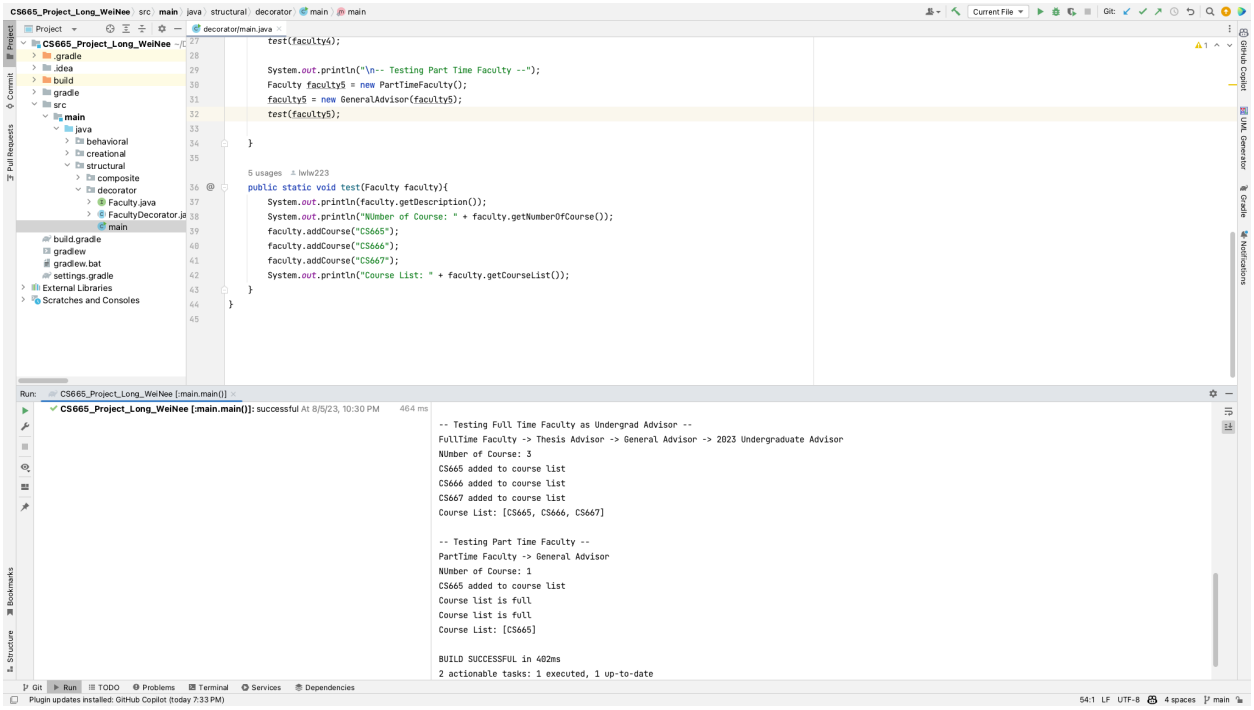
Scenario:
The department has a chairperson, several full-time faculties, and several part-time faculty. Among the full-time faculty, there is one Graduate Advisor and one Undergraduate Advisor that serves in that position for a year. All full-time faculty are also a Thesis Advisor, and all faculty are General Advisors that receive queries from students.

Justification:
The decorator pattern allows us to add new responsibility to an object without affecting the entire class hierarchy. In this scenario, we have different faculty members with varying roles (Graduate advisor, Undergraduate advisor, Thesis advisor, and General advisor). Using the decorator pattern, we can define decorator classes for each specific role, and they will be responsible for adding the specific functionality to an object.

For example, a full-time faculty member can be decorated with GraduateAdvisor decorator and UndergraduateAdvisor decorator to take on the roles of Graduate Advisor and Undergraduate Advisor for a specific year. Similarly, a full-time or part-time faculty member can be decorated with ThesisAdvisor decorator to become a Thesis Advisor, and all faculty members can be decorated with GeneralAdvisor decorator to handle general student queries.

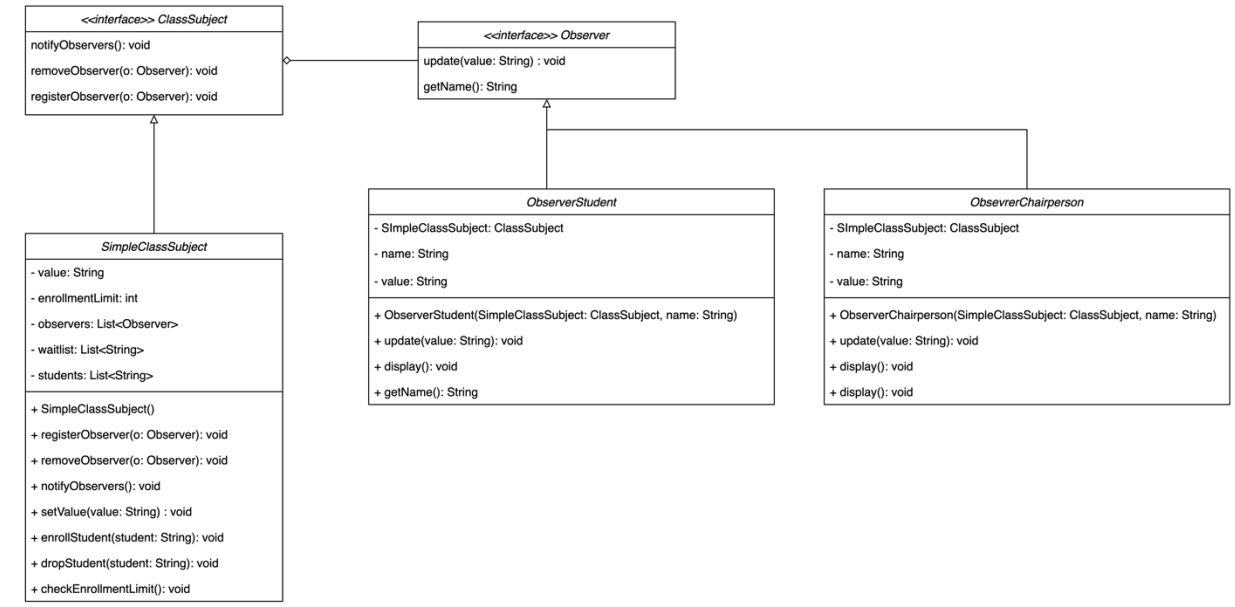
Output:


CS665_Project_Long_WeiNee [:main.main()]: successful At 8/5/23, 10:30 PM 464 ms
-- Testing Chairperson --
Number of Chairperson instances at this moment = 1
New Chairperson created
Chairperson -> General Advisor
NUmber of Course: 1
CS665 added to course list
Course list is full
Course list is full
Course List: [CS665]
-- Testing Full Time Faculty --
FullTime Faculty -> Thesis Advisor -> General Advisor
NUmber of Course: 3
CS665 added to course list
CS666 added to course list
CS667 added to course list
Course List: [CS665, CS666, CS667]
-- Testing Full Time Faculty as Grad Advisor --
FullTime Faculty -> Thesis Advisor -> General Advisor -> 2023 Graduate Advisor
NUmber of Course: 3
CS665 added to course list
CS666 added to course list
CS667 added to course list
Course List: [CS665, CS666, CS667]
-- Testing Full Time Faculty as Undergrad Advisor --
FullTime Faculty -> Thesis Advisor -> General Advisor -> 2023 Undergraduate Advisor
NUmber of Course: 3
CS665 added to course list
CS666 added to course list
CS667 added to course list
Course List: [CS665, CS666, CS667]


5 usages lwlw223
CS665_Project_Long_WeiNee [:main.main()]: successful At 8/5/23, 10:30 PM 464 ms
-- Testing Full Time Faculty as Undergrad Advisor --
FullTime Faculty -> Thesis Advisor -> General Advisor -> 2023 Undergraduate Advisor
NUmber of Course: 3
CS665 added to course list
CS666 added to course list
CS667 added to course list
Course List: [CS665, CS666, CS667]
-- Testing Part Time Faculty --
PartTime Faculty -> General Advisor
NUmber of Course: 1
CS665 added to course list
Course list is full
Course list is full
Course List: [CS665]
BUILD SUCCESSFUL in 402ms
2 actionable tasks: 1 executed, 1 up-to-date

Behavioral Pattern 1: Observer

Class Diagram 5: Behavioral pattern: Observer



Scenario:
When a student tries to enroll in a class that is already full, he will be waitlisted. When someone else drops that class, the wait-listed student will get a notification saying the class has an opening. The chairperson will get a notification whenever the class reaches enrollment limit.

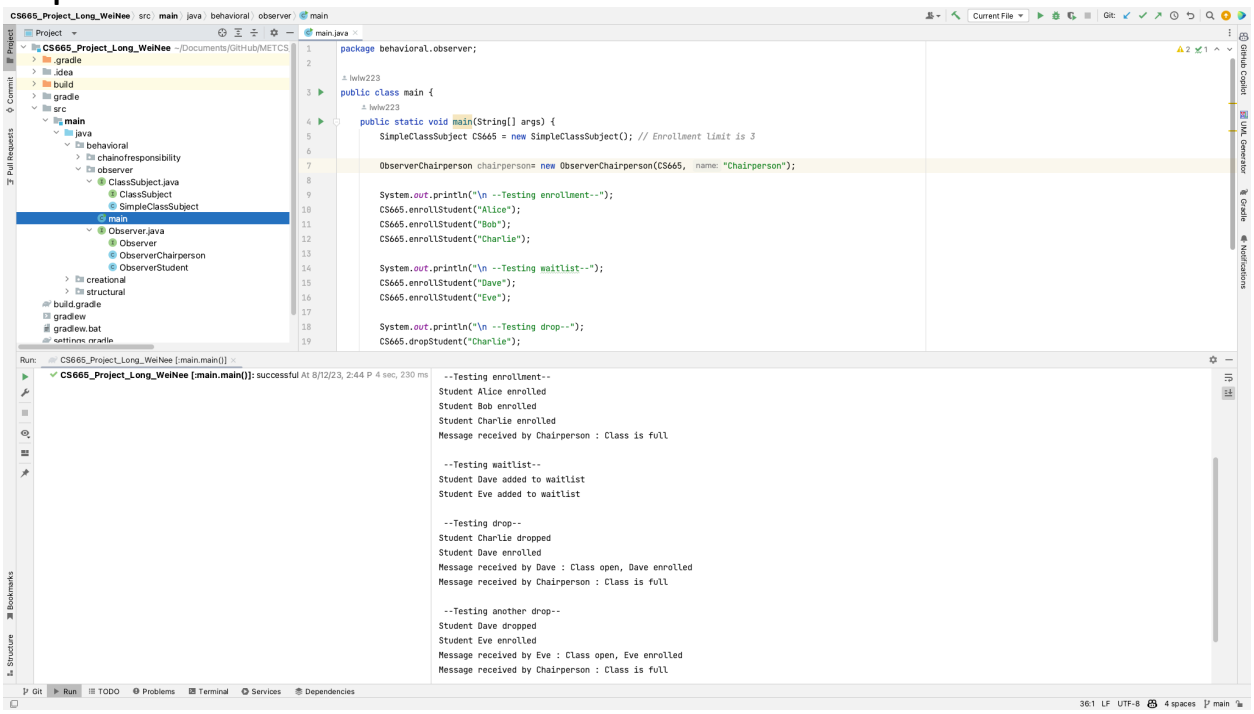
Justification:
The observer pattern is like a subscription mechanism that gives a clean approach to notifying multiple objects about any events that happens to the object they're observing. In this scenario, we have two entities that need to be notified about an event.

- Waitlisted student: Needs to be notified when a class has an opening.
- Chairperson: Needs to be notified when a class reaches enrollment limit.

The observed subject is the class with a limited number of seats. The class has a list of observers (waitlisted students and the chairperson). Whenever a student tries to enroll in a full class, he is added to the waitlist, becoming an observer of the class subject. When someone drops the class, the class subject notifies the first student in the waitlist. The chairperson is notified whenever a class is full.

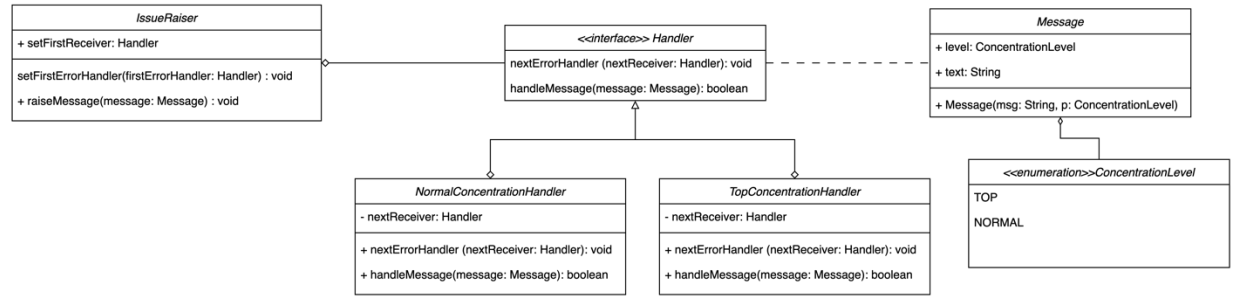
This has allowed us to create a decoupled notification system, where the subject does not need to know about its observers.

Output:


CS665_Project_Long_WeiNee
src
main
java
behavioral
observer
main
Current File
Git:
Project
CS665_Project_Long_WeiNee ~/Documents/GitHub/METCS
.gradle
.idea
build
gradle
src
main
java
behavioral
chainofresponsibility
observer
ClassSubject.java
ClassSubject
SimpleClassSubject
main
Observer.java
Observer
ObserverChairperson
ObserverStudent
creational
structural
build.gradle
gradlew
gradlew.bat
settings.gradle
main.java
package behavioral.observer;
lwlw223
public class main {
lwlw223
public static void main(String[] args) {
SimpleClassSubject CS665 = new SimpleClassSubject(); // Enrollment limit is 3
ObserverChairperson chairperson= new ObserverChairperson(CS665, name: "Chairperson");
System.out.println("\n --Testing enrollment--");
CS665.enrollStudent("Alice");
CS665.enrollStudent("Bob");
CS665.enrollStudent("Charlie");
System.out.println("\n --Testing waitlist--");
CS665.enrollStudent("Dave");
CS665.enrollStudent("Eve");
System.out.println("\n --Testing drop--");
CS665.dropStudent("Charlie");
Run:
CS665_Project_Long_WeiNee [:main.main()]
CS665_Project_Long_WeiNee [:main.main()]: successful At 8/12/23, 2:44 P 4 sec, 230 ms
--Testing enrollment--
Student Alice enrolled
Student Bob enrolled
Student Charlie enrolled
Message received by Chairperson : Class is full
--Testing waitlist--
Student Dave added to waitlist
Student Eve added to waitlist
--Testing drop--
Student Charlie dropped
Student Dave enrolled
Message received by Dave : Class open, Dave enrolled
Message received by Chairperson : Class is full
--Testing another drop--
Student Dave dropped
Student Eve enrolled
Message received by Eve : Class open, Eve enrolled
Message received by Chairperson : Class is full
Git
Run
TODO
Problems
Terminal
Services
Dependencies
36:1
LF
UTF-8
4 spaces
main

Behavioral Pattern 2: Chain of Responsibility

Class Diagram 6: Behavioral pattern: Chain of Responsibility



Scenario:
Full-time faculty member:

- Handle requests related to the specific concentrations they are responsible for.

Chairman:

- Highest level of responsibility
- Handles requests related to top-level concentrations.

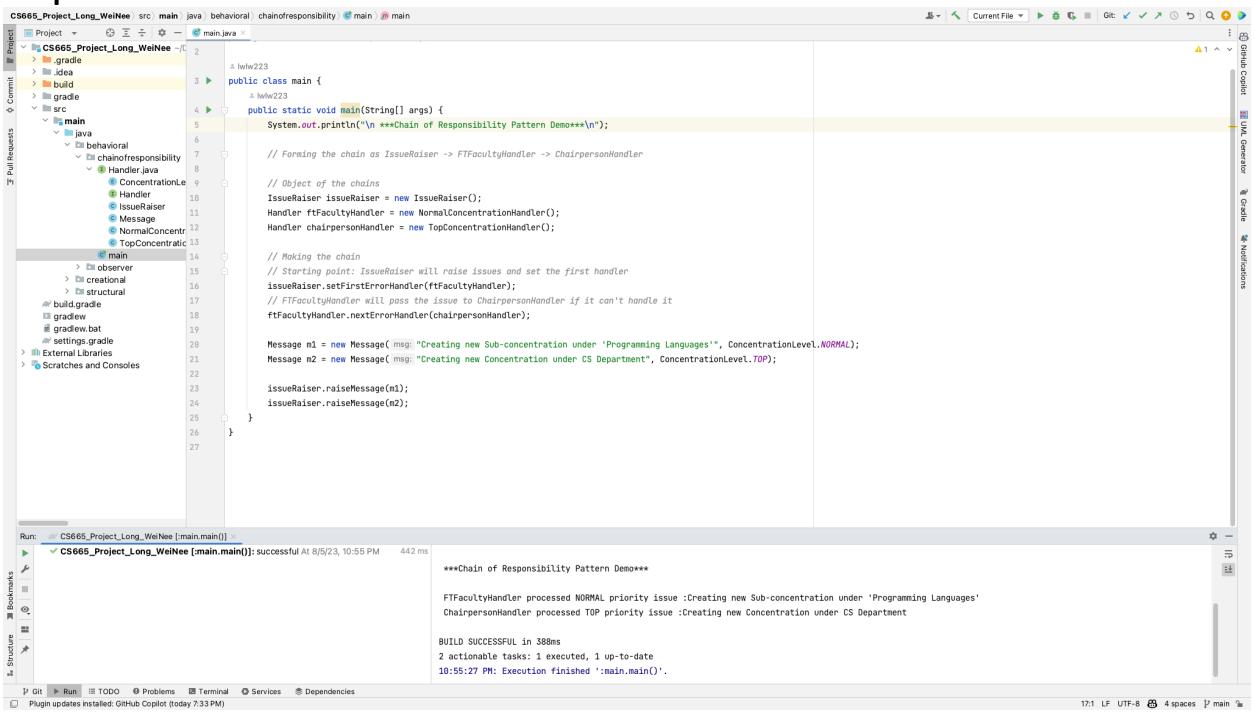
Justification:
The chain of responsibility pattern lets us pass requests along a chain of handlers. Each handler will decide to either process the handler or to pass it to the next handler. In this scenario, we have multiple level of responsibilities based on the hierarchy.

Using the chain of responsibility pattern, we can build a chain of handlers starting from the full-time faculty members to the chairman. When a request is made, it is passed through the chain until a suitable handler is found to handle the request.

For example:

- When a NORMAL request is made, the full-time faculty will handle it.
- When a TOP request is made, the full-time faculty will pass the request to the chairman to handle it.

Output:


CS665_Project_Long_WeiNee
src
main
java
behavioral
chainofresponsibility
main
main
Current File
Git:
Project
main.java
CS665_Project_Long_WeiNee
.gradle
.idea
build
gradle
src
main
java
behavioral
chainofresponsibility
Handler.java
ConcentrationLe
Handler
IssueRaiser
Message
NormalConcentr
TopConcentratic
main
observer
creational
structural
build.gradle
gradlew
gradlew.bat
settings.gradle
External Libraries
Scratches and Consoles
lwlw223
public class main {
lwlw223
public static void main(String[] args) {
System.out.println("\n ***Chain of Responsibility Pattern Demo***\n");
// Forming the chain as IssueRaiser -> FTFacultyHandler -> ChairpersonHandler
// Object of the chains
IssueRaiser issueRaiser = new IssueRaiser();
Handler ftFacultyHandler = new NormalConcentrationHandler();
Handler chairpersonHandler = new TopConcentrationHandler();
// Making the chain
// Starting point: IssueRaiser will raise issues and set the first handler
issueRaiser.setFirstErrorHandler(ftFacultyHandler);
// FTFacultyHandler will pass the issue to ChairpersonHandler if it can't handle it
ftFacultyHandler.nextErrorHandler(chairpersonHandler);
Message m1 = new Message(msg: "Creating new Sub-concentration under 'Programming Languages'", ConcentrationLevel.NORMAL);
Message m2 = new Message(msg: "Creating new Concentration under CS Department", ConcentrationLevel.TOP);
issueRaiser.raiseMessage(m1);
issueRaiser.raiseMessage(m2);
}
}
Run:
CS665_Project_Long_WeiNee [:main.main()]
CS665_Project_Long_WeiNee [:main.main()]: successful At 8/5/23, 10:55 PM
442 ms
Chain of Responsibility Pattern Demo
FTFacultyHandler processed NORMAL priority issue :Creating new Sub-concentration under 'Programming Languages'
ChairpersonHandler processed TOP priority issue :Creating new Concentration under CS Department
BUILD SUCCESSFUL in 388ms
2 actionable tasks: 1 executed, 1 up-to-date
10:55:27 PM: Execution finished ':main.main()'.
Git
Run
TODO
Problems
Terminal
Services
Dependencies
Plugin updates installed: GitHub Copilot (today 7:33 PM)
17:1
LF
UTF-8
4 spaces
main