I’ll show one **representative code snippet** for each type (you’ll have six total).

**1. Behavioral → Strategy Pattern (Payment Methods)**

// PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

// CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

// UpiPayment.java

public class UpiPayment implements PaymentStrategy {

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using UPI.");

}

}

// StrategyDemo.java

public class StrategyDemo {

public static void main(String[] args) {

PaymentStrategy card = new CreditCardPayment();

PaymentStrategy upi = new UpiPayment();

card.pay(1000);

upi.pay(500);

}

}

👉 Shows **Strategy Pattern**: interchangeable algorithms for payments.

**2. Behavioral → Observer Pattern (Task Conflict Notifications)**

// Observer.java

public interface Observer {

void update(String message);

}

// TaskConflictObserver.java

public class TaskConflictObserver implements Observer {

@Override

public void update(String message) {

System.out.println("NOTIFICATION: " + message);

}

}

// Subject.java

import java.util.\*;

public class Subject {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer obs) { observers.add(obs); }

public void notifyObservers(String msg) {

for (Observer o : observers) o.update(msg);

}

}

// ObserverDemo.java

public class ObserverDemo {

public static void main(String[] args) {

Subject subject = new Subject();

subject.addObserver(new TaskConflictObserver());

subject.notifyObservers("New task conflicts with existing one!");

}

}

**3. Creational → Factory Pattern (Shapes)**

// Shape.java

public interface Shape { void draw(); }

public class Circle implements Shape { public void draw(){System.out.println("Circle");}}

public class Rectangle implements Shape { public void draw(){System.out.println("Rectangle");}}

// ShapeFactory.java

public class ShapeFactory {

public Shape getShape(String type) {

switch(type.toLowerCase()) {

case "circle": return new Circle();

case "rectangle": return new Rectangle();

default: throw new IllegalArgumentException("Unknown shape");

}

}

}

// FactoryDemo.java

public class FactoryDemo {

public static void main(String[] args) {

ShapeFactory factory = new ShapeFactory();

Shape s1 = factory.getShape("circle");

s1.draw();

}

}

**4. Creational → Singleton Pattern (Logger)**

public class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) instance = new Logger();

return instance;

}

public void log(String msg) {

System.out.println("[LOG] " + msg);

}

}

**5. Structural → Adapter Pattern (Media Players)**

public interface MediaPlayer { void play(String filename); }

public class Mp3Player implements MediaPlayer {

public void play(String filename){ System.out.println("Playing mp3: " + filename); }

}

public interface AdvancedMediaPlayer { void playVlc(String filename); }

public class VlcPlayer implements AdvancedMediaPlayer {

public void playVlc(String filename){ System.out.println("Playing vlc: " + filename); }

}

public class MediaAdapter implements MediaPlayer {

private AdvancedMediaPlayer adv;

public MediaAdapter(AdvancedMediaPlayer adv){ this.adv = adv; }

public void play(String filename){ adv.playVlc(filename); }

}

**6. Structural → Composite Pattern (Company Employees)**

import java.util.\*;

interface Employee {

void showDetails();

}

class Developer implements Employee {

private String name;

Developer(String name){ this.name = name; }

public void showDetails(){ System.out.println("Developer: " + name); }

}

class Manager implements Employee {

private String name;

private List<Employee> team = new ArrayList<>();

Manager(String name){ this.name = name; }

public void add(Employee e){ team.add(e); }

public void showDetails(){

System.out.println("Manager: " + name);

for(Employee e: team) e.showDetails();

}

}

// CompositeDemo.java

public class CompositeDemo {

public static void main(String[] args){

Manager m = new Manager("Alice");

m.add(new Developer("Bob"));

m.add(new Developer("Charlie"));

m.showDetails();

}

}

**🚀 Exercise 2 — Astronaut Daily Schedule Organizer (Java Console App)**

Key design patterns used:

* **Singleton** → ScheduleManager
* **Factory** → TaskFactory
* **Observer** → ConflictObserver

**Model**

// Priority.java

public enum Priority { HIGH, MEDIUM, LOW }

// Task.java

public class Task {

private String description;

private String startTime;

private String endTime;

private Priority priority;

public Task(String desc, String start, String end, Priority pr){

this.description = desc;

this.startTime = start;

this.endTime = end;

this.priority = pr;

}

public String getDescription(){ return description; }

public String getStartTime(){ return startTime; }

public String getEndTime(){ return endTime; }

public Priority getPriority(){ return priority; }

@Override

public String toString(){

return startTime + " - " + endTime + ": " + description + " [" + priority + "]";

}

}

**Factory**

public class TaskFactory {

public static Task createTask(String desc, String start, String end, Priority pr){

return new Task(desc, start, end, pr);

}

}

**Observer**

public interface TaskObserver {

void onConflict(Task existing, Task newTask);

}

public class ConflictObserver implements TaskObserver {

@Override

public void onConflict(Task existing, Task newTask){

System.out.println("Error: Task '" + newTask.getDescription() +

"' conflicts with '" + existing.getDescription() + "'");

}

}

**Singleton Manager**

import java.util.\*;

public class ScheduleManager {

private static ScheduleManager instance;

private List<Task> tasks = new ArrayList<>();

private List<TaskObserver> observers = new ArrayList<>();

private ScheduleManager(){}

public static ScheduleManager getInstance(){

if(instance == null) instance = new ScheduleManager();

return instance;

}

public void addObserver(TaskObserver obs){ observers.add(obs); }

public void addTask(Task t){

for(Task existing: tasks){

if(isConflict(existing, t)){

notifyConflict(existing, t);

return;

}

}

tasks.add(t);

System.out.println("Task added successfully.");

}

public void removeTask(String desc){

Task toRemove = tasks.stream()

.filter(t -> t.getDescription().equalsIgnoreCase(desc))

.findFirst().orElse(null);

if(toRemove != null){

tasks.remove(toRemove);

System.out.println("Task removed successfully.");

} else {

System.out.println("Error: Task not found.");

}

}

public void viewTasks(){

if(tasks.isEmpty()){ System.out.println("No tasks scheduled for the day."); return; }

tasks.stream().sorted(Comparator.comparing(Task::getStartTime))

.forEach(System.out::println);

}

private boolean isConflict(Task t1, Task t2){

return !(t2.getEndTime().compareTo(t1.getStartTime()) <= 0 ||

t2.getStartTime().compareTo(t1.getEndTime()) >= 0);

}

private void notifyConflict(Task existing, Task newTask){

for(TaskObserver obs: observers) obs.onConflict(existing, newTask);

}

}

**Console App**

import java.util.Scanner;

public class AstronautApp {

public static void main(String[] args) {

ScheduleManager manager = ScheduleManager.getInstance();

manager.addObserver(new ConflictObserver());

Scanner sc = new Scanner(System.in);

while(true){

System.out.println("Enter command (add/remove/view/exit): ");

String cmd = sc.nextLine();

switch(cmd.toLowerCase()){

case "add":

System.out.print("Description: ");

String desc = sc.nextLine();

System.out.print("Start (HH:mm): ");

String start = sc.nextLine();

System.out.print("End (HH:mm): ");

String end = sc.nextLine();

System.out.print("Priority (HIGH/MEDIUM/LOW): ");

Priority pr = Priority.valueOf(sc.nextLine().toUpperCase());

Task task = TaskFactory.createTask(desc, start, end, pr);

manager.addTask(task);

break;

case "remove":

System.out.print("Enter task description to remove: ");

manager.removeTask(sc.nextLine());

break;

case "view":

manager.viewTasks();

break;

case "exit":

System.out.println("Exiting...");

sc.close();

return;

default:

System.out.println("Invalid command.");

}

}

}

}

✅ With this setup:

* You have **six design pattern demos**.
* You have a **console Astronaut Daily Schedule Organizer** with **Singleton + Factory + Observer**.
* Code follows **SOLID** and is interview-friendly.

Do you want me to zip this **Java project with full folder structure** (like I did earlier for C#) so you can **directly upload to GitHub**?

**You said:**

ss

**ChatGPT said:**

Got it 👍 you want a **Java program** for the **Virtual Classroom Manager** problem (Exercise 8).

I’ll build it **step by step** in plain Java (console-based, no frameworks).  
It will use **design patterns** + **SOLID principles** as requested:

* **Singleton** → ClassroomManager (only one instance to manage classrooms/students/assignments).
* **Factory Method** → EntityFactory to create Classroom, Student, Assignment.
* **Observer** → notify when new assignments are scheduled (subscribers = students).

**📌 Java Code: VirtualClassroomManager.java**

import java.util.\*;

// -------------------- Models --------------------

class Classroom {

private String name;

private List<Student> students = new ArrayList<>();

private List<Assignment> assignments = new ArrayList<>();

public Classroom(String name) {

this.name = name;

}

public String getName() { return name; }

public void addStudent(Student student) {

students.add(student);

System.out.println("Student " + student.getId() + " has been enrolled in " + name + ".");

}

public void addAssignment(Assignment assignment) {

assignments.add(assignment);

System.out.println("Assignment for " + name + " has been scheduled.");

// Notify all students (Observer Pattern)

for (Student student : students) {

student.notifyAssignment(assignment);

}

}

public void listStudents() {

if (students.isEmpty()) {

System.out.println("No students enrolled in " + name + ".");

} else {

System.out.println("Students in " + name + ":");

for (Student s : students) {

System.out.println(" - " + s.getId());

}

}

}

public boolean hasStudent(String id) {

return students.stream().anyMatch(s -> s.getId().equals(id));

}

public void submitAssignment(String studentId, String details) {

if (!hasStudent(studentId)) {

System.out.println("Error: Student " + studentId + " not found in " + name + ".");

return;

}

System.out.println("Assignment submitted by Student " + studentId + " in " + name + ".");

}

}

class Student {

private String id;

public Student(String id) {

this.id = id;

}

public String getId() { return id; }

// Observer method

public void notifyAssignment(Assignment assignment) {

System.out.println("Notification for " + id + ": New assignment added -> " + assignment.getDetails());

}

}

class Assignment {

private String details;

public Assignment(String details) {

this.details = details;

}

public String getDetails() { return details; }

}

// -------------------- Factory --------------------

class EntityFactory {

public static Classroom createClassroom(String name) {

return new Classroom(name);

}

public static Student createStudent(String id) {

return new Student(id);

}

public static Assignment createAssignment(String details) {

return new Assignment(details);

}

}

// -------------------- Singleton Manager --------------------

class ClassroomManager {

private static ClassroomManager instance;

private Map<String, Classroom> classrooms = new HashMap<>();

private ClassroomManager() {}

public static ClassroomManager getInstance() {

if (instance == null) {

instance = new ClassroomManager();

}

return instance;

}

public void addClassroom(String name) {

if (classrooms.containsKey(name)) {

System.out.println("Error: Classroom " + name + " already exists.");

return;

}

classrooms.put(name, EntityFactory.createClassroom(name));

System.out.println("Classroom " + name + " has been created.");

}

public void addStudent(String studentId, String className) {

Classroom classroom = classrooms.get(className);

if (classroom == null) {

System.out.println("Error: Classroom " + className + " not found.");

return;

}

classroom.addStudent(EntityFactory.createStudent(studentId));

}

public void scheduleAssignment(String className, String details) {

Classroom classroom = classrooms.get(className);

if (classroom == null) {

System.out.println("Error: Classroom " + className + " not found.");

return;

}

classroom.addAssignment(EntityFactory.createAssignment(details));

}

public void submitAssignment(String studentId, String className, String details) {

Classroom classroom = classrooms.get(className);

if (classroom == null) {

System.out.println("Error: Classroom " + className + " not found.");

return;

}

classroom.submitAssignment(studentId, details);

}

public void listClassrooms() {

if (classrooms.isEmpty()) {

System.out.println("No classrooms available.");

return;

}

System.out.println("Available Classrooms:");

for (String name : classrooms.keySet()) {

System.out.println(" - " + name);

}

}

public void listStudents(String className) {

Classroom classroom = classrooms.get(className);

if (classroom == null) {

System.out.println("Error: Classroom " + className + " not found.");

return;

}

classroom.listStudents();

}

}

// -------------------- Main (Console UI) --------------------

public class VirtualClassroomManager {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

ClassroomManager manager = ClassroomManager.getInstance();

System.out.println("=== Virtual Classroom Manager ===");

System.out.println("Commands: add\_classroom <name>, add\_student <id> <class>, schedule\_assignment <class> <details>, submit\_assignment <id> <class> <details>, list\_classrooms, list\_students <class>, exit");

while (true) {

System.out.print("> ");

String input = sc.nextLine();

String[] parts = input.split(" ", 3);

if (parts[0].equalsIgnoreCase("exit")) break;

switch (parts[0]) {

case "add\_classroom":

if (parts.length < 2) { System.out.println("Usage: add\_classroom <name>"); break; }

manager.addClassroom(parts[1]);

break;

case "add\_student":

if (parts.length < 3) { System.out.println("Usage: add\_student <id> <class>"); break; }

manager.addStudent(parts[1], parts[2]);

break;

case "schedule\_assignment":

if (parts.length < 3) { System.out.println("Usage: schedule\_assignment <class> <details>"); break; }

manager.scheduleAssignment(parts[1], parts[2]);

break;

case "submit\_assignment":

if (parts.length < 3) { System.out.println("Usage: submit\_assignment <id> <class> <details>"); break; }

String[] more = parts[2].split(" ", 2);

if (more.length < 2) { System.out.println("Usage: submit\_assignment <id> <class> <details>"); break; }

manager.submitAssignment(parts[1], more[0], more[1]);

break;

case "list\_classrooms":

manager.listClassrooms();

break;

case "list\_students":

if (parts.length < 2) { System.out.println("Usage: list\_students <class>"); break; }

manager.listStudents(parts[1]);

break;

default:

System.out.println("Unknown command.");

}

}

sc.close();

System.out.println("Exiting Virtual Classroom Manager.");

}

}

**✅ Example Run**

> add\_classroom Math

Classroom Math has been created.

> add\_student S1 Math

Student S1 has been enrolled in Math.

> schedule\_assignment Math Algebra\_HW

Assignment for Math has been scheduled.

Notification for S1: New assignment added -> Algebra\_HW

> submit\_assignment S1 Math Algebra\_HW

Assignment submitted by Student S1 in Math.

> list\_students Math

Students in Math:

- S1