Formating Instructions: Please use this provided LaTeX template to complete your homework. When including figures such as UMLs, it is highly encouraged to use tools (graphviz, drawio, tikz, etc..). 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 the Structural patterns discussed in the lectures.
This is an individual assignment, you may not work in groups
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 addressed, if a principle is not applicable to the current pattern, say so.

1. (14 points) We have an application that 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
   2. Expected database service method names are different than the ones provided by the latter two services. Both names and number of the methods are different. The adapter pattern makes the application work with these incompatible interfaces.
   3. UML

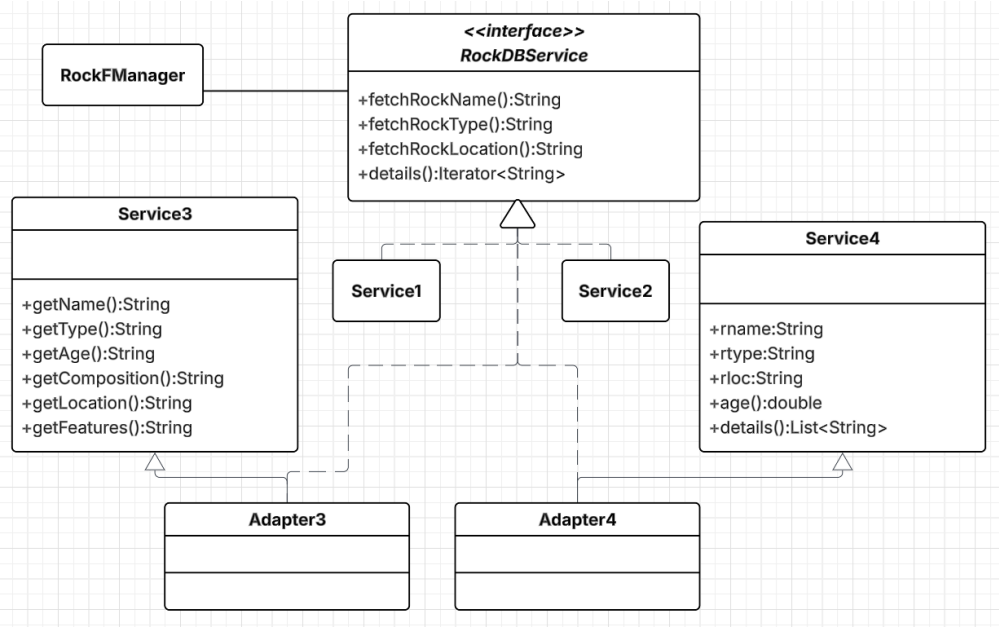

Figure 1: Option1: class Adapter

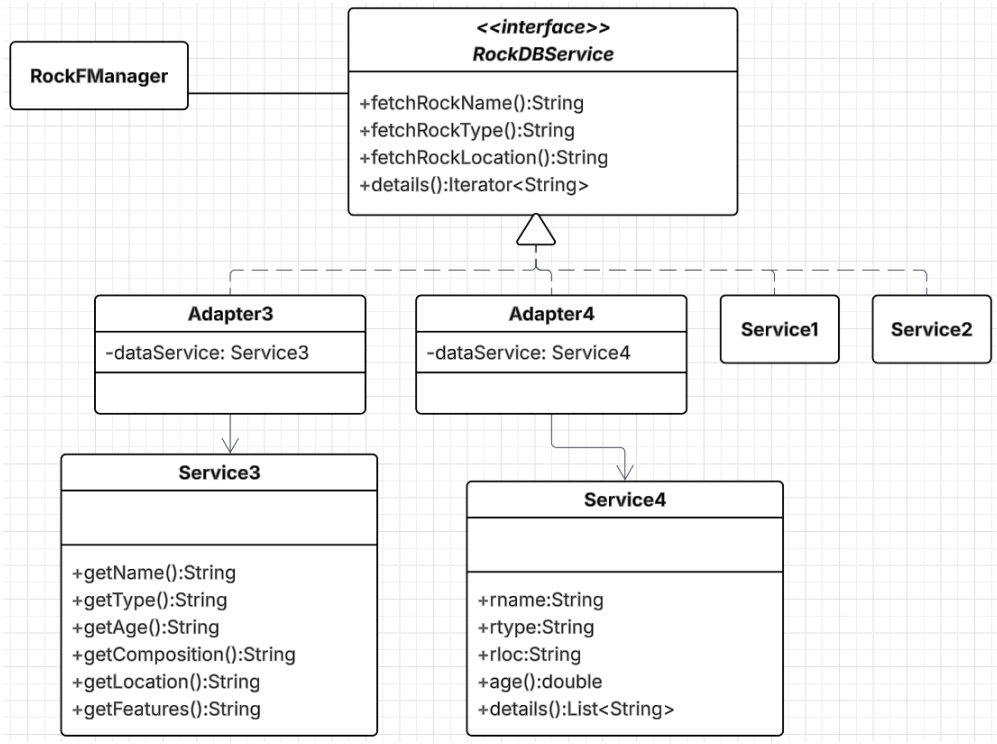

Figure 2: Option2: object Adapter

4. Code Option 1 (class Adapter):

```
public interface RockDBService{ //Target interface
    public String fetchRockName();
    public String fetchRockType();
    public String fetchRockLocation();
    public Iterator<String> details() ;
}
public class Adapter3 implements RockDBService extends Service3{ //Adapter
     /*could use any name for Service3*/ //Service3 is the adaptee
     public Adapter3() {super();}  /* no constructor is ok in java*/
    public String fetchRockName(){
            return super.getName();}
    public String fetchRockType(){
         return super.getType();}
    public String fetchRockLocation(){
        return super.getLocation();}
    public Iterator<String> details() {
        List<String> lst=new ArrayList<String>();
        lst.add(super.getComposition();
        lst.add(super.getAge();
        lst.add(super.getFeatures();
        return lst.iterator(); }
```

```
}
public class Adapter4 implements RockDBService extends Service4{ //Adapter
     public String fetchRockName(){
            return super.rname();}  /*just rname(); is ok as well*/
    public String fetchRockType(){
         return rtype();}
    public String fetchRockLocation(){
        return rloc();}
    public Iterator<String> details() {
        List<String> lst=new ArrayList<String>(rdetail());
        lst.add(age();
        return lst.iterator(); }
}

//Client usage:
public class App{
   public void someSetupMethod(){
              RockDBService service3=new Adapter3();
              /* must show this object creation*/
              RockDBService service4=new Adapter4();
 }
 public void someQuery(){
        /*this is an example. There could be other scenarios*/
         for(RockDBService s:services){
                System.out.println(s. fetchRockName());
                System.out.println(s. fetchRockType());
                System.out.println(s. fetchRockLocation());
                     s.details().forEach(d -> System.out.println(d));
        }
 }
  private List<RockDBService> services=new ArrayList< RockDBService>();
/*there could be other ways to save the references to the adapters*/
  /*....other methods**/
}
```

Option 2 (object Adapter):

```
    public interface RockDBService{
    public String fetchRockName();
    public String fetchRockType();
    public String fetchRockLocation();
    public Iterator<String> details() ;
}
public class Adapter3 implements RockDBService{
     private Service3 /*could be any name*/    dataservice;
     public Adapter3(Service3 adaptedService){
           dataservice=adaptedService;   }
    public String fetchRockName(){
```

```
            return dataservice.getName();}
    public String fetchRockType(){
         return dataservice.getType();}
    public String fetchRockLocation(){
        return dataservice.getLocation();}
    public Iterator<String> details() {
        List<String> lst=new ArrayList<String>();
        lst.add(dataservice.getComposition();
        lst.add(dataservice.getAge();
        lst.add(dataservice.getFeatures();
        return lst.iterator(); }
}
public class Adapter4 implements RockDBService{
     private Service4 /*could be any name*/    dataservice;
     public Adapter4(Service4 adaptedService){
           dataservice=adaptedService;   }
    public String fetchRockName(){
            return dataservice.rname();}
    public String fetchRockType(){
         return dataservice.rtype();}
    public String fetchRockLocation(){
        return dataservice.rloc();}
    public Iterator<String> details() {
        List<String> lst=new ArrayList<String>( dataservice.rdetail());
        lst.add(dataservice.age();
        return lst.iterator(); }
}
//Client usage:
public class App{
   public void someSetupMethod(){
           RockDBService service3=new Adapter3(new Service3());
           /*must show this object creation*/
           RockDBService service4=new Adapter4(new Service4());
 }
 public void someQuery(){
        /*this is an example. There could be other scenarios*/
         for(RockDBService s:services){
                System.out.println(s. fetchRockName());
                System.out.println(s. fetchRockType());
                System.out.println(s. fetchRockLocation());
                     s.details().forEach(d -> System.out.println(d));
        }
 }
  private List<RockDBService> services=new ArrayList< RockDBService>();
/*there could be other ways to save the references to the adapters*/
  /*....other methods**/
```

```
}
```

5. SOLID
   - Single responsibility: adapter's responsibility is interface conversion. There is no other responsibility. Having this responsibility in the adapter relives the client code to make method name selection based on the database service name.
   - Open-closed: For another database service, implement a new adapter (extension) without affecting the rest of the code. If object adapter is used, subclasses of the service could be adapted without changing the rest of the code.
   - Liskov Sub.: Adapter is a subtype of the service. It can be used in any place where the actual service is used without affecting the correctness/assumptions of the client code.
   - Interface segregation: Adapter only implements the interface our application needs, which are these four methods.
   - Dependency inversion: Application depends on the database service interface rather than the adapter or the actual service class.

2. (14 points) We are developing a new mobile game with a "Base Builder" theme. Players cancconstruct 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)

   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
   2. The problem is about treating individual parts of a nested object hierarchy uniformly. The composite pattern will allow the client to interact with all composite pieces such as a fortress with the same interface as a leaf piece like a wall.
   3. UML

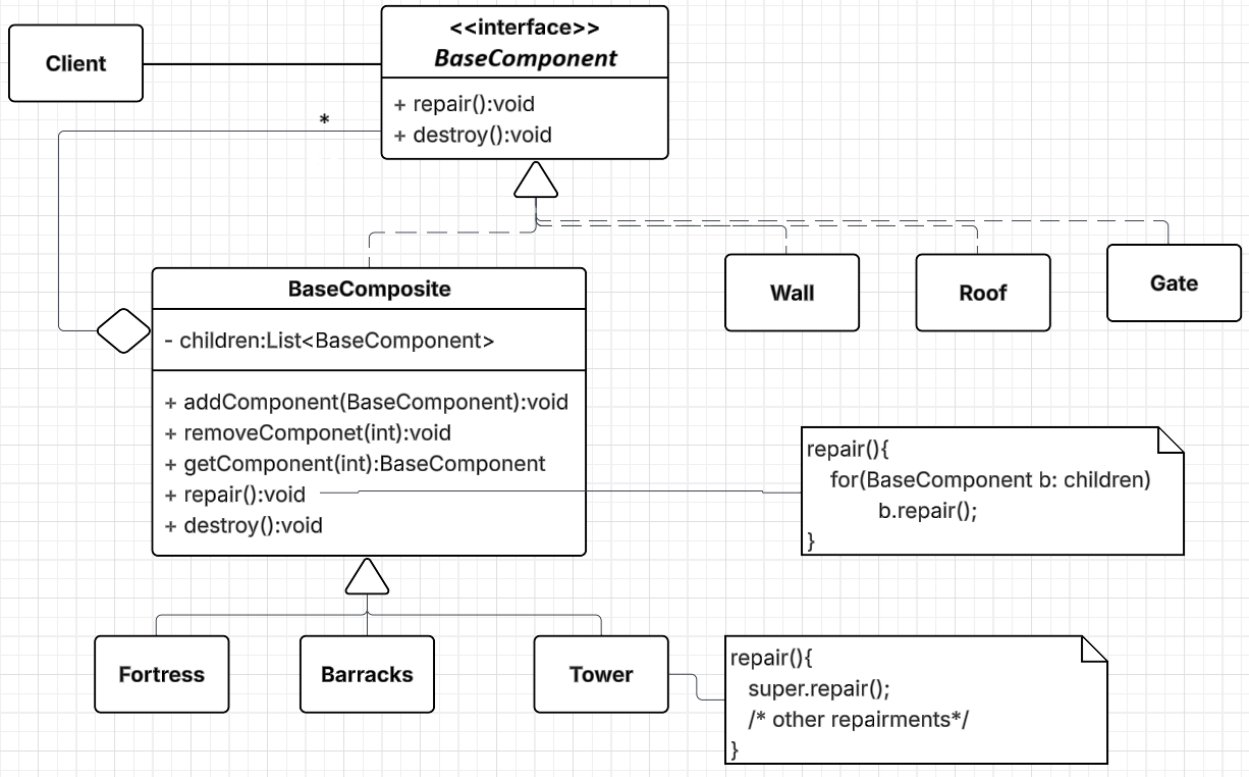

Figure 3: OPTION1:BaseComponent is the Component class, BaseComposite is an abstract composite class while Fortress, Barracks, and Tower are all concrete composite classes. Wall, Roof, and Gate are all leaves.

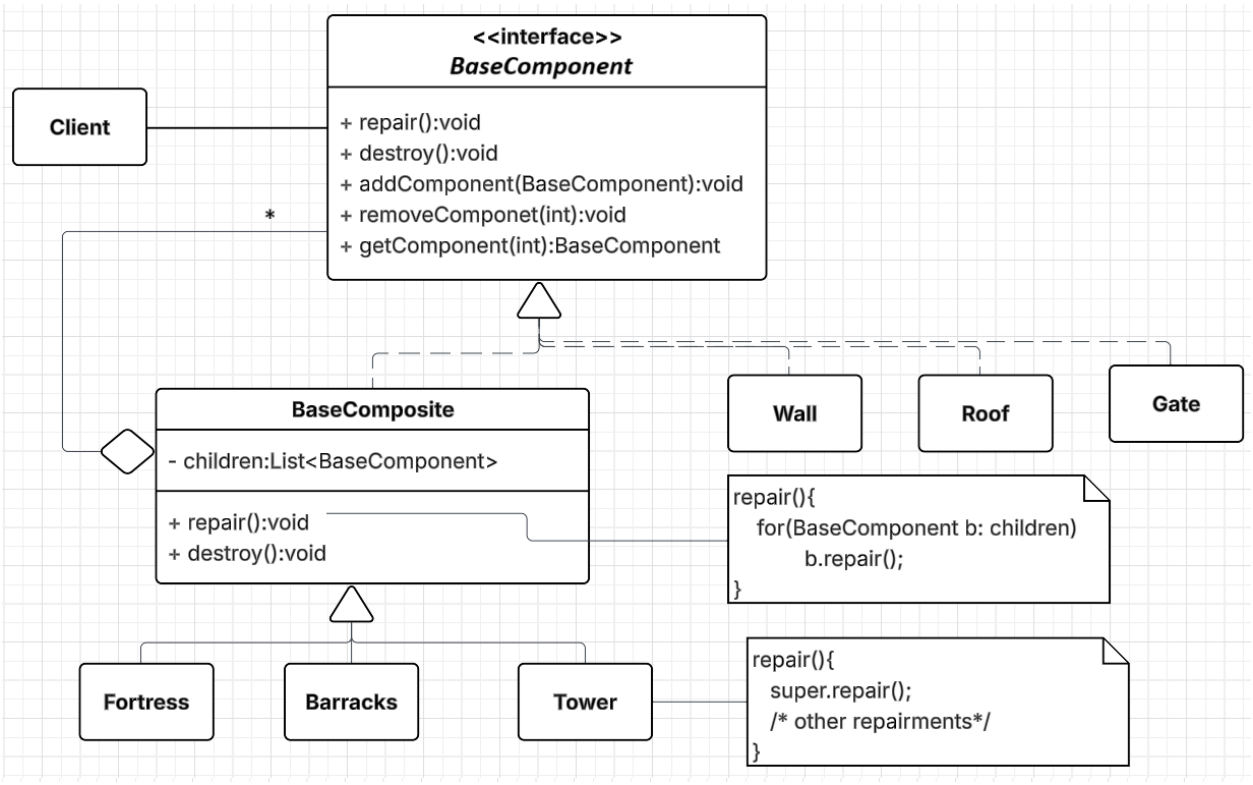

Figure 4: OPTION2:BaseComponent is the Component class, BaseComposite is an abstract composite class while Fortress, Barracks, and Tower are all concrete composite classes. Wall, Roof, and Gate are all leaves.

4. Code

```
//The Component interface
public interface BaseComponent{
    public void repair();
    public void destroy();
    /* OPTION 1 -have these child management methods here
        and treat the Leaf as if it has zero children.*/
    public void addComponent(BaseComponent);
    public void removeComponent(int);
    public BaseComponent getComponent(int);
    /*OPTION 2 - do not have these methods here.
        Loosing uniformity for Interface seggregation*/
}

//an abstract composite
public abstract class BaseComposite extends BaseComponent{
    private List<BaseComponent> children = new List<BaseComponent>();
    public abstract void repair();
    public abstract void destroy();
    public void addComponent(BaseComponent b) {children.add(b);};
```

```
20      public void removeComponent(int i) {children.removeComponent(i);};
21      public BaseComponent getComponent(int index){return
         ↪  children.get(index);};
22
23          /* Alternative-1: only have the above methods*/
24          /* Alternative-2: have a default implementation for repair and
             ↪  destroy operations
25          that directs all the children do the operation.
26          Then, let the subclass use it with super.op() */
27      public void repair(){
28          for (BaseComponent b : this.children){
29              b.repair()
30          }
31      public void destroy(){
32          for (BaseComponent b : this.children){
33              b.destroy()
34          }
35          /*end of Alternative-2*/
36          /*Do not implement both as it will complicate the design */
37  }
38
39  //concrete composite --Alternative-1
40  public class Fortress extends BaseComposite{
41      public void repair(){  /*Alternative 1*/
42          for (BaseComponent b : this.children){
43              b.repair()
44          }
45          /*repairs fortress*/
46          System.out.println("repairs fortress");
47      }
48
49      public void destroy(){
50          for (BaseComponent b : this.children){
51              b.destroy()
52          }
53          /*destroys fortress*/
54          System.out.println("destroys fortress");
55      }
56  }
57
58  //concrete composite  --Alternative -2
59  public class Barracks extends BaseComposite{
60      public void repair(){
61          super.repair();
62          /*repairs Barracks*/
63          System.out.println("repairs barracks");
```

```
64      }
65
66      public void destroy(){
67          super.destroy();
68          /*destroys Barracks*/
69          System.out.println("destroys barracks");
70      }
71  }
72
73  //concrete composite --Alternative-1
74  public class Tower extends BaseComponent{
75      public void repair(){
76          for (BaseComponent b : this.children){
77              b.repair()
78          }
79          /*repairs Tower*/
80          System.out.println("repairs tower");
81      }
82
83      public void destroy(){
84          for (BaseComponent b : this.children){
85              b.destroy()
86          }
87          /*destroys Tower*/
88          System.out.println("destroys tower");
89      }
90      public void addComponent(BaseComponent b) {
91          /*The composite pattern does not say this,
92          * but we can tweak it a bit to restrict what components to add*/
93
94          //add only non-BaseComposite types
95          //to prevent towers having fortresses
96          if( b instanceof BaseComposite)
97             //loosing information here. we can return -1 for warning.
98             return;
99
100         children.add(b);
101     }
102 }
103
104 //leaf
105 public class Wall extends BaseComponent{
106     public void repair(){/*repairs wall*/
107         System.out.println("repairs wall");
108     }
109     public void destroy(){/*destroys wall*/
```

```
110             System.out.println("destroys wall");
111         }
112     }
113
114     //leaf
115     public class Roof extends BaseComponent{
116         public void repair(){/*repairs roof*/
117             System.out.println("repairs roof");
118         }
119         public void destroy(){/*destroys roof*/
120             System.out.println("destroys roof");
121         }
122     }
123
124     //leaf
125     public class Gate extends BaseComponent{
126         public void repair(){/*repairs gate*/
127             System.out.println("repairs gate");
128         }
129         public void destroy(){/*destroys gate*/
130             System.out.println("destroys gate");
131         }
132     }
133
134
135
136     public class Main {
137         public static void repairService(BaseComponent b) {b.repair();};
138
139         public static void main(String args[]){
140             BaseComponent myFort = new Fortress();
141             BaseComponent myWall = new Wall();
142             myBase.addComponent(myWall);
143             myBase.addComponent(new Wall());
144             myBase.addComponent(new Wall());
145             myBase.addComponent(new Wall());
146             myBase.addComponent(new Gate());
147
148             BaseComponent myTower = new Tower();
149             myTower.addComponent(new Wall());
150             myTower.addComponent(new Wall());
151             myTower.addComponent(new Wall());
152             myTower.addComponent(new Wall());
153             myTower.addComponent(new Gate());
154             myTower.addComponent(new Roof());
155
```

```
156             myBase.addComponent(myTower);
157
158             repairService(myWall);
159             repairService(myFort);
160         }
161     }
```

5. SOLID
    - Single responsibility: Composite breaks down reponsibilities in a part whole heirarchy so that all members can ensure a single responsibility.
    - Open-closed: Adding new components to the composite structure does not require any modifications to existing code.
    - Liskov Subsitution: Composites substitute for their Component super type without affecting correctness.
    - Interface segregation: (When using Option 1) This implementation forces leafs to include add and remove component methods that they don't use, violating ISP. This can be avoided by not including child management inside the component interface, but then there will be a difference between composites and leafs for the user.
    - Dependency inversion: Decouples a high-level component from the low level implementation of the individual pieces. The user can rely on abstractions of a single interface without knowing the details of each leaf and components implementation.

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
	+2	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
	+2	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
	+1	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
	+1	correctly implements and uses all pattern related methods
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user