Formating Instructions: You may use any tool to write your homework as long as it has this look like this. We provide a LaTeX template to complete your convenience. When including figures such as UMLs, it is highly encouraged to use tools (graphviz, drawio, tikz, etc..). You may directly insert an image. Figures may be hand draw, however, these may not receive credit if the grader cannot read it. For ease of grading, when including code please have it as part of the same pdf as the question while also including correct formatting/indent, preferably syntax highlighting. Latex includes the minted, or lstlisting package as a helpful tool. For this assignment all code must be full code (no pseudo-code) and be written in either Java or C++. However, implements may ignore all logic not relevant to the design pattern with simple print out statements of "[BLANK] logic done here"

Question Instructions: In this homework assignment, you will apply one or more Structural patterns discussed in the lectures.
This is a group assignment that requires two students per group.
For each question:

1. Give the name of the design pattern(s) you are applying to the problem.
2. Present your reasons why this pattern will solve the problem. Please be specific to the problem and do not give general applicability statements. If there is an alternative pattern, explain why you preferred this one..
3. Show you design with a UML class diagram. If the pattern collaborations would be more visible with another diagram (e.g. sequence diagram), give that diagram as well.
   (a) Your diagram should show every participant in the pattern including the pattern related methods.
   (b) In pattern related classes, give the member (method and attribute) names that play a role in the pattern and effected by the pattern. Optionally, include the member names mentioned in the question. You are encouraged to omit the other methods and fields.
   (c) For the non-pattern related classes, you are not expected to give detailed class names etc. You may give a high-level component, like "UserInterface" or "DBManagement"
4. Give Java or C++ code for your design showing how you have implemented the pattern.
   (a) Pattern related methods and attributes should appear in the code
   (b) Client usage of the pattern should appear in the code
   (c) Non-pattern related parts of the methods could be a simple print. (e.g. "System.out.println()", "cout")
5. Evaluate your design with respect to SOLID principles. Each principle should be address, if a principle is not applicable to the current pattern, say so.

1. (12 points) We have an application thatsss manages and displays data about various kinds of rock formations. The application takes the information from several databases using their APIs. Each database has a different API, which makes our application unnecessarily complicated. We do not want to pollute the code with conditionals that select the right method signature whenever we need to access one of the databases.

   Our application makes the following method calls.

   ```
   public String fetchRockName();
   public String fetchRockType();
   public String fetchRockLocation();
   public Iterator<String> details();
   ```

   Three of the database services provide these methods. However, one database service provides the methods getName(), getType(), getAge(), getComposition(),getLocation(), and getFeatures(). All of these methods return String. Another one provides: rname(), rtype(), rloc(), age(), rdetail(). All of these methods return String except rdetail() returns a list of Strings. Suggest a structural design pattern to make our application work with these database services without conditional statements to select the right method name. (address all items 1-5)

   1. Adapter Pattern
   2. Adaptor pattern is to convert the interface of one class to another, so that incompatible interfaces can work together. In this case:
      - Adaptee → the class that already exists but has an incompatible interface. one service has methods like getName(), getType(), etc., while another service has methods like rname(), rtype(), etc.)
      - Adapter → the new classes to translate the adaptee's API into the interface client expects.
      - Target → the existing interface
      - Client → the application
   3. Show you design with a UML class diagram.

      See attachment "adaptor-rock-application.drawio.pdf"
   4. Give Java or C++ code for your design showing how you have implemented the pattern.

      ```
      // Target Interface
      public interface IRockDataSource {
          String fetchRockName();
          String fetchRockType();
          String fetchRockLocation();
          Iterator<String> details();
      }

      public class StandardDBOne implements IRockDataSource {
          public String fetchRockName() { return "StandardDBOne name"; }
          public String fetchRockType() { return "StandardDBOne type"; }
      ```

```java
    public String fetchRockLocation() { return "StandardDBOne location"; }
    public Iterator<String> details() {
        return Arrays.asList("string", "string").iterator();
    }
}

public class StandardDBTwo implements IRockDataSource {
    public String fetchRockName() { return "StandardDBTwo name"; }
    public String fetchRockType() { return "StandardDBTwo type"; }
    public String fetchRockLocation() { return "StandardDBTwo location"; }
    public Iterator<String> details() {
        return Arrays.asList("string", "string").iterator();
    }
}

public class StandardDBThree implements IRockDataSource {
    public String fetchRockName() { return "StandardDBThree name"; }
    public String fetchRockType() { return "StandardDBThree type"; }
    public String fetchRockLocation() { return "StandardDBThree location"; }
    public Iterator<String> details() {
        return Arrays.asList("string", "string").iterator();
    }
}

// Incompatible service #1
class IncompatibleDBOne {
    public String getName() { return "IncompatibleDBOne getName"; }
    public String getType() { return "IncompatibleDBOne getType"; }
    public String getAge() { return "IncompatibleDBOne getAge"; }
    public String getComposition() { return "IncompatibleDBOne getComposition"; }
    public String getLocation() { return "IncompatibleDBOne getLocation"; }
    public String getFeatures() { return "IncompatibleDBOne getFeatures"; }
}

calss IncompatibleDBOneAdaptor implements IRockDataSource {
    private IncompatibleDBOne incompatibleDB;
    public IncompatibleDBOneAdaptor(IncompatibleDBOne dbService){
        this.incompatibleDB = dbService;
    }

    @Override
    public String fetchRockName() { return incompatibleDB.getName(); }

    @Override
    public String fetchRockType() { return incompatibleDB.getType(); }

    @Override
```

```java
    public String fetchRockLocation() { return incompatibleDB.getLocation(); }

    @Override
    public Iterator<String> details() {
        return Arrays.asList(
            incompatibleDB.getAge(),
            incompatibleDB.getComposition(),
            incompatibleDB.getFeatures(),
            ).iterator();
    }
}

// Incompatible service #2
class IncompatibleDBTwo {
    public String rname() { return "IncompatibleDBTwo rname"; }
    public String rtype() { return "IncompatibleDBTwo rtype"; }
    public String rloc() { return "IncompatibleDBTwo rloc"; }
    public String age() { return "IncompatibleDBTwo age"; }
    public List<String> rdetail() {
        return Arrays.asList("string", "string");
    }
}

calss IncompatibleDBTwoAdaptor implements IRockDataSource {
    private IncompatibleDBTwo incompatibleDB;
    public IncompatibleDBTwoAdaptor(IncompatibleDBTwo dbService){
        this.incompatibleDB = dbService;
    }

    @Override
    public String fetchRockName() { return incompatibleDB.rname(); }

    @Override
    public String fetchRockType() { return incompatibleDB.rtype(); }

    @Override
    public String fetchRockLocation() { return incompatibleDB.rloc(); }

    @Override
    public Iterator<String> details() {
        return incompatibleDB.rdetail().iterator();
    }
}

public class Main {
public static void main(String[] args) {
        List<RockDataSource> sources = Arrays.asList(
```

```java
            new StandardDBOne(),
            new StandardDBTwo(),
            new StandardDBThree(),
            new IncompatibleDBOneAdaptor( new IncompatibleDBOne() ),
            new IncompatibleDBTwoAdaptor( new IncompatibleDBTwo() ),
        );

        for (RockDataSource src : sources) {
            System.out.println("Rock: " + src.fetchRockName());
            System.out.println("Type: " + src.fetchRockType());
            System.out.println("Location: " + src.fetchRockLocation());
            System.out.println("Details: ");
            src.details().forEachRemaining(System.out::println);
        }
    }
}
```

5. Evaluate your design with respect to SOLID principles.

- Single Responsibility Principle: Every adapter has only one responsibility.
- Open-Closed Principle: New adapters can be added in the future without modifying existing logic
- Liskov Substitution Principle: N/A
- Interface Segregation Principle: Client only depends on 'IRockDataSource', which only contains the necessary methods
- Dependency Inversion Principle: client depends on the abstraction (RockDataSource) rather than concrete classes

2. (14 points) We are developing a new mobile game with a "Base Builder" theme. Players can construct complex structures on a map. These structures are composed of smaller, individual building components. For example, a `Fortress` might be made of `Walls`, `Towers`, and a `Gate`. A `Tower` might, in turn, be made of a `Base`, a `Body`, and a `Roof`. The game needs to perform operations on these structures, such as `repair`, `upgrade` or `destroy`.

   The challenge is that the game's logic needs to treat a single component (like a `Wall` and a complete structure (like an entire `Fortress` uniformly. For example, a player should be able to click on a single wall to repair it, but they should also be able to select the entire fortress and issue a single command to repair every component within it. Currently, the code has separate methods and complex conditional logic to handle single components versus groups, leading to a brittle and unmanageable codebase.

   Suggest a structural design pattern to simplify the management of these in-game structures.

   Initially, we come up with these classes: Wall, Gate, Roof, Fortress, Tower, and Barracks. The operations we have are `repair()` and `destroy()` among others.

   In your design, show which class(es) or which objects plays which participant of the pattern. (you may use notes on the UML class diagram or just a few sentences under the diagram.)

   Address all items 1-5 on the title page of the homework.

Additional tasks:

Client Code: Behavior Simulation What is expected in item 4(b).

Write code/pseudocode or class stubs to simulate

- Building a Structure: Create a `Fortress` object with several walls and then create a `Tower` object with walls and a roof and make it a part of the Fortress.
- Repair Service(using Dependency Injection): Write a method that takes an object of type `Wall`, `Gate`, `Roof`, `Fortress`, `Tower`, or `Barracks`. This method calls the `repair` operation on the object it receives without needing to know if it's a single wall or a more complex building.

  Simulate a repair process on a single `Wall` object.

  Simulate a repair process on the entire `Fortress` object.

Extensibility in Action:

New Component: Imagine a new building component, a Cannon. A Cannon is a single piece of equipment that can be added to a Fortress.

Challenge: Extend your design to support the Cannon without modifying your existing Repair Service method above.

Show how to add a `Cannon` to the existing hierarchy.

1. Composite Pattern
2. The entire construction follows a tree structure which is what composite pattern designed to deal with. With composite pattern, clients can treat individual objects (such as Walls and Gate) and composite objects (such as Fortress and Tower) uniformly. Other consideration such as façade pattern might be good in terms of simplifying the composite structures, but the client might still have to add a bunch of conditional logic to handle `repair()` and `destroy()` on each object.
3. Show you design with a UML class diagram.

   See attachment "composite-builder-application.drawio.pdf"
4. Give Java or C++ code for your design showing how you have implemented the pattern

```java
import java.util.ArrayList;
import java.util.List;

interface BuildingComponent {
    void repair();

    void destroy();
}

// leaf
class Wall implements BuildingComponent {
    String name;
```

```java
    public Wall(String name) {
        this.name = name;
    }

    // @Override
    public void repair() {
        System.out.println("Repairing " + name);
    }

    // @Override
    public void destroy() {
        System.out.println("Destroying " + name);
    }
}

class Roof implements BuildingComponent {
    String name;

    public Roof(String name) {
        this.name = name;
    }

    // @Override
    public void repair() {
        System.out.println("Repairing " + name);
    }

    // @Override
    public void destroy() {
        System.out.println("Destroying " + name);
    }
}

class Gate implements BuildingComponent {
    String name;

    public Gate(String name) {
        this.name = name;
    }

    // @Override
    public void repair() {
        System.out.println("Repairing " + name);
    }

    // @Override
    public void destroy() {
```

```java
        System.out.println("Destroying " + name);
    }
}

/**
 * New Component: Imagine a new building component, a Cannon.
 * A Cannon is a single piece of equipment that can be added to a Fortress.
 */
class Cannon implements BuildingComponent {
    String name;

    public Cannon(String name) {
        this.name = name;
    }

    // @Override
    public void repair() {
        System.out.println("Repairing " + name);
    }

    // @Override
    public void destroy() {
        System.out.println("Destroying " + name);
    }
}

// composite
class CompositeStructure implements BuildingComponent {
    private String name;
    private List<BuildingComponent> children = new ArrayList<>();

    public CompositeStructure(String name) {
        this.name = name;
    }

    public void add(BuildingComponent component) {
        children.add(component);
    }

    public void remove(BuildingComponent component) {
        children.remove(component);
    }

    @Override
    public void repair() {
        System.out.println("Repairing " + name);
        for (BuildingComponent c : children) {
```

```java
            c.repair();
        }
    }

    @Override
    public void destroy() {
        System.out.println("Destroying " + name);
        for (BuildingComponent c : children) {
            c.destroy();
        }
    }
}

class Fortess extends CompositeStructure {
    public Fortess(String name) {
        super(name);
    }
}

class Tower extends CompositeStructure {
    public Tower(String name) {
        super(name);
    }
}

class RepairService {
    public void performRepair(BuildingComponent component) {
        component.repair();
    }
}

public class client {
    public static void main(String[] args) {

        /**
         * Building a Structure:
         * Create a Fortress object with several walls and then create a
         * Tower object with walls and a roof and make it a part of the Fortress.
         */
        Fortess fortress = new Fortess("Fortress");
        fortress.add(new Wall("Fortress Wall 1"));
        fortress.add(new Wall("Fortress Wall 2"));

        Tower tower = new Tower("tower");
        tower.add(new Wall("Tower Wall 1"));
        tower.add(new Wall("Tower Wall 2"));
```

```
        fortress.add(tower);

        /**
         * Repair Service(using Dependency Injection)
         */
        RepairService repairService = new RepairService();
        // Repair single Wall
        Wall singleWall = new Wall("Single Wall");
        repairService.performRepair(singleWall);

        /** Extensibility in Action: adding a cannon without modifying repairService */

        Cannon cannon = new Cannon("Cannon 1");
        // Add new Cannon component to the fortress
        fortress.add(cannon);

        // Repair a fortress
        repairService.performRepair(fortress);

    }
}
```

5. Evaluate your design with respect to SOLID principles.

   - Single Responsibility Principle: Wall repairs itself and Fortress manages its own children
   - Open-Closed Principle: New components like Cannon can be added without modifying existing logic
   - Liskov Substitution Principle: The ‘repair‘ and ‘destroy‘ methods defined in BuildingComponent work on both leaf (Wall, Roof..) and composite structure (Fortress, Tower...)
   - Interface Segregation Principle: There are only two methods in ‘BuildingComponent‘
   - Dependency Inversion Principle: For example, the RepairService depends on the abstraction class (BuildingComponent)

Table 1: Grading Rubric for **12** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (4 points)	0	missing
	+2	includes all participants (including client) that play a role in the pattern
	+1	all class relations are correct
	+1	includes all class members that are related to the pattern
4 (4 points)	0	missing
	+1	includes all pattern related methods and attributes
	+2	includes client usage
	+1	correctly implements and uses all pattern related methods
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user

Table 2: Grading Rubric for **14** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (5 points)	0	missing
	+2	includes all participants (including client) that play a role in the pattern
	+2	all class relations are correct
	+1	includes all class members that are related to the pattern
4 (5 points)	0	missing
	+1	includes all pattern related methods and attributes
	+2	includes client usage
	+2	correctly implements and uses all pattern related methods
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user