Chapter-3

DESIGN INTERVIEW QUESTIONS

3.1 Glossary

- A *design pattern* is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Because design patterns consist of proven reusable architectural concepts, they are reliable and they speed up software development process. In simple words, there are a lot of common problems which a lot of developers have faced over time. These common problems ideally should have a common solution too. It is this solution when documented and used over and over becomes a design pattern.
- We cannot always apply the same solution to different problems. Design patterns would study the different problems at hand and suggest different design patterns to be used. However, the type of code to be written in that design pattern is solely the discretion of the Project Manager who is handling that project.
- Design patterns should present a higher abstraction level though it might include details of the solution. However, these details are lower abstractions and are called *strategies*. There may be more than one way to apply these strategies in implementing the patterns.
- To use design patterns in our application:
  1. We need to understand the problem at hand. Break it down to fine grained problems. Each design pattern is meant to solve certain kinds of problems. This would narrow down our search for design patterns.
  2. Read the problem statement again along with the solution which the design pattern will provide. This may instigate to change a few patterns that we are to use.
  3. Now figure out the interrelations between different patterns. Also decide what all patterns will remain stable in the application and what all need to change.
- *Refactoring*: Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Learning different design patterns is not sufficient to becoming a good designer. We have to understand these patterns and use them where they have more benefits. Using too many patterns (more than required) would be over-engineering and using less design patterns than required would be under-engineering. In both these scenarios we use refactoring. Refactoring is a change made to the internal structure of the software to make it easier to understand and cheaper to modify, without changing its observable behavior.
- Patterns: There are 23 design patterns [given by *Gang of Four*]. These patterns are grouped under three categories:
  - *Creational* Patterns: These patterns deal with object creation mechanisms. The basic form of object creation could result in design problems and adds complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation. Creational design patterns are further categorized into *Object*-creational patterns and *Class*-creational patterns, where Object-creational patterns deal with Object creation and Class-creational deal with Class-instantiation.
    - *Examples*: Factory, Abstract Factory, Builder, Prototype and Singleton patterns
  - *Structural* Patterns: Structural Patterns describe how objects and classes can be combined to form larger structures. The difference between class patterns and object patterns is that class patterns describe abstraction with the help of inheritance and how it can be used to provide more useful program interface. Object patterns, on other hand, describe how objects can be associated and composed to form larger, more complex structures.
    - *Examples*: Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy patterns
  - *Behavioral* Patterns: Behavioral patterns are those which are concerned with interactions between the objects. They identify common communication patterns between objects and realize these patterns.
    - *Examples*: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template and Visitor patterns

3.2 Tips

- Provide consistent and intuitive class interface: Clients which uses the class only need to know how to use the class and should not be forced to know how the functionality if provided.
- Provide common properties of classes in base class: Common properties of various classes should be identified and should be provided in a common base class (which may be an abstract, concrete or a pure interface class). Placing virtual methods in the base class interface is the basis of runtime polymorphism; it also avoids code duplication and unnecessary downcasts.
- Do not expose implementation details in the public interface of the class: If the class interface exposes any implementation details of the class, it violates the rules of abstraction. Also, any changes to internal details of the class can affect the interface of the class, which is not desirable.
- Consider providing helper classes while designing large classes.
- Keep the data members private.
- Provide lowest possible access to methods.
- Try to keep *loose* coupling between classes: Classes should be either independent of other classes or they should use only the public interface of other classes. Strive for such loose coupling between the classes because such classes are easy to understand, use, maintain and modify.
- While designing the classes beware of order of initialization as they are provided by the compiler or the implementation.
- Avoid calling virtual functions in constructors: Constructors do not support runtime polymorphism fully as the derived objects are still in construction when base class constructor executes. So, avoid calling virtual functions from base class constructors, which might result bugs in the code.
- Consider providing *factory* methods (refer *Problems* section).
- Make constructor *private* if there are *only* static members in the class: When there are only static members in the class, there is no need to instantiate it.
- Avoid creating useless temporary objects.
- Prefer virtual function over *RTTI* (RunTime Type Identification): Extensive use of RTTI can potentially make the code difficult to manage. Whenever you find cascading *if – else* statements for matching particular types to take actions, consider redesigning it by using virtual functions.
- Write unit test cases for classes.
- Overload functions only if the functions semantically do the same thing.
- Do not use inheritance when only values are different among classes.
- Hierarchically partition the namespace.
- Try eliminating all forms of code *duplication*.
- Use *design patterns* whenever *appropriate*.
- Know and make best use of the standard library.
- Provide catch handlers for specific exceptions before general exceptions.
- Forget about inheritance when you're modeling objects. It's just one way of implementing common code. When you're modeling objects just pretend you're looking at each object through an interface that describes what it can be asked to do. Don't make decisions about inheritance relationships based on names.
- Learn about test-driven development. Nobody gets their object model right up front but if we do test-driven development we're putting in the groundwork to make sure our object model does what it needs to and making it safe to refactor when things change later.
- Create loosely coupled classes: Often times when we give each object only one responsibility, the objects tightly couple together. This is a mistake, because it makes reuse harder later on.
- Practice as many problems as possible. This gives more experience and control in designing the better objects.

Problems and Questions with Answers

Note: For some design problems we have used *Java* classes to make the discussion simple. If you are not a *Java* professional, feel free to download the *C++* code online or try designing your own classes.

Problem-1 Explain Singleton design pattern.

Answer: There are only two points in the definition of a singleton design pattern,

- There should be only one instance allowed for a class and
- We should allow global point of access to that single instance.

The key is not the problem and definition. In singleton pattern, trickier part is implementation and management of that single instance.

Strategy for Singleton instance creation: We suppress the constructor (by making it *private*) and don't allow even a single instance for the class. But we declare an attribute for that same class inside, create instance for that and return it.

```
class Singleton {
  public:
     static Singleton* getInstance(){
          if (instance == 0) {
```

```
                instance = new Singleton();
            }
            return instance;
        }
    private:
        Singleton(){}
        static Singleton* instance;
};
```

We need to be careful with multiple threads. In a single-threaded environment, this works fine. If we don't synchronize the method which is going to return the instance, there is a possibility of allowing multiple instances in a multi-threaded scenario. Making the classic *Singleton* implementation thread safe is easy. Just acquire a lock before testing the instance.

```
class Singleton {
  public:
    static Singleton* getInstance(){
        Lock lock; // acquire lock (parameters omitted for simplicity)
            if (instance == 0) {
                    instance = new Singleton();
             }
             return instance;
     }
  private:
    Singleton(){}
     static Singleton* instance;
};
```

The problem with this solution is that it may be expensive. Each access to the Singleton requires acquisition of a lock, but in reality, we need a lock only when initializing *instance*. That should occur only the first time instance is called. If the *instance* is called *n* times during the course of a program run, we need the lock only for the first call. To solve this problem, we generally go for *double-checked locking* mechanism.

```
  class Singleton {
    public:
        static Singleton* getInstance(){
            if (instance == 0) {
                Lock lock; // acquire lock (parameterss omitted for simplicity)
                if (instance == 0) {
                    instance = new Singleton;
                }
            }
            return instance;
        }
    private:
        Singleton(){}
         static Singleton* instance;
  };
```

In the above example for singleton pattern, we can see that it is thread-safe.

Early and lazy instantiation in singleton pattern: The above example code is a sample for *lazy* instantiation for singleton design pattern. The single instance will be created at the time of first call of the *getInstance*() method. We can also implement the same singleton design pattern in a simpler way but that would instantiate the single instance early at the time of loading the class. Following example code describes how we can instantiate early. It also takes care of the multithreading scenario.

```
  class Singleton {
    static Singleton instance = new Singleton();
    Singleton() {}
    public:
      static Singleton getInstance() {
            return instance;
      }
  };
```

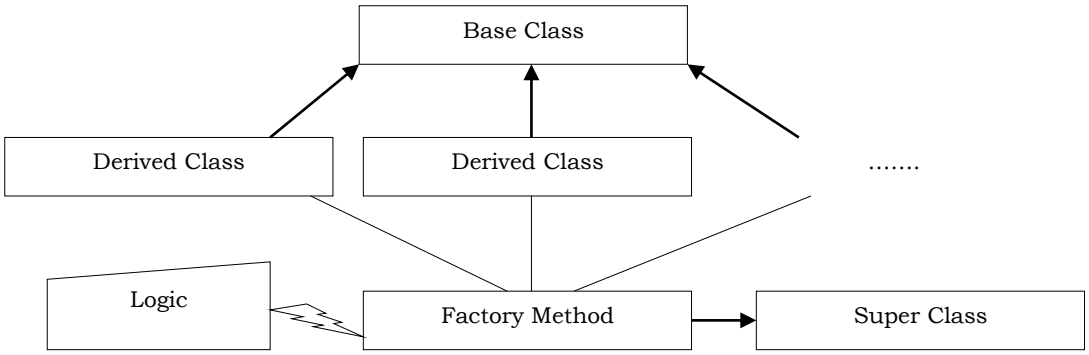
Problem-2 Explain *factory* method design pattern.

Answer: *Factory* method is just a fancy name for a method that instantiates objects. Like a factory, the job of the *factory method* is to create (or manufacture) objects. A *factory* method pattern is a *creational* pattern. It is used to instantiate an object from one among a set of classes based on some *logic*. *Factory* methods have many advantages over constructors. Depending on the situation, consider providing *factory* methods instead of constructors or in

addition to existing constructors. The usual approach to create new objects is by calling constructors. *Factory* methods provide alternative approach for this.

Assume that we have a set of classes which extends a common super class or interface. Now we will create a concrete class with a method which accepts one or more arguments. This method is our *factory* method. What it does is, based on the arguments passed *factory* method does logical operations and decides on which subclass to instantiate. This *factory* method will have the super class as its return type. *Factory* methods can not only return the object of the same type as its class, but also the objects of its derived class types.

Another point to note here is that, we cannot make constructors as *virtual* but *factory* methods can be declared as *virtual*. For the same reason, *factory* methods design patterns are also called as *virtual constructor*s. Now, let us implementation the factory method with a sample application.




```
//Base class that serves as type to be instantiated for factory method pattern
class Pet {
    public:
        virtual void petSound() = 0;
};
//Derived class 1 that might get instantiated
//by a factory method pattern
class Dog: public Pet{
  public:
    void petSound(){
        cout << "Bow Bow...\n";
    }
};
//Derived class 2 that might get instantiated
//by a factory method pattern
class Cat: public Pet{
  public:
    void petSound(){
        cout << "Meaw Meaw...\n";
    }
};
//Factory method pattern implementation that instantiates objects based on logic
class PetFactory {
  public:
    Pet* getPet(int petType) {
        Pet *pet = NULL;
        // Based on business logic factory instantiates an object
        if (petType == 1)
            pet = new Dog();
        else if (petType == 2)
            pet = new Cat();
        return pet;
    }
};
```

Now, let us create the factory method to instantiate

```
//using the factory method pattern
int main(){
    //creating the factory
    PetFactory *petFactory = new PetFactory();
    //factory instantiates an object
    Pet pet = petFactory→getPet(2);
    //you don't know which object factory created
    cout <<"Pet Sound: ";
    pet→petSound();
}
```

Problem-3 Explain prototype design pattern.

Answer: The prototype means making a clone. *Cloning* is the operation of replicating an object. The cloned object, the copy, is initialized with the current state of the object on which clone was invoked. *Cloning* an object is based on the concepts of *shallow* and *deep* copying.

- *Shallow* Copying: When the original object is changed, the new object is changed as well. This is due to the fact that the shallow copy makes copies of only the references, and not the objects to which they refer.
- *Deep* Copying:When the original object is modified, the new object remains unaffected, since the entire set of objects to which the original object refers to were copied as well.

This implies cloning of an object avoids creation. If the cost of creating a new object is large and creation is resource intensive, we clone the object. For example, we can consider construction of a home. *Home* is the final end product (object) that is to be returned as the output of the construction process. It will have many steps, like basement construction, wall construction, roof construction and so on. Finally the whole *home* object is returned. In *Java*, we use the interface *Cloneable* and call its method *clone*() to clone the object.

To implement the prototype design pattern, we just have to copy the existing instance in hand. Simple way is, clone the existing instance in hand and then make the required update to the cloned instance so that you will get the object you need. Always remember while using clone to copy, whether you need a shallow copy or deep copy and decide based on your business needs. Using clone to copy is entirely a design decision while implementing the prototype design pattern. *Clone* is not a mandatory choice for prototype pattern.

Implementation: We declare an abstract base class that specifies a pure virtual *clone*() method. The client code first invokes the factory method. This factory method, depending on the parameter, finds out concrete class. On this concrete class call to the *clone*() method is called and the object is returned by the factory method. In the below example *Quotation* class is an abstract class that has a pure virtual method *clone*(). *CarQuotation* and *BikeQuotation* are concrete implementation of a *Quotation* class.

```
class Quotation{
   protected:
      string type;
      int value;
   public:
      virtual Quotation* clone() = 0;
      string getType(){
         return type;
      }
      int getValue(){
         return value;
      }
};
```

```
class CarQuotation : public Quotation{
   public:
      CarQuotation(int number){
         type = "Car";
         value = number;
      }
      Quotation* clone(){
         return new CarQuotation(*this);
      }
};

class BikeQuotation : public Quotation{
   public:
      BikeQuotation(int number){
         type = "Bike";
         value = number;
      }
      Quotation* clone(){
         return new BikeQuotation(*this);
      }
};
```

QuotationFactory is a class that has *factory* method *CreateQuotation*(…). This method requires a parameter and depending on this parameter it returns the concrete implementation of *Quotation* class.

```
class QuotationFactory{
   private :
      Quotation *carQuotation;
      Quotation *bikeQuotation;
   public :
     QuotationFactory(){
         carQuotation  = new CarQuotation(10);
         bikeQuotation = new BikeQuotation(20);
     }
     ~QuotationFactory(){
       delete bikeQuotation;
       delete carQuotation;
     }
     Quotation* createQuotation(int typeID) {
       if(typeID == 1)
            return carQuotation→clone();
       else return bikeQuotation→clone();
     }
};
```

For the above discussion, a test code can be given as:

```
int main(){
 QuotationFactory* qf = new QuotationFactory();
```

```
    Quotation* q;
    q = qf→createQuotation(1);
    delete q;
    q = qf→createQuotation(2);
    delete q;
    delete qf;
    return 0;
}
```

Java has a built-in clone method in native code. This method is defined by the parent class, *Object*, and presents the following behaviour: *clone*() takes a block of memory from the *Java* heap. The block size is the same size as the original object. It then performs a bitwise copy of all fields from the original object into fields in the cloned object. This kind of operation is known as a *shallow* copy.

When copying involves primitive types, i.e. a byte, modifying either the original or the new copy will not modify the other object. This occurs because shallow copying of primitive types values result in separate values being stored in the cloned object. However, when a clone involves reference types, the pointer is copied. As a result, the original and the copy objects will contain pointers to the same object.

Following sample *Java* source code demonstrates the prototype pattern. We have a basic *Animal* in hand and to make a different object (clonedAnimal), we copy the existing instance. Then make necessary modifications to the copied instance.

```
    public Animal clone() {
        Animal clonedAnimal = NULL;
        try {
                clonedAnimal = (Animal) super.clone();
                clonedAnimal.setDescription(description);
                clonedAnimal.setNumberOfLegs(numberOfLegs);
                clonedAnimal.setName(name);
        } catch (CloneNotSupportedException e) {
                e.printStackTrace();
        }
        return clonedAnimal;
    }
```

In addition to the above, *C++* and *Java* have a similar behaviour regarding cloning objects. A clear difference is that *C++* does not provide any built-in clone method as *Java* does. Despite of this fact, *C++* is such a powerful language that with the use of a custom-maid copy constructor and pure virtual functions the same behavior can be accomplished.

Problem-4 Explain decorator design pattern.

Answer: Decorator design pattern is used to extend or modify the behavior of an *instance* at runtime. Inheritance can used to extend the abilities of a *class* (for all instances of class). Unlike inheritance, we can choose any single object of a class and modify its behavior leaving the other instances unmodified. In implementing the decorator pattern we construct a wrapper around an object by extending its behavior. The wrapper will do its job either before or after and delegate the call to the wrapped instance.

We start with an interface (pure virtual functions) which creates a blue print for the class which will have decorators. Then implement that interface with basic functionalities. Till now we have got an interface and an implementation concrete class. Now, create an abstract class that contains (aggregation relationship) an attribute type of the interface. The constructor of this class assigns the interface type instance to that attribute. This class is the decorator base class. Now we can extend this class and create as many concrete decorator classes we want. The concrete decorator class will add its own methods. Either after or before executing its own method the concrete decorator will call the base instance's method. Key to this decorator design pattern is the binding of method and the base instance happens at runtime based on the object passed as parameter to the constructor. Its dynamically customizes the behavior of that specific instance alone.

Following given example is an implementation of decorator design pattern. *House* is a classic example for decorator design pattern. We create a basic *House* and then add decorations like colors, lights etc. to it as we prefer. The added decorations change the look and feel of the basic house. We can add as many decorations as we want. This sample scenario is implemented below.

```
class House {
   public:
        virtual string makeHouse() = 0;
};
```

The above is an interface depicting a house. I have kept things as simple as possible so that the focus will be on understanding the design pattern. Following class is a concrete implementation of this interface. This is the base class on which the decorators will be added.

```
class SimpleHouse: public House {
```

```
    public:
        string makeHouse() {
            return "Base House";
        }
};
```

Following class is the decorator base class. It is the core of the decorator design pattern. It contains an attribute for the type of interface (*House*). Instance is assigned dynamically at the creation of decorator using its constructor. Once assigned that instance method will be invoked.

```
class HouseDecorator: public House {
    House *house;
    public:
        HouseDecorator(House *housePtr) {
        house = housePtr;
        }
        string makeHouse() {
            return house→makeHouse();
        }
};
```

Following two classes are similar. These are two decorators, concrete class implementing the abstract decorator. When the decorator is created the base instance is passed using the constructor and is assigned to the super class. In the makeHouse method we call the base method followed by its own method *addColors*(). This *addColors*() extends the behavior by adding its own steps.

```
class ColorHouseDecorator: public HouseDecorator {
  private:
  string addColors() {
      return " + Colors";
  }
  public:
  ColorHouseDecorator(House *housePtr): HouseDecorator(housePtr) {
  }
  string makeHouse() {
      return HouseDecorator::makeHouse() + addColors();
  }
};

class LightsDecorator: public HouseDecorator {
  private:
  string addLights() {
      return " + Lights";
  }
  public:
  LightsDecorator(House *housePtr) : HouseDecorator(housePtr) {
  }
  string makeHouse() {
      return HouseDecorator::makeHouse() + addLights();
  }
};
```

Execution of the decorator pattern: To test the pattern we can create a simple house and decorate that with colors and lights. We can use as many decorators in any order we want. This excellent flexibility and changing the behavior of an instance of our choice at runtime is the main advantage of the decorator design pattern.

```
int main() {
    House *house = new LightsDecorator(new ColorHouseDecorator(new SimpleHouse()));
    cout<<house→makeHouse();
}
```

Problem-5 What is the difference between inheritance and decorator pattern? Can't we add extra functionality with inheritance?

Answer: In the previous problem, to add colors and lights for decoration we can simply create a subclasses to *House*. If we wish to add colors and lights to all houses then it make sense create a subclass to *House*. Decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class, decorating can provide new behavior at run-time for individual objects.

The major point of the pattern is to enable run-time changes: we may not know how we want the house to look until the program is running, and this allows us to easily modify it. Granted, this can be done via the subclassing, but not as nicely.

Problem-6 Explain builder design pattern.

Answer: Builder pattern is a creational pattern used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

For example, we can consider construction of a car. *Car* is the final end product (object) that is to be returned as the output of the construction process. It will have many steps, like settting base, wheels, mirrors, lights, engine, roof, interior and so on. Finally, the whole car object is returned. We can build cars with different properties. In Java API, *StringBuffer* and *StringBuilder* are some examples of builder pattern. Following is the interface that will be returned as the product from the builder.

```
class CarPlan {
    public:
    virtual void setBase(string basement) = 0;
    virtual void setWheels(string structure) = 0;
    virtual void setEngine(string structure) = 0;
    virtual void setRoof(string structure) = 0;
    virtual void setMirrors(string roof) = 0;
    virtual void setLights(string roof) = 0;
    virtual void setInterior(string interior) = 0;
};
```

Concrete class for the above interface: The builder constructs an implementation for the following class.

```
class Car: public CarPlan {
   private:
   string base, wheels, engine, roof, mirrors, lights, interior;
    public:
   void setBase(string base) {
       this→base = base;
    }
    void setWheels(string wheels) {
       this→wheels = wheels;
    }
    void setEngine(string engine) {
       this→engine = engine;
    }
    void setRoof(string roof) {
       this→roof = roof;
    }
    void setMirrors(string mirrors) {
       this→mirrors = mirrors;
    }
    void setLights(string lights) {
       this→lights = lights;
    }
    void setInterior(string interior) {
       this→interior = interior;
    }
 };
```

Now, we will define the builder interface with multiple different implementation of this interface in order to facilitate, the same construction process to create different representations.

```
class CarBuilder {
    public:
    virtual void buildBase() = 0;
    virtual void buildWheels() = 0;
    virtual void bulidEngine() = 0;
    virtual void bulidRoof() = 0;
    virtual void bulidMirrors() = 0;
    virtual void bulidLights() = 0;
    virtual void buildInterior() = 0;
    virtual Car* getCar() = 0;
};
```

First implementation of a builder.

```
class LowPriceCarBuilder: public CarBuilder {
    private:
            Car *car;
    public:
```

Second implementation of a builder.

```
class HighEndCarBuilder: public CarBuilder {
        private:
                Car *car;
        public:
```

```
        LowPriceCarBuilder() {
            car = new Car();
        }
        void buildBase() {
            car→setBase ("Low priced base");
        }
        void buildWheels() {
            car→setWheels("Cheap Tyres");
        }
        void bulidEngine() {
            car→setEngine("Low Quality Engine");
        }
        void bulidRoof() {
            car→setRoof("No flexible roof");
        }
        void bulidMirrors(){
            car→setMirrors("Cheap Mirrors");
        }
        void bulidLights(){
            car→setLights("Cheap Lights");
        }
        void buildInterior(){
            car→setInterior("Cheap Iterior");
        }
        Car* getCar() {
            return this→car;
        }
};
```

```
        HighEndCarBuilder() {
            car = new Car();
        }
        void buildBase() {
            car→setBase("Quality base");
        }
        void buildWheels() {
            car→setWheels("Quality Tyres");
        }
        void bulidEngine() {
            car→setEngine("High-end Engine");
        }
        void bulidRoof() {
            car→setRoof("Flexible roof");
        }
        void bulidMirrors(){
            car→setMirrors("Quality Mirrors");
        }
        void bulidLights(){
            car→setLights("Quality Lights");
        }
        void buildInterior(){
            car→setInterior("High-end Iterior");
        }
        Car* getCar() {
            return this→car;
        }
};
```

Following class constructs the car and most importantly, this maintains the building sequence of object.

```
class MechanicalEngineer {
    private:
            CarBuilder *carBuilder;
    public:
    MechanicalEngineer(CarBuilder *carBuilder){
        this→carBuilder = carBuilder;
    }
    Car *getCar() {
        return carBuilder→getCar();
    }
    void buildCar() {
        carBuilder→buildBase();
        carBuilder→buildWheels();
        carBuilder→bulidEngine();
        carBuilder→bulidRoof();
        carBuilder→bulidMirrors();
        carBuilder→bulidLights();
        carBuilder→buildInterior();
    }
};
```

Testing the sample builder design pattern.

```
    int main() {
        CarBuilder *lowPriceCarBuilder = new LowPriceCarBuilder();
        MechanicalEngineer *engineer = new MechanicalEngineer(lowPriceCarBuilder);
        engineer→buildCar();
        Car *car = engineer→getCar();
        cout<<"Builder constructed: " + car;
    }
```

Problem-7 Explain abstract factory design pattern.

Answer: This pattern is one level of abstraction higher than factory pattern. This means that the abstract factory returns the factory of classes. Like Factory pattern (returns one of the several sub-classes), this returns such factory which later will return one of the sub-classes.

Let's understand this pattern with the help of an example. Suppose we need to get the specification of various parts of a car. The different parts of car are, say wheels, mirrors, engine and body. The different types of cars are BenQ, BMW, GeneralMotors and so on. So, here we have an abstract base class *Car*.

```
class Car {
  public:
        Parts *getWheels() = 0;
        Parts *getMirrors() = 0;
        Parts *getEngine() = 0;
        Parts *getBody() = 0;
};
```

This class, as you can see, has four methods all returning different parts of car. They all return an object called *Parts*. The specification of *Parts* will be different for different types of cars. Let's have a look at the class *Parts*.

```
class Parts {
  public:
        string specification;
        Parts(string specification) {
                this→specification = specification;
        }
        string getSpecification() {
                return specification;
        }
};
```

And now let's go to the sub-classes of *Car*. They are *BenQ*, *BMW* and *GeneralMotors*.

```
class BenQ: public Car {
  public:
        Parts *getWheels() {
              return new Parts("BenQ Wheels");
        }
        Parts *getMirrors() {
              return new Parts("BenQ Mirrors");
        }
        Parts *getEngine() {
              return new Parts("BenQ Engine");
        }
        Parts *getBody() {
              return new Parts("BenQ Body");
        }
};
```

```
class BMW: public Car {
    public:
        Parts *getWheels() {
            return new Parts("BMW Wheels");
        }
        Parts *getMirrors() {
            return new Parts("BMW Mirrors");
        }
        Parts *getEngine() {
            return new Parts("BMW Engine");
         }
         Parts *getBody() {
            return new Parts("BMW Body");
        }
};
```

```
class GeneralMotors: public Car {
    public:
        Parts *getWheels() {
                return new Parts("GeneralMotors Wheels");
        }
        Parts *getMirrors() {
                return new Parts("GeneralMotors Mirrors");
        }
        Parts *getEngine() {
                return new Parts("GeneralMotors Engine");
        }
        Parts *getBody() {
                return new Parts("GeneralMotors Body");
        }
};
```

Now let's have a look at the *Abstract* factory which returns a factory "Car". We call the class CarType.

```
class CarType {
  private:
        Car *car;
  public:
        void mainFunction() {
                CarType *type = new CarType();
                Car *car = type→getCar("BenQ");
                cout<<"Wheels: " << car→getWheels()→getSpecification();
                cout<<"Mirrors: " << car→getMirrors()→getSpecification();
                cout<<"Engine: " << car→getEngine()→getSpecification();
                cout<<"Body: " << car→getBody()→getSpecification();
        }
        Car *getCar(string carType) {
```

```
                if (carType.compare("BenQ"))
                car = new BenQ();
                else if(carType.compare("BMW"))
                car = new BMW();
                else if(carType.compare("GeneralMotors"))
                car = new GeneralMotors();
                return car;
        }
};
```

When to use Abstract Factory Pattern? One of the main advantages of *Abstract Factory Pattern* is that it isolates the concrete classes that are generated. The names of actual implementing classes are not needed to be known at the client side. Because of the isolation, we can change the implementation from one factory to another.

Problem-8 Difference between abstract factory & builder pattern?

Answer: Abstract factory may also be used to construct a complex object, and then what is the difference with builder pattern? In builder pattern emphasis is on 'step by step'. Builder pattern will have many number of small steps. Those steps will have small units of logic enclosed in it. There will also be a sequence involved. It will start from step 1 and will go on up to step n and the final step is returning the object. In these steps, every step will add some value in construction of the object. That is you can imagine that the object grows stage by stage. Builder will return the object in last step. But in abstract factory how complex the built object might be, it will not have step by step object construction.

Problem-9 Explain adapter design pattern.

Answer: Adapter pattern converts the existing interfaces to a new interface to achieve compatibility and reusability of the unrelated classes in one application. Adapter pattern is also known as Wrapper pattern. An adapter allows classes to work together that normally could not because of incompatible interfaces. The adapter is also responsible for transforming data into appropriate forms. When a client specifies its requirements in an interface, we can usually create a new class that implements the interface and subclasses an existing class. This approach creates a class adapter that translates a client's calls into calls to the existing class's methods.

Have you ever faced any problem with power sockets of 2 pin and 3 pin? If we have a 3-pin cable and a 2-pin power socket, then we use an intermediate expansion box and fit the 3-pin cable to it and attach the cable of expansion box to 2-pin power socket. In similar way this design pattern works.

There are two ways of implementing the *Adapter* Pattern, either use the Inheritance or use the composition. Let us do it with the approach of Inheritance. The first step in implementation is defining the *Cable* object which has the different specification ($3-Pin$) to that of Socket.

```
class Cable {
  private:
        string specification;
        string getInput() {
           return specification;
        }
  public:
        Cable(){
           specification = "3-Pin";
        }
};
```

At very minimal, we can define the Socket interface and a sample concrete adapter for it as:

```
        class Socket {
             public;
                virtual string & getOutput() = 0;
        };
```

```
class ExpansionAdapter: public Socket {
        Cable *cable;
         public:
                string & getOutput() {
                        cable = new Cable();
                        string & output = cable→getInput();
                        return output;
                }
};
```

Observe that, we are having two different methods for *Cable* (getInput()) and *Socket* (getOutput()). To test the pattern, we can simply create a *Socket* and use adapter class to convert the interface.

```
class Client {
  Socket *socket;
  public:
    void funtionTest() {
       socket = new ExpansionAdapter();
       socket→getOutput();
    }
```

```
};
```

Problem-10 Explain facade design pattern.

Answer: Let us consider a simple example for understanding the pattern. While walking past the road, we can only see this glass face of the building. We do not know anything about it, the wiring, the pipes and other complexities. The face hides all the complexities of the building and displays a good looking front. This is how facade pattern is used. It hides the complexities of the system and provides a simple interface to the client from where the client can access the system. In Java, the interface JDBC is an example of facade pattern. We as users or clients create connection using the "java.sql.Connection" interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.

Let's try to understand the facade pattern using a simple example (Graphical User Interface, GUI). Any typical GUI will have a title, menu items and content to be shown in the middle content area. To provide these features we can have different classes as shown below.

```
class GUIMenu{
    public:
        void drawMenuButtons() {}
};
class GUITitleBar{
    public:
        void showTitleBar(const string & caption) {}
};
```

```
class GUIContent{
    public:
        void showButtons() {}
        void showTextFields() {}
        void setDefaultValues() {}
};
```

Now, to create a simple GUI the users have to think about all these classes. To make it simple, what we can do is create another class which combines all these into a single interface (facade).

```
class MyGUI{
    private:
        GUIMenu* menu;
        GUITitleBar* titleBar;
        GUIContent* content;
    public:
        MyGUI(){
            menu = new GUIMenu();
            titleBar = new GUITitleBar();
            content = new GUIContent();
        }
        ~MyGUI()    {
            delete menu;
            delete titleBar;
            delete content;
    }
    void drawGUI()    {
        content→showButtons();
        content→showTextFields();
        content→setDefaultValues();
        menu→drawMenuButtons();
        titleBar→showTitleBar("Title of the GUI");
    }
};
```

To create a GUI, users can simply use the MyGUI class with one simple call as shown below.

```
int main(){
    MyGUI* facade = new MyGUI();
    facade→drawGUI();
    return 0;
}
```

In this way the implementation is left to the façade. The client is given just one interface and can access only that. This hides all the complexities. All in all, the *Façade* pattern hides the complexities of system from the client and provides a simpler interface. Looking from other side, the facade also provides the implementation to be changed without affecting the client code.

Problem-11 Explain visitor design pattern.

Answer: *Visitor* pattern is useful if we want to perform operations on a collection of objects of different types. One approach is to iterate through each element in the collection and then do something specific to each element, based on its class. If we just wanted to print out the elements in the collection, we could write a simple method like:

```
public void visitElements(vector<string> collection) {
    for(vector<string>::const_iterator iterator = collection.begin(); iterator!= collection.end(); ++ iterator)
```

```
        // print objects data
    }
```

If we don't know what type of objects are in the collection then that can get pretty tricky. The above approach will not work if the objects are from different classes. I mean, what if, for example, we have a vector of hashtables? In this case, we might need to check the type of object dynamically at runtime as shown below. For each of the type in the collections we need to put a separate conditional statement. If we keep on adding if-else conditions, then the code will lose its maintainability.

```
                    Class AbstractElement{
                        //some operations
                    };
class ConcreteFirstElement: public AbstractElement {
    public:
        void operationOne() {
            //operation on ConcreteFirstElement objects.
        }
};
```

```
class ConcreteSecondElement: public AbstractElement {
    public:
        void operationTwo() {
            //operation on ConcreteFirstElement objects.
        }
};
```

```
                    void visitElements(vector<AbstractElement*>& elements) const{
                        vector<AbstractElement*>& elems = elements.getElements();
                        for(vector<AbstractElement*>::iterator it = elems.begin(); it != elems.end(); ++it ){
                            if (typeid(*it).name()  == "ConcreteFirstElement"){
                                //Some processing for type1
                            }
                            else if (typeid(*it).name()  == "ConcreteSecondElement"){
                                //Some different processing for type-2
                            }
                        }
                    }
```

The purpose of the *Visitor* pattern is to encapsulate an operation that we want to perform on the elements of a data structure. In this way, we can change the operation being performed on a structure without the need of changing the classes of the elements that we are operating on. Using a *Visitor* pattern allows us to decouple the classes for the data structure and the algorithms used upon them.

Each node in the data structure *accepts* a visitor, which sends a message to the visitor which includes the node's class. The visitor will then execute its algorithm for that element. This process is known as *double dispatching*. The node makes a call to the visitor, passing itself in, and the visitor executes its algorithm on the node. In double dispatching, the call made depends upon the type of the visitor and of the host (data structure node), not just of one component.

Now, let us concentrate on implementing *Visitor* pattern. Let us consider a very simple abstract data structure, *AbstractElement* and its two possible concrete classes, *ConcreteFirstElement* and *ConcreteSecondElement*. The data structure provides functions that any processing algorithm might use to manipulate the data. Note that these operations, represented here simply as the *operationOne*() and *operationTwo*() methods, are the atomic operations for manipulating the data upon which all more complex operations are built.

```
                    Class AbstractElement{
                        virtual void accept (AbstractVisitor& v) = 0;
                    };
class ConcreteFirstElement: public AbstractElement {
    public:
        void accept(AbstractVisitor& v) {
            v.visitConcreteFirstElement(*this);
        }
        void operationOne() {
            // operation on ConcreteFirstElement
            // objects.
        }
};
```

```
class ConcreteSecondElement: public AbstractElement {
    public:
        void accept(AbstractVisitor& v) {
            v.visitConcreteSecondElement(*this);
        }
        void operationTwo() {
            // operation on ConcreteFirstElement
            // objects
        }
};
```

The algorithms (*Visitor's*) on the data are represented by the abstract *AbstractVisitor* interface and its two possible algorithms *ConcreteFirstVisitor* and *ConcreteSecondVisitor*. In many traditional data processing algorithm architectures, *dumb* data is passed to a processing algorithm, which determines the specific type of the data and then processes it accordingly. In the *Visitor* design pattern, the data is given the ability to determine how it is processed. Instead of requiring the algorithm to query the data object to dynamically determine its type, the visitor pattern recognizes that each data type, *ConcreteFirstElement* and *ConcreteSecondElement* here, already intrinsically knows its own type.

The key is the *accept*() method in the ConcreteElement classes. The body of this method shows the double dispatching call, where the visitor is passed in to the accept method, and that visitor is told to execute its visit method, and is handed the element by the element itself. This makes for very robust code, since all of the decision making as to what to execute where and when it taken care of by the dispatching. Nobody ever needs to check anything: they just do what it is that they do, with whatever they're handed.

```
class AbstractVisitor{
  virtual void visit(ConcreteFirstElement& first) const = 0;
  virtual void visit(ConcreteSecondElement& second) const = 0;
  virtual ~AbstractVisitor() {}
};
class ConcreteFirstVisitor: public AbstractVisitor{
  public:
    void visit(ConcreteFirstElement& first) const {
    }
    void visit(ConcreteSecondElement& second) const{
    }
};
class ConcreteSecondVisitor: public AbstractVisitor{
  public:
    void visit(ConcreteFirstElement& first) const {
    }
    void visit(ConcreteSecondElement& second) const{
    }
};
```

The visitor simply provides methods for processing each respective data type and lets the data object determine which method to call. Since the data object intrinsically knows its own type, the determination of which method on the visitor algorithm to call is trivial. Thus the overall processing of the data involves a dispatch through the data object and then a subsequent dispatch into the appropriate processing method of the visitor. This is called *double dispatching*.

One key advantage of using the Visitor Pattern is that adding new operations to perform upon our data structure is very easy. All we have to do is create a new visitor and define the operation there. The problem with the visitor pattern is that because the each visitor is required to have a method to service every possible concrete data, the number and type of concrete classes cannot be easily altered once the visitor pattern is implemented.

Problem-12 Explain iterator design pattern.

Answer: The *Iterator* pattern is one of the most simple and frequently used design patterns. The iterator pattern is a behavioral object design pattern. The iterator pattern allows the traversal through the elements in a group of objects. The Iterator Pattern is very useful for allowing iteration through data structures without having to worry about how the data structure is stored. In *C++* there are iterator types associated with each container. Using these types is fine if we are iterating inside the class which defined the data structure. However, clients of class should not be exposed to the underlying data structure. For example, if a class returned a list of names, stored as a vector, the client of that class might iterate through the vector like this:

```
vector<string> names = myClassObj.getNames();
for(vector<string>::iterator itr = names.begin(); itr != names.end(); ++itr){
    // do something
}
```

In this case, the client of class now depends on the underlying data structure of *items*. If the data structure should ever change, the client must also be changed. The *Iterator* pattern makes the client *loosely* coupled to the *myClassObj*, and the code for iterating through a list of names might look like this:

```
MyClass::NameIterator nameItr = myClassObj.getNameIterator();
while(nameItr.hasNext()){
    // do something
}
```

An alternative to the Iterator pattern we are about to see would be to typedef the iterator type in MyClass like this:

```
class MyClass{
    // ...
    typedef vector<string>::iterator NameIterator;
};
```

However, this has the disadvantage of relying on the data structure to either be part of the standard library containers or conform to one of the standard library iterator interfaces. You could not, for example, create an iterator over a raw array or a *vector<bool>*.

We can write our own iterator by implementing methods like *hasNext*(), *next*(), and so on.

```
template <typename T>
class Iterator{
  public:
        typedef T value_type;
        virtual bool hasNext() const = 0;
        virtual T next() = 0;
};
```

Our interface is parameterized by T, which is the value type of the iterator (that is, it is the type we want to return from next, which is also the underlying data type of the container). There's nothing too special about this. Just note that next increments the iterator when calling it.

When writing an iterator for a class, it is very common for the iterator class to be an inner class of the class that we would like to iterate through. Here is example *NameManager* class. It has a list of names of type $vector :: string$. Names can be added via the *addName*() method. The *getNameIterator*() method returns an iterator of names. The *NameIterator* class is an *inner* class of *NameManager* that implements the *Iterator* interface for names objects. It contains basic implementations of the *hasNext*() and *next*() methods.

```
class NameManager {
    typedef std::vector<std::string> NameCollection;
    NameCollection m_names;
    public:
    class NameIterator: public Iterator< NameCollection::value_type >{
            // only NameManager should be allowed access to the constructor
            friend class NameManager;
    private:
            NameManager::NameCollection & m_names;
            NameManager::NameCollection::iterator m_itr;
            NameIterator(NameManager::NameCollection & names) :
            m_names(names), m_itr(m_names.begin()) {}
    public:
            virtual bool hasNext(void){
                    return m_itr != m_names.end();
            }
            virtual NameIterator::value_type next(void){
                    NameIterator::value_type value = (*m_itr);
                    ++m_itr;
                    return value;
            }
    };
    void addName(NameCollection::value_type name){
            m_names.push_back(name);
    }
    NameIterator getNameIterator(void){
            return NameIterator(m_names);
    }
};
```

The *below* code demonstrates the iterator pattern. It creates three names and adds them to the *nameMgr* object. Next, it gets an name iterator from the *nameMgr* object and iterates over them.

```
int main(void) {
    NameManager nameMgr;
    nameMgr.addName("Jobs");
    nameMgr.addName("Bill");
    nameMgr.addName("Larry");

    NameManager::NameIterator nameItr = nameMgr.getNameIterator();

    while(nameItr.hasNext()){
            cout << nameItr.next() << std::endl;
    }
}
```

Problem-13 Explain command design pattern.

Answer: *Command* pattern is an object behavioral pattern. This allows us to achieve complete decoupling between the sender and the receiver. A *sender* is an object that invokes an operation, and a *receiver* is an object that receives the request to execute a certain operation. With decoupling, the sender has no knowledge of the receiver's interface. The term *request* refers to the command that is to be executed.

This pattern is different from the *Chain of Responsibility* in a way that, in the earlier one, the request passes through each of the classes (in the hierarchy) before finding an object that can take the responsibility. The command pattern however finds the particular object according to the command and invokes only that one.

A classic example of this pattern is *switch*. In this example we configure the switch with 2 commands: to turn the air conditioner *on* and to turn off the air conditioner. Let's have a look at this example with $C++$ code. Below code shows the *receiver* class.

```
/*Receiver class*/
class AirConditioner {
```

```
 public:
    AirConditioner() { }
     void start() {
          cout << "The Air Conditioner is on" << endl;
    }
     void stop() {
          cout << "The Air Conditioner is off" << endl;
    }
};
```

The *Command* pattern convert the request itself into an object. This object can be stored and passed like other objects. The key to command pattern is a *Command* interface. Command interface declares an interface for executing operations.

```
/*the Command interface*/
class Command {
    public:
    virtual void execute()=0;
};
```

A benefit of this particular implementation of the command pattern is that the switch can be used with any device, not just an air conditioner - the *Switch* in the following example turns a air conditioner on and off, but the *Switch's* constructor is able to accept any subclasses of *Command*(acts as *invoker*) for its 2 parameters. For example, we could configure the *Switch* to start a *motor*.

Our objective is to develop a *Switch* that can turn either object on or off. We see that the *AirConditioner* have different interfaces, which means the *Switch* has to be independent of the *receiver* interface. To solve this problem, we need to parameterize each of the *Switchs* with the appropriate command. When the *start*() and *stop*() operations are called, they will simply make the appropriate command to *execute*( ). The *Switch* will have no idea what happens as a result of *execute*( ) being called. *Switch* is called the invoker because it invokes the execute operation in the command interface.

```
class Switch {
public:
    Switch(Command& startCmd, Command& stopCmd)
                :startCommand(startCmd), stopCommand(stopCmd){
    }
     void start(){
          startCommand.execute();
    }
     void stop(){
          stopCommand.execute();
    }
 private:
    Command& startCommand;
    Command& stopCommand;
};
```

Each concrete *Command* class specifies a receiver-action by storing the receiver (*AirConditioner*) as an instance variable. It provides different implementations of the *execute*() method to invoke the request. The receiver has the knowledge required to carry out the request.

The concrete commands, *StartCommand* and *StopCommand*, implements the execute operation of the command interface. It has the knowledge to call the appropriate receiver object's operation. It acts as an adapter in this case. By the term adapter, I mean that the concrete *Command* object is a simple connector, connecting the *invoker* and the *receiver* with different interfaces.

```
/*the Command for turning on the Air Conditioner*/
class StartCommand: public Command {
   public:
     StartCommand(AirConditioner& airConditioner):
                    theAirConditioner(airConditioner) {
     }
     virtual void execute(){
          theAirConditioner.start();
    }
    private:
    AirConditioner& theAirConditioner;
};
/*the Command for turning off the Air Conditioner*/
class StopCommand: public Command{
   public:
    StopCommand(AirConditioner& airConditioner)
```

```
                    :theAirConditioner(airConditioner){
    }
    virtual void execute() {
        theAirConditioner.stop();
    }
    private:
     AirConditioner& theAirConditioner;
};
/*The test class or client*/
int main() {
    AirConditioner ac;
    StartCommand switchUp(ac);
    StopCommand switchDown(ac);

    Switch s(switchUp, switchDown);
    s.start();
    s.stop();
}
```

Notice in the code example above that the *Command* pattern completely decouples the object that invokes the operation (*Switch*) from the ones having the knowledge to perform it (*AirConditioner*).

Problem-14 Explain memento design pattern.

Answer:

```
class Memento {
  private:
      string state;
  public:
      Memento(string& stateToSave) {
          state = stateToSave;
      }
      string& getSavedState() {
          return state;
      }
};
class Originator {
  private:
      string state;
  public:
      void setState(string& state) {
          cout << "Setting state to " << state;
          this→state = state;
      }
      Memento* saveToMemento() {
          cout<<"Saving to Memento.";
          return new Memento(state);
      }
      void restoreFromMemento(Memento* m) {
          state = m→getSavedState();
          cout<<"Restoring state
                  from Memento:"<<state;
      }
};
class Caretaker {
  private:
      list<Memento*> savedStates;
  public:
      void addMemento(Memento* m) {
          savedStates.push_back(m);
      }
      Memento* getMemento() {
          return savedStates.back();
      }
};
int main() {
    Caretaker *caretaker = new Caretaker();
    Originator *originator = new Originator();
    Originator→setState("State1");
    originator→setState("State2");
    caretaker→addMemento(
            originator→saveToMemento() );
    originator→setState("State3");
    caretaker→addMemento(
        originator→saveToMemento() );
        originator→restoreFromMemento(
                caretaker→getMemento());
}
```

We all use this pattern at least once every day. *Memento* pattern provides an ability to restore an object to its previous state (undo via rollback). *Memento* pattern is used by two objects: originator and a caretaker. The originator is some object that has an internal state. *Caretaker* is going to perform some action to the originator, but wants to be able to undo the change.

- Identify a class that needs to be able to take a snapshot of its state (originator role.)
- Design a class that does nothing more than accept and return this snapshot (memento role).
- *Caretaker* role asks the *Originator* to return a Memento and cause the *Originator's* previous state to be restored of desired.
- Notion of undo or rollback has now been objectified (i.e. promoted to full object status).

Client requests a *Memento* from the source object when it needs to checkpoint the source object's state. Source object initializes the *Memento* with its state. The client is the care taker of the *Memento*, but only the source object can store and retrieve information from the *Memento*. If the client subsequently needs to "rollback" the source object's state, it hands the *Memento* back to the source object for reinstatement.

An unlimited "undo" and "redo" capability can be readily implemented with a stack of command objects and a stack of *Memento* objects.

Problem-15 Explain strategy design pattern.

Answer: *Strategy* pattern is an example for behavioral pattern and is used when we want different algorithms needs to be applied on values (objects). That means, *Strategy* design pattern defines a set of algorithms and make them interchangeable. *Strategy* lets the algorithm vary independently from clients that use it. The strategy pattern is also known as the *Policy* pattern. We can apply *Strategy* pattern when we need different variants of an algorithm (each algorithm can be assumed as a separate class) and these related classes differ only in their behavior.

Let us consider a real-time example for understanding the pattern. Assume we have a *vector or* an *array* container. To sort the items in that list, we can use bubble sort, quick sort, heap sort etc... But we can use only one algorithm at a time out of all the possible. As a first step we need to define the *Strategy*(algorithms) and *Context*(vector *or* array container) interfaces. Context pass data to Strategy. Strategy has point to *Context* to get data from *Context*. Strategies can be used as template parameters (*Template* design pattern) if strategy can be selected at compile-time and does not change at run-time.

```
class SortInterface {
    public:
        virtual void sort(int[], int) = 0;
};
```

The concrete implementations of the SortInterface can be given as:

```
class QuickSort: public SortInterface {
    public:
        void sort(int array[], int size) {
            //Quick sort logic
        }
};
```

```
class BubbleSort: public SortInterface {
    public:
        void sort(int array[], int size) {
            //Bubble sort logic
        }
};
```

The dependent abstract *Context* class and its dependent concrete class can be given as:

```
class Sorter {
  private:
    SortInterface* strategy;
  public:
    void setSorter(SortInterface* strategy) {
        this→strategy = strategy;
    }
    SortInterface* getSorter() {
        return this→strategy;
    }
    virtual void doSort(int listToSort[], int size) = 0;
};
```

```
class MySorter: public Sorter {
    public:
        void doSort(int listToSort[], int size) {
            getSorter()→sort(listToSort, size);
            // other processing here
        }
};
```

Sample Client Code:

```
int main() {
    int listToBeSoted[] = {18, 26, 26, 12, 127, 47, 62, 82, 3, 236, 84, 5};
    int listSize = sizeof(listToBeSoted)/sizeof(int);
    MySorter *mysorter = new MySorter();
    Mysorter→setSorter(new BubbleSort());
    mysorter→doSort(listToBeSoted, listSize);
    mysorter→setSorter(new QuickSort());
    mysorter→doSort(listToBeSoted, listSize);
}
```

Problem-16 Explain state design pattern.

Answer: *State* pattern allows objects to behave in different ways depending on internal state (*Context*). The *Context* can have a number of internal states, whenever the *request*() method is called on the *Context*, the message is delegated to the *State* to handle. The *State* interface defines a common interface for all concrete states, encapsulating all behaviour associated with a particular state. The concrete state provides its own implementation for the request. When a Context changes state, what really happens is that we have a different *ConcreteState* associated with it.

This is all quite similar to the *Strategy* pattern, except the changes happen at runtime rather than the client deciding. *State* saves us from lots of conditional code in *Context*: by changing the *ConcreteState*object used, we can change the behavior of the context. Let us consider a real-time example for understanding the pattern (Music player). Similar to *Strategy* pattern here also we have *Context* and a *State* interface.

```
class State {
    public:
        virtual void pressPlay(MusicPlayerContextInterface* context) = 0;
```

```
};
```

The concrete implementations of the *State* interface can be given as:

```
class StandbyState: public State{
        public:
                void pressPlay(
                    MusicPlayerContextInterface *context){
                  context→setState(new PlayingState());
                }
};
class PlayingState: public State {
        public:
                void pressPlay(
                    MusicPlayerContextInterface *context){
                    context→setState(new StandbyState());
                }
};
```

A dependent abstract *Context* class and its dependent concrete class can be given as:

```
//Context Interface
class MusicPlayerContextInterface{
      State *state;
      public:
          virtual void requestPlay() = 0;
          virtual void setState(State *state) = 0;
          virtual State* getState() = 0;
};
//Sample Test Code
int main(){
   MusicPlayerContext* musicPlayer = new
                MusicPlayerContext(new StandbyState());
     musicPlayer→requestPlay();
     musicPlayer→requestPlay();
     musicPlayer→requestPlay();
     musicPlayer→requestPlay();
     return 0;
}
```

```
//Concrete Context
class MusicPlayerContext: public
                        MusicPlayerContextInterface{
        State *state;
        public:
            MusicPlayerContext(State *state){
              this→state= state;
            }
            void requestPlay(){
               state→pressPlay(this);
            }
            void setState(State *state){
               this→state = state;
            }
            State* getState(){
               return state;
            }
};
```

This shows how the state pattern works at a simple level. Few advantages of *State* pattern are:

- State pattern provides a clear state representation of an object
- It allows a clean way for an object to partially change its type at runtime

Problem-17 Explain observer design pattern.

Answer: Have you ever used RSS feeds? If so, you already know about *Observer* pattern. The *Observer* pattern defines a link between objects so that when one object's state changes, all dependent objects are updated automatically (Producer/Consumer problem which we coded in *Java* is also a classic example). There is always an *Observer* (also called *Listeners*) and *Observable* (also called *Providers*) object around us. We are an *Observer*, TV is an *Observable* object. That means, *Observer* pattern is designed to help cope with one to many relationships between objects, allowing changes in an object to update many associated objects. This is a behavioral pattern.

To implement *Observer* pattern, we define the interfaces for *Listener* and *Observer*. Note that, *Listener* has methods *addObserver* and *removeObserver* to keep track of observers.

```
class Listener {
  public:
    virtual void addObserver(Observer* o)=0;
    virtual void removeObserver(Observer* o) =0;
    virtual string& getState()=0;
    virtual void setState(string state) = 0;
    virtual void notifyObservers() = 0;
};
```

```
class Observer {
     public:
        virtual void update(Listener* l)=0;
};
```

For simplicity, let us assume that the changes are notified as strings. To support multiple observers we used *list*. One possible implementation of above interfaces can be given as: