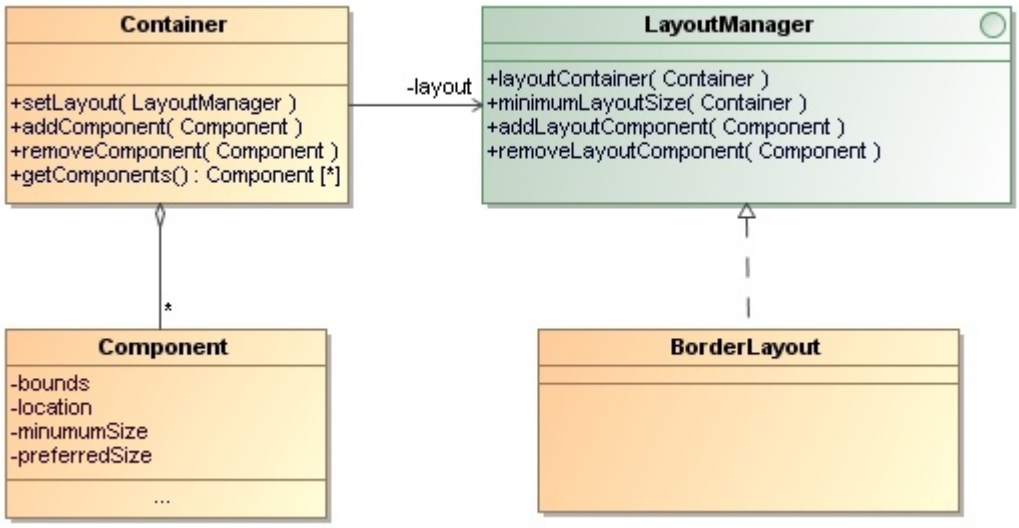
1. Java Layout Managers are an example of the Strategy pattern.
a. Complete the following table:

Name in Pattern	Name in This Example
`Context`	
`Strategy`	
`ConcreteStrategy`	
`setStrategy( Strategy )`	
`doStrategy( Context )`	



b. What is the *motivation (forces)* for using the Strategy Pattern here?

c. What is/are the benefit(s) of using the Strategy Pattern? Compare to alternative of coding the layout behavior in the container itself.

2. *Coupling*: Many applications perform *Logging*. An application uses a *logger* to record errors, warnings, and other messages to a file, database, or network server.
Here is a `BankAccount` class that uses a `ConsoleLogger` to record messages.

```
public class ConsoleLogger {
     /** log level controls which messages are displayed */
     private int loglevel = 1;
     /** write error messages */
     public void error( String msg ) {
          if ( loglevel >= 0 )
          System.out.println( "Error: " + msg );
     }
     /** write warning messages */
     public void warning( String msg ) {
          if ( loglevel >= 1 )
          System.out.println( "Warning: " + msg );
     }
     /** write information messages */
     public void info( String msg ) {
          if ( loglevel >= 2 )
          System.out.println( "Info: " + msg );
     }
     /** set the logging level */
     public void setLoglevel( int level ) { loglevel = level; }
}
```

Here's part of the BankAccount to illustrate how logging is used:

```
public class BankAccount {
     ConsoleLogger log = new ConsoleLogger( );

     public void deposit(double amount ) {
          if (amount < 0 )
               log.error( "Attempt to deposit negative money" );
          else if ( amount == 0 )
               log.warning( "Attempt to deposit nothing" );
          else {
               balance = balance + amount;
               log.info("Deposited " + amount );
          }
     }
}
```

Which class is *coupled to* (*depends on*) which other class?

How could you define and use an *interface* (let's call it "Logger") to eliminate this coupling?

Draw a UML class diagram showing the methods of your interface, the ConsoleLogger, and BankAccount. Show the relationships between them.

3. After you define the Logger interface, you ***still*** have a problem with coupling: how is the BankAccount going to get a Logger object?

If the BankAccount creates the logger itself, like this:

```
Logger logger = new ConsoleLogger( );
```

then we still have coupling to ConsoleLogger.

(a) How could we use *dependency injection* to eliminate the dependency on ConsoleLogger? Write a class diagram and example Java code.

(b) Design another way to eliminate the dependency on ConsoleLogger in BankAccount.

4. Study the Player-Role pattern and answer these questions.

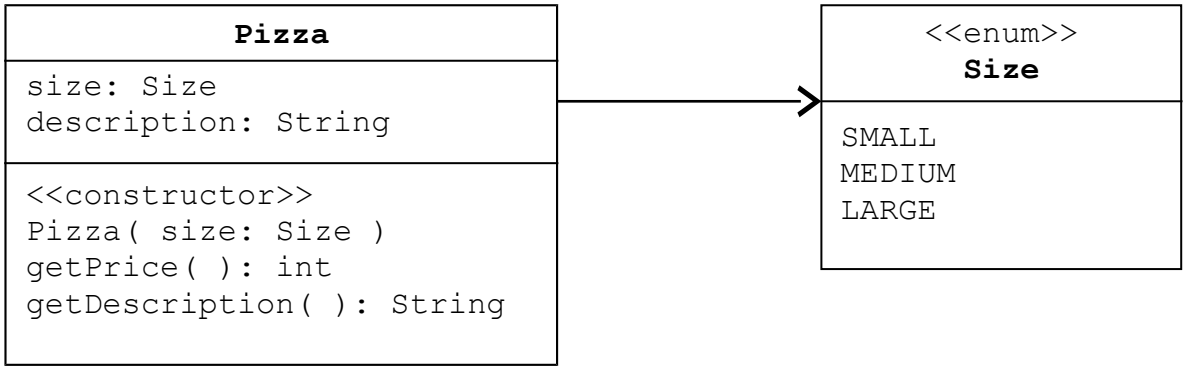
a) What is the situation where the pattern applies?
b) What are the forces (goals) that motivate this solution?
c) Draw UML diagrams (class and/or sequence) to show how it is used.
d) Where in eXceedVote might we want to apply this pattern?

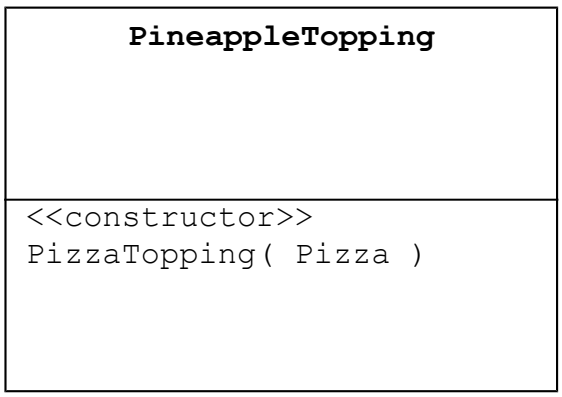
```
Pizza pizza = new Pizza( Size.LARGE );
System.out.println( pizza.getDescription( ) );
          // prints "large pizza"
System.out.println( pizza.getPrice() );
          // prints 180 (maybe)
```

A pizza with pineapple topping would behave like this (using same `pizza` as above)

```
pizza = new PineappleTopping( pizza );
System.out.println( pizza.getDescription( ) );
          // prints "large pizza with pineapple"
System.out.println( pizza.getPrice() );
          // prints 230 (if pineapple costs 50 Baht)
```

Complete the UML diagram below showing the relation between the PizzaTopping and Pizza. Show <u>all</u> relations between the topping and the pizza. There is more than one possible correct answer, but you *must* use the decorator pattern.





What methods must a pizza topping have, according to decorator pattern?

Write code for the `getDescription( )` method of `PineappleTopping`. This illustrates how the decorator pattern works.

6. Suppose the Pizza class also has a setSize( ) and getSize( ) method. The decorator doesn't want to change how these methods work. Does PineappleTopping need to override these methods or not?

If "no", explain why.

If "yes", give example code for how they should be overridden.

7. What is the difference between a *Decorator* and an *Adapter*?