1. Fundamental Patterns

On the forum code directory you will find the classes for the shape hierarchy as shown in Figure 1. This hierarchy will be used as basis for many exercises. You should be familiar with UML Class Diagrams.

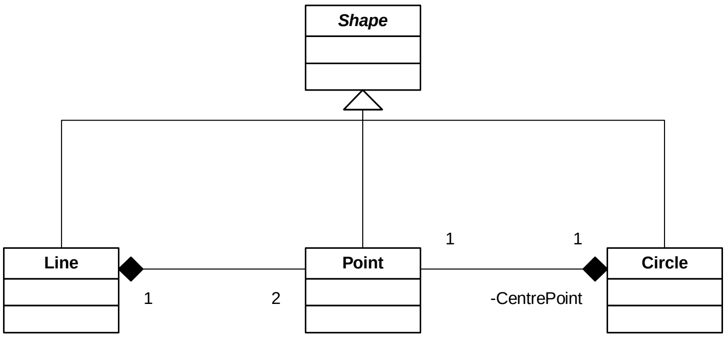

Figure 1: Shape Hierarchy

The `Shape` class has the following methods:

```
Shape();                              // Default constructor
Shape(const Shape& shp);              // Copy constructor
Shape& operator=(const Shape& shp);   // Assignment operator
```

The `Point` class has 'x' and 'y' private data-members and the following methods:

```
Point();                              // Default constructor
Point(double xs, double ys);          // Constructor with coordinates
Point(const Point& pt);               // Copy constructor
Point& operator=(const Point& pt);    // Assignment operator
double x();                           // Return x coordinate
double y();                           // Return y coordinate
void x(double xs);                    // Set x coordinate
void y(double ys);                    // Set y coordinate
```

The `Line` class has two private `Point` data-members for the start- and endpoint and the following methods:

```
Line();                                     // Default constructor
Line(const Point& pt1s, const Point& pt2s); // Constructor with points
Line (const Line & pt);                     // Copy constructor
Line & operator=(const Line & pt);          // Assignment operator
Point p1();                                 // Return point 1
Point p2();                                 // Return point 2
void p1(const Point& ps);                   // Set point 1
void p2(const Point& ps);                   // Set point 2
```

The `Circle` class has a private *Point* data-member for the centre-point and a double for the radius. Also the following methods are available:

```
Circle();                                   // Default constructor
Circle(const Point& c, double r);           // Constructor point and radius
Circle (const Circle & pt);                 // Copy constructor
Circle & operator=(const Circle & pt);      // Assignment operator
Point CenterPoint();                        // Return center point
double Radius();                            // Return radius
void CenterPoint(Point centre);             // Set center point
void Radius(double radius);                 // Set radius
```

The classes are created in such a way that deep copies are created from all the arguments.

1. Testing a Schape Hierarchy

Examine the shape hierarchy code. After that, create a program that tests the shape hierarchy. Create a print function for all shapes that will be removed later on.
In the test program you can use the overloaded ostream operators to display the shapes.

2. Delegation of line length calculation

In this exercise we add a method to the `Point` class to calculate the distance between two points. We also add a method to the `Line` class to calculate the length of the line. Because the `Line` class uses `Point` objects for its start- and endpoint, we delegate the line length calculation to the `Point` class.

a) In the `Point` class, create a method called `Distance()`. The method should accept a `Point` object as argument and returns a `double`.

```
double Distance(const Point& p);
```

b) The implementation of the `Distance()` method should use the Pythagoras algorithm to calculate the distance between the current point and the argument point. Use the `sqrt()` method to calculate the square root.

$$d = \sqrt{(x2-x1)^2 + (y2-y1)^2}.$$

c) Create another *`Distance`*`()` function in the Point class with no arguments. This function returns the distance to the origin (0,0).
d) In the *Line* class, create a method called `Length()`. The method has no arguments and should return a double.
e) We delegate the length calculation to the `Point` class thus in the `Length()` method we call the `Distance()` method on the startpoint object with the endpoint object as argument.
f) Create a test program that tests the `Distance()` and `Length()` methods.

2. Singleton Pattern

1. Origin point Singleton

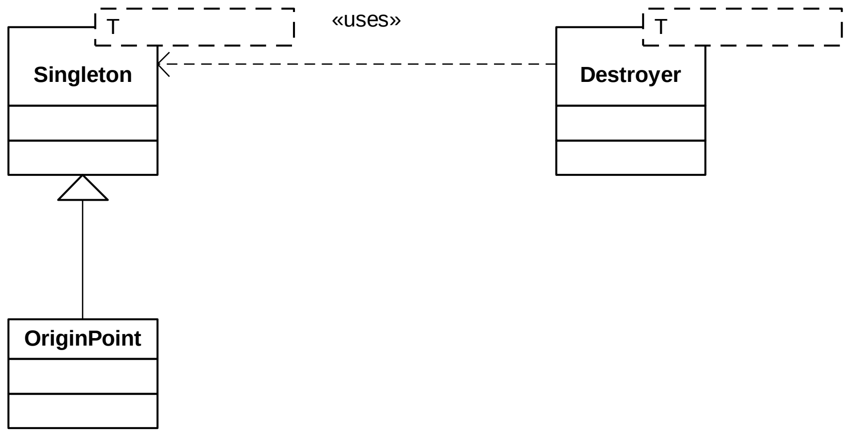

Figure 2: Origin point Singleton

In this exercise we derive an `OriginPoint` class from the singleton class with an instantiated Point object.

We use the code from the previous exercise.

a) Create the `OriginPoint` sub class that derives from the Singleton class.
b) The Singleton class is already created. The files are in the StartCode\Singleton directory
c) Change the `Distance()` function of the Point class to use the new `OriginPoint` class.
d) Write a test program to test the origin point. Calculate the distance between a point and the origin point, change the origin point and try the distance function again.

3. Composite Pattern

1. Shape Composite

In this exercise we extend the shape hierarchy with a shape composite class. The shape composite will be implemented using an STL list and can contain multiple shapes. Since the shape composite is a shape the composite can contain other composites. We start with the code from the "Singleton" exercise 2.
The UML diagram of the shape composite is shown in Figure 3.

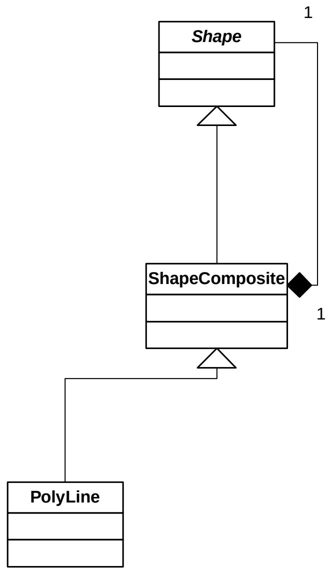

Figure 3: Shape Composite

a) Create a `ShapeComposite` class that derives from the `Shape` class.
b) The `ShapeComposite` has a private `std::list` data-member that contains `Shape` pointers.
c) Create the Copy constructor and Assignment operator but make them private. Do not implement them yet.
d) Create a public function called `AddShape(Shape* s)` that puts the given shape in the list.
e) Create a public typedef for an iterator and one for a const iterator. Use the iterators of the list class.
f) Create two functions `begin()` and `end()` to return the begin iterator or end iterator.
g) Create a function `count()` that returns the size of the shape composite (list).
h) Write a test program to test the `ShapeComposite`. Try to place shape composites in other shape composites. In that test program write a `Print(ShapeComposite sc)` function that recursively prints the composite.

4. Prototype

1. Changing the shape hierarchy

a) In this exercise we will use the Prototype pattern to define a method `Clone()` that all the derived classes should implement. A user can use this method to create a copy of a Shape without knowing the exact type.
b) Change the base class `Shape`. Add a Pure Virtual Member function called `Clone()`. It returns a `Shape*`.
c) Add the function to all shape classes.
d) Change the composite to use the new function to implement the copy constructor and assignment operator.
e) Create a program to test the newly added functionality. The easiest way would be to use the previous program and call a copy constructor for the composite.

5. Proxy Pattern

1. Access Proxy

In this exercise you create an `Account` class. The `Account` class will be used in a banking application. Normal users may withdraw an amount from an account but only supervisors may query the balance since that is sensitive information. Since we do not want to bloat the `Account` object with access permission code, we create a proxy class to control the access to the `GetBalance()` method of the `Account` class. An UML diagram of the account and proxy classes can be found in Figure 44.
In this exercise we use a simple static password to control access but in a real-life situation you can use a more elaborate security system.

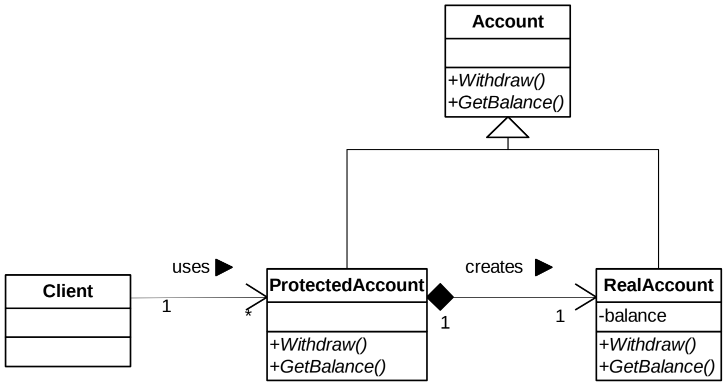

Figure 4: Access Account class via a Proxy

a) Create a simple class `Exception` with no members.
b) Create a `NoFundsException` class. Make it a subclass of `Exception`. We throw this exception when there are not enough funds to execute a withdraw.
c) Create a `NoAccessException` class. Make it a subclass of *Exception*. We throw this exception when the client has no permission to access the `GetBalance()` method.
d) Create an abstract base class called *Account*. Give this class a `Withdraw()` method that accepts a double as argument. This method can throw a `NoFundsException`.
e) Also give the class a `GetBalance()` method. This method has no arguments and returns a double.
f) Create the `RealAccount` class that implements the `Account` interface. The `RealAccount` class has a private variable for the balance. Give it a constructor with an initial balance as argument. Implement the `Withdraw()` and `GetBalance()` methods. The `Withdraw()` method must throw a `NoFundsException` when there is not enough balance to execute the withdraw.
g) Create the `ProtectedAccount` class. Create a constructor with an initial balance and a password string as arguments. In the constructor, create an *Account* object and store it in a private variable.
h) Implement the `Withdraw()` method. Delegate the implementation to the embedded `RealAccount` object.
i) Implement the `GetBalance()` method. In this method, check if the password is correct. If not, throw a `NoAccessException`. Do not forget to add this exception to the *throws list* of the method. If the password is correct, delegate the implementation to the embedded *Account* object.
j) Create a test program that tests the `Account` and `ProtectedAccount` class. Test it with a valid and invalid password. Use error handling to catch `NoFunds`- and `NoAccessExceptions`.

6. Strategy Pattern

1. Calculate point distance using a Strategy Pattern

The function `Point::distance()` for calculating the distance between two points is expensive because of the `sqrt()` function that is needed. Some applications do not need this high level of accuracy and thus a less accurate but more efficient algorithm is used. To this end, create two algorithms for calculating the distance between two points:

- Exact: Pythagoras formula (as before)
- Approximate: the taximan's formula

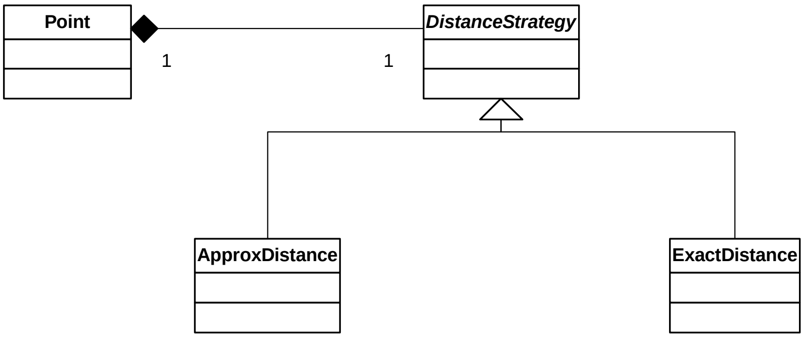

Figure 5: Distance Strategy

The taximan's formula is described as follows: calculate the sum of the absolute values of the difference between the individual x and y co-ordinates of the two points.

Create a test program. Experiment with different distance algorithms.

Modify the code so that the Strategy object is not part of the member data but is given as a parameter in the argument list. What are the advantages and disadvantages of this approach?

a) Create a new abstract class called `DistanceStrategy`. This class has an abstract method to calculate the distance between two points.

```
double Distance(const Point& p1, const Point& p2);
```

b) Create a new subclass of `DistanceStrategy` called `ExactDistance`. This class implements the distance method using an exact algorithm (Phytagoras):

$$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}.$$

c) Create another subclass of `DistanceStrategy` called `ApproximateDistance`. This class implements the distance method using a quicker but inaccurate algorithm (taxi driver):

$$d = Abs(x2 - x1) + Abs(y2 - y1).$$

d) Create a static data-member in the `Point` class that contains a `DistanceStrategy` object. Add a static method to set the `DistanceStrategy` object.

e) Replace the implementation in the `Distance()` method of the `Point` class to use the `DistanceStrategy` object.

f) Write a program that calculates the distance between two points using the different strategies.

2. Strategy object as method argument

In the previous exercise we stored the strategy in a static variable in the *Point* class. Another possibility is to pass the strategy to use as argument in the `Distance()` method. Thus add a `Distance()` method that accepts a *Point* object and a `DistanceStrategy` object. Calculate the distance using the passed strategy object. Adapt your test program so that it uses the new *Distance()* method.

Question: What are the advantages and disadvantages of each method?

7. Factory Patterns

1. Adding shape factories to the shape hierarchy

In this exercise we extend the shape hierarchy with some shape factories. The `ConsoleShapeFactory` creates shapes interactively. The UML diagram is shown in Figure 6.

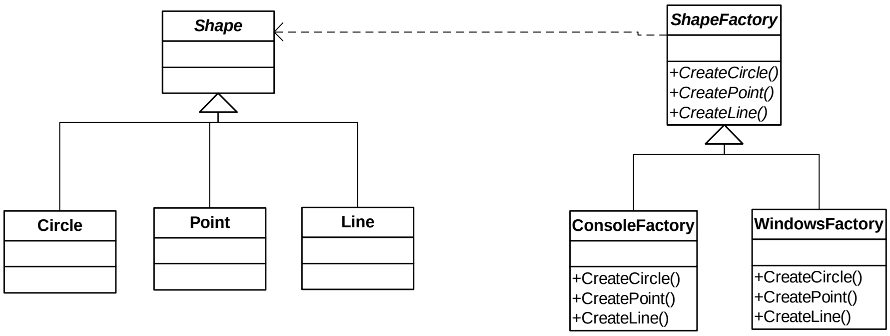

Figure 6: Shape Factory

a) Create a new abstract base class called `ShapeFactory`.
b) The `ShapeFactory` should have a pure virtual member function for each type of *Shape*. In this case we use the shapes *Circle*, *Line* and *Point*.
c) Create a new class called `ConsoleShapeFactory` that is derived from `ShapeFactory`.
d) Implement the `CreateShape()` functions. The methods should prompt the user for the necessary data needed to create a particular shape.
e) Create a test program that uses the base class `ShapeFactory` to create some shapes.

8. Decorator Pattern

1. Name decorator for the shape hierarchy

In this exercise we extend the shape hierarchy with a name decorator. This enables us to give *some* shape objects a name. The UML diagram of the decorator is shown in Figure 7.

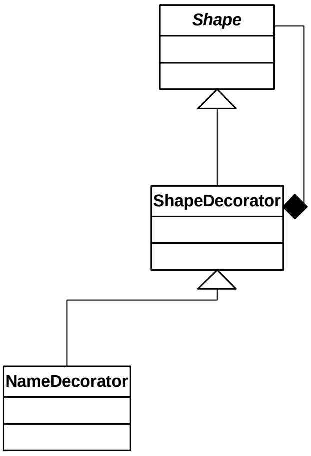

Figure 7: Shape decorator

a) Create a new abstract class `ShapeDecorator` that derives from *Shape*.
b) Add a private data-member to the `ShapeDecorator` class that holds the shape to decorate.
c) Add a default constructor that sets the shape data-member to 'null'.
d) Add a constructor that has a `Shape*` as argument. Assign the shape object to the shape data-member.
e) Add two public methods called `GetShape()` and `SetShape()` that return and set the decorated shape, respectivily.
f) Create a new class `NameDecorator` that derives from the `ShapeDecorator` class.
g) Add a private variable to the `NameDecorator` class that holds the name (String).
h) Add a default constructor that calls the base class constructor and sets the name to an empty string.
i) Add a constructor that has a `Shape*` and a std::`string` as argument. Call the base class constructor with the *Shape* as argument and set the name data-member.
j) Add two public methods called `GetName()` and `SetName()` that return and set the name.
k) Since the `NameDecorator` is a *Shape* we have to implement the `Clone()`. The `Clone()` method returns a copy of the `NameDecorator`.

9. State Pattern

1. Stack implementation using State Pattern

In this exercise we implement a stack using the state pattern. The stack has two methods namely the `Push()` method to push elements onto the stack and the *Pop()* method to pop elements from the stack.

A stack can be in three different states. The stack can be empty, the stack can be partially filled or the stack is full. A state transition diagram of the stack can be found in Figure 9. The UML diagram of the *Stack* is shown in Figure .

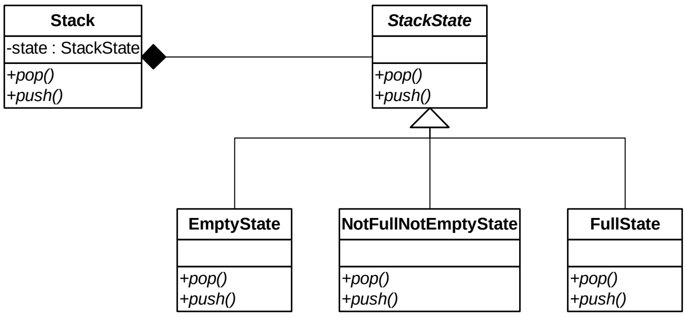

Figure 8: Stack using State Pattern

a) Create a new class called `Stack`.
b) The stack uses an array to store its elements. You may choose the kind of elements to store (e.g. int, double etc.). Declare the array as private data-member.
c) Declare a private data-member for the current index.
d) Declare a private data-member of the type `StackState` that holds the current state.
e) Create a private method called *Init(int size)*. This method has one argument that holds the size of the stack to be created. In this method, perform the following:
   - Check if the size is at least 1. If less set the size to 1.
   - Create the array for the elements with the given size.
   - Set the current index to 0 (first element to be set).
   - Set the state to a new instance of `EmptyState`.
f) Create a default constructor that calls the `Init()` method with size 1.
g) Create a constructor with a size argument. This constructor calls the `Init()` method with the given size.
h) Create a public `Push()` method that gets an "element" as argument. This method calls the `Push()` method on the current state object.
i) Create a public `Pop()` method that returns an "element". This method calls the `Pop()` method on the current state object.
j) Implement the different stack states. The states are implemented as member classes that can access the private members of the stack. Each state has a `Push()` and a `Pop()` method.
   Create the `StackState` abstract base class. Implement the `Push()` and *Pop()* methods so that they can be used by derived classes.
   - The `Push()` method has an "element" as argument. It stores the passed element in the array at the current index and then increases the current index.
   - The `Pop()` method returns an "element". The method should first decrease the current index and then return the element at the new current index.
k) Implement the `EmptyState` subclass:
   - The `Pop()` method should throw an exception since you cannot pop from an empty stack.
   - The `Push()` method should call the `Push()` method of the base class. Then it should change the current state. If the current index==array.length then it should set the current state to `FullState`. Else it should set the current state to `NotFullNotEmptyState`.

l) Implement the `NotFullNotEmptyState`.
   - The `Pop()` method changes the current state to `EmptyState` if the current index==1. Then it should call the `Pop()` method of the base class.
   - The `Push()` method should call the *push()* method of the base class. Then it should change the current state to `FullState` if the current index==array.length.

m) Finally implement the `FullState`.
   - The `Pop()` method should change the current state to *EmptyState* if the current index==1. Otherwise it should change to `NotFullNotEmptyState`. Then call the `Pop()` method of the base class.
   - The `Push()` method should throw an exception since we can't push onto a full stack.

n) The states should be singletons. For simplicity you can derive the states from the singleton template.

o) Write a test program to test the stack.

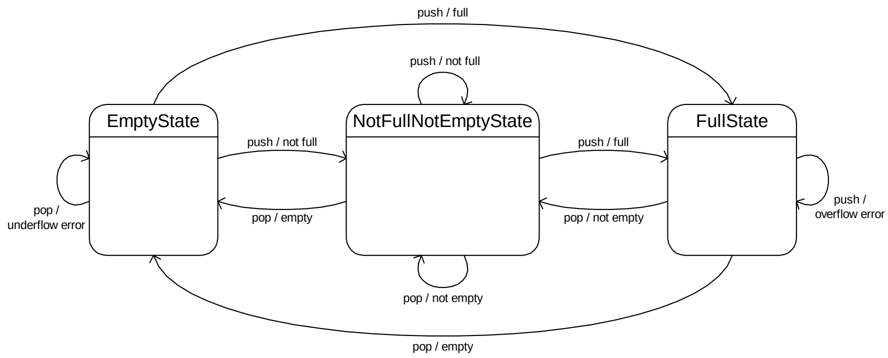

Figure 9: Stack State Transition Diagram

10. Visitor Pattern

1. Creating a Visitor for the Shape Hierarchy

Create functionality that allows us to translate any `Shape` in a given direction. In particular, it should be possible to translate any `Shape` by a given distance. For example, translating a Point $pt == (x; y)$ by a distance d results in the new point with co-ordinates:

- $(x + d; y + d)$

For the other classes the rules are:

- *Line*: translate each endpoint
- *Circle*: translate center point
- *Polyline*: translate all vertex points

If there is time…
Create a `Print` visitor that dumps the values in Shapes to screen (hint: use the << operator code)
The UML diagram of the visitor is shown in Figure 10.

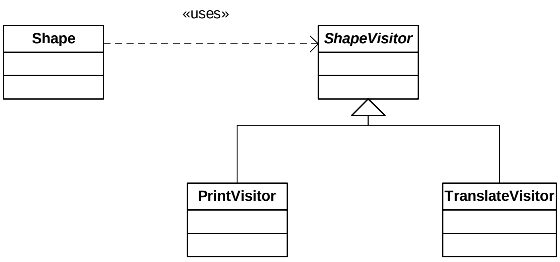

Figure 10: Shape visitor

a) Create a `ShapeVisitor` abstract base class. For every shape in the shape hierarchy. It has an abstract `Visit()` method with the appropriate shape as argument e.g. `void visit(Point& p);`
b) The `Visit()` method for the `ShapeComposite` can be implemented. In this method iterate in the composite and call the `Accept()` method on every shape in the composite. Pass the current visitor (*this) as argument.
c) Add a public abstract method to the `Shape` class and call it `Accept()`. This method has a `ShapeVisitor` object as argument.
d) Implement the `Accept()` method for each class. The method must call the `Visit()` method on the visitor with the current shape (*this) as argument.
e) Add a new class called `PrintVisitor` that extends the `ShapeVisitor` class. Implement all the `Visit()` methods. The `Visit()` methods print the shape information on the screen. The line *Visit* method can call the point visit method to print his embedded points. The shape composite *Visit* method can call the base class shape composite visit method to iterate the composite.
f) Write a test program to test the visitors. Try to put composites in other composites and confirm that also the contents of the composites are traversed correctly.

11. Observer Pattern

1. Counter Observer

In this exercise we create a counter class that plays the role of an observable. Thus we are separating the data from its view. The counter has two functions; one increases the current value and one decrements the current value. Many observers can then observe the counter. The observer displays the current counter value. The `LongFormat` observer displays the value using a no digit format and the `DoubleFormat` observer displays the value using a two digit precision format.
The UML diagram for the observer is shown in Figure 11.

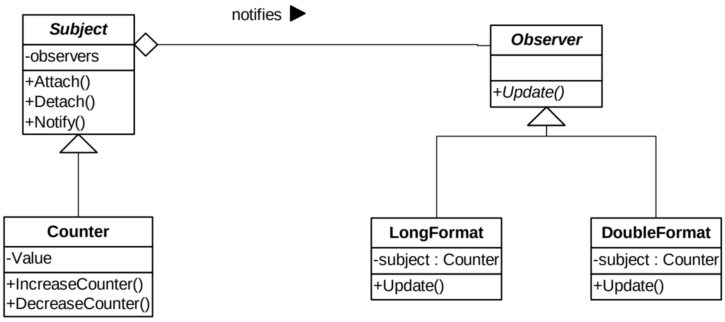

Figure 11: Counter observer

a) Create a class called `Subject`. Add the functions according to the above diagram. `Attach()` adds an observer to the list and a `Detach()` will remove it. The `Notify()` iterates in the list and call the `Update()` function in the observers.
b) Create a new class called `Counter` that derives from the `Subject` class.
c) The `Counter` class has a private variable that holds the counter value.
d) Create a `main()` method that creates a new `Counter` object.
e) Add a public method called `GetCounter()` that returns the current counter value.
f) Add a public method called `IncreaseCounter()`. This method increases the counter value, then it calls the `Notify()` methods of the `Observable` base class.
g) Add a public method called `DecreaseCounter()`. This method is similar to the `IncreaseCounter()` except it decreases the counter.
h) Add a default constructor to the `Counter` class. The default constructor initialises the counter.
i) Now we implement the `Observer` class.
j) Add a public method called `Update()`. This should have one argument namely an instance of `Subject`.
k) Create the two observer classes and implement the `Update()` method of the `Observer` class. The `Update()` method should get the current counter value from the `Observable` object (cast to `Counter`) and put it in the label.
l) Finally test the program. Add several observers and see that they all display the counter value.

12. Propagator Pattern

1. Turning Counter Observable in Propagator

In this exercise we turn the `Counter` observable of the previous exercise into a Propagator. We thus do not need the `CounterObserver` anymore. To achieve this we must move the functionality of the `CounterObserver` to the `Counter` class. The UML diagram of the propagator is shown in Figure 12.

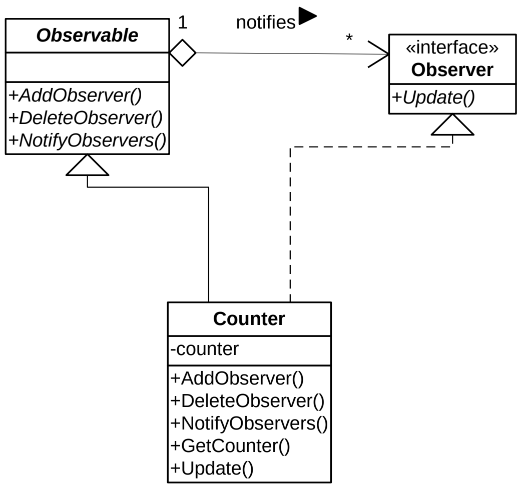

Figure 12: Counter propagator