Agile Design Principles

Agile programming is based on developing the software product in small incremental blocks, when the client"s requests and the solutions offered by the programmer progress simultaneously. Agile programming is based on a close relation between the final quality of the product and the frequent deliveries of incrementally developed functionalities. The more deliveries are carried out, the higher the quality of the final product.

In an agile implementation process, the modification requests are regarded as positive, no matter the development stage of the project. This is due to the fact that the modification requirements prove the fact that the team has understood what is necessary for the software product to comply with the necessities of the market.

For this reason, it is necessary for an agile team to maintain the code structure as flexible as possible, so that the new requirements of the clients have the smallest impact possible on the existing architecture. However, this doesn"t mean that the team will make an extra effort to take into consideration the future requirements and necessities of the clients, nor that it will spend more time to implement an infrastructure which might support possible requirements necessary in the future. Instead, this means that they will focus on developing the current product as well as possible.

With this purpose in view, we shall investigate some of the software design principles necessary to be applied by an agile programmer from one iteration to another, in order to maintain the project"s code and design as clean and flexible as possible. These principles were suggested by Robert Martin in his book called "Agile Software Development: Principles, Patterns and Practices (1)".

Single Responsibility Principle: SRP

A class should have only one reason to change.

In the SRP context, responsibility can be defined as "a reason to change". When the requirements of the project modify, the modifications will be visible through the alteration of the responsibilities of the classes. If a class has several responsibilities, then, it will have more reasons to change. Having more coupled responsibilities, the modifications on a responsibility will imply modifications on the other responsibilities of the class. This correlation leads to a fragile design.

Fragility means that a modification of the system leads to a break in design, in places that have no conceptual connection to the part which has been modified.

Example:

Suppose we have a class which encapsulates the concept of phone and the associated functionalities.

```
class Phone
{
   public void Dial(const string&
   phoneNumber);
```

```
  public void Hangup();
 public void Send(const string&
  message);

 public  Receive(const string&
  message);
 };
```

This class might be considered reasonable. All the four methods defined represent functionalities related to the phone concept. However, this class has two responsibilities. The methods Dial and Hang-up are responsible for performing the connection, while the methods send and Receive are responsible for data transmission.

In the case where the signature of the methods responsible for performing the connection would be subjected to changes, this design would be rigid, since all the classes which call the Dial and Hangup methods would have to be recompiled. In order to avoid this situation, a re-design is necessary, to divide the two responsibilities.

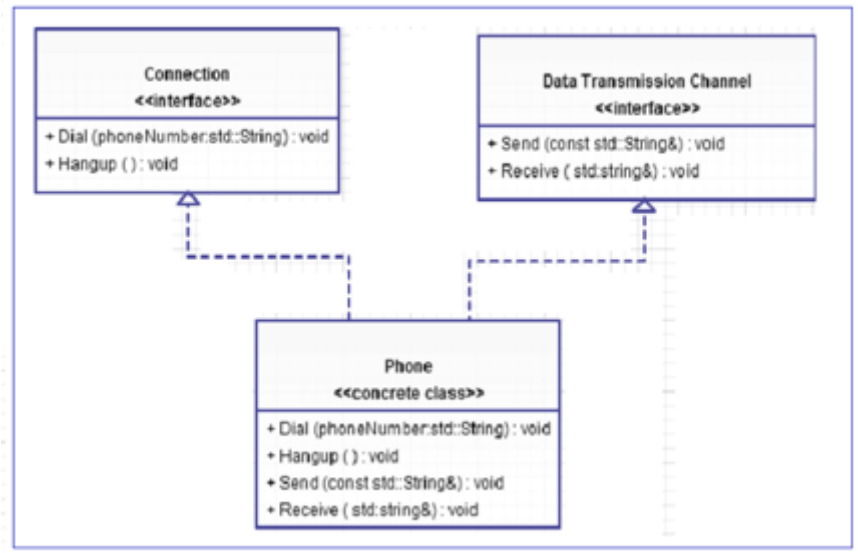

Figure 1

In this example, the two responsibilities are separated, so that the class that uses them - Phone, does not have to couple the two of them. The changes of the connection will not affect the methods responsible with data transmission. On the other hand, in the case where the two responsibilities do not show reasons for modification in time, their separation is not necessary either. In other words, the responsibilities of a class should be separated only if there are real chances that the responsibilities would produce modifications, mutually influencing each other.

Conclusion

The Single-Responsibility Principle is one of the simplest of the principles but one of the most difficult

to get right. Finding and separating those responsibilities is much of what software design is really about. In the rest of the principles of agile software design we will analyse further on, we will come back to this issue in one way or another.

Open Closed Principle: OCP

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

When a single modification on a software module results in the necessity to modify a series of other modules, the design suffers from rigidity. The OCP principle advocates design refactoring so that further modifications of the same type will no longer produce modifications on the existing code, which already functions, instead it will only require adding new modules.

A software module observing the Open-Closed principle has two main characteristics:

- *"Open for extensions."*

This means that the behaviour of the code can be extended. When the requirements of the project are modified, the code can be extended by implementing the new requirements, meaning that one can modify the behaviour of the already existing module.

- *"Closed for modifications."*

The implementation of the new requirements does not need modifications on the already existing code.

Abstraction is the method which allows the modification of the behaviour of a software module, without modifying its already existing code. In C++, Java or any other object oriented language, it is possible to create an abstraction which offers a fixed interface and an unlimited number of implementations, namely different behaviours (2).

Fig. 2 presents a block of classes that do not conform to the open-closed principle. Both the Client class and the Server class are concrete. The Client class uses the Server class. If we want for a Client object to use a different server object, the Client class must be changed to name the new server class.

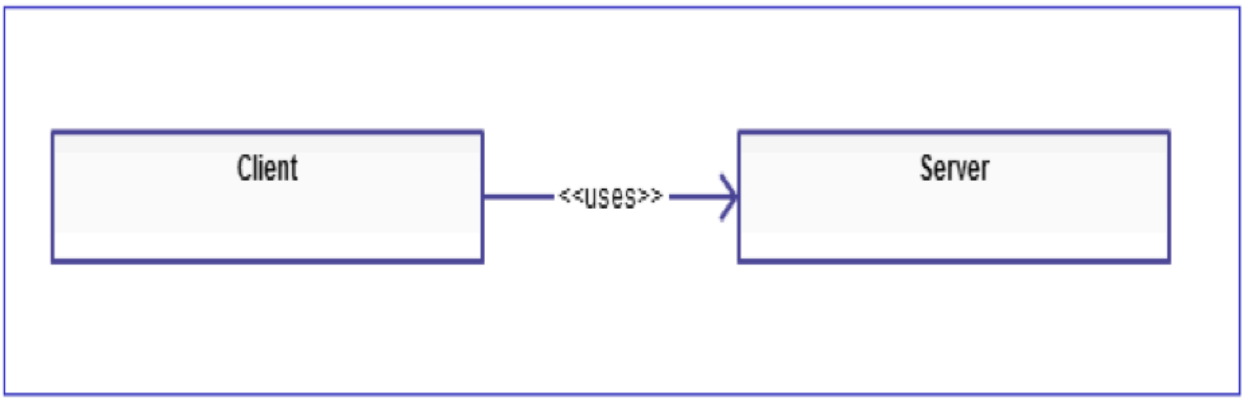

Figure 2. Example which does not comply with the OCP 1 principle

In Fig.3, the same design as the one in Fig.2 is presented, but this time the open-closed principle is observed. In this case, the abstract class AbstractServer was introduced, and the Client class uses this abstraction. However, the Client class will actually use the Server class which implements the ClientInterface class. If, in the future, one wishes to use another type of server, all that needs to be done is to implement a new class derived from the ClientInterface class, but this time the client doesn"t need to be modified.

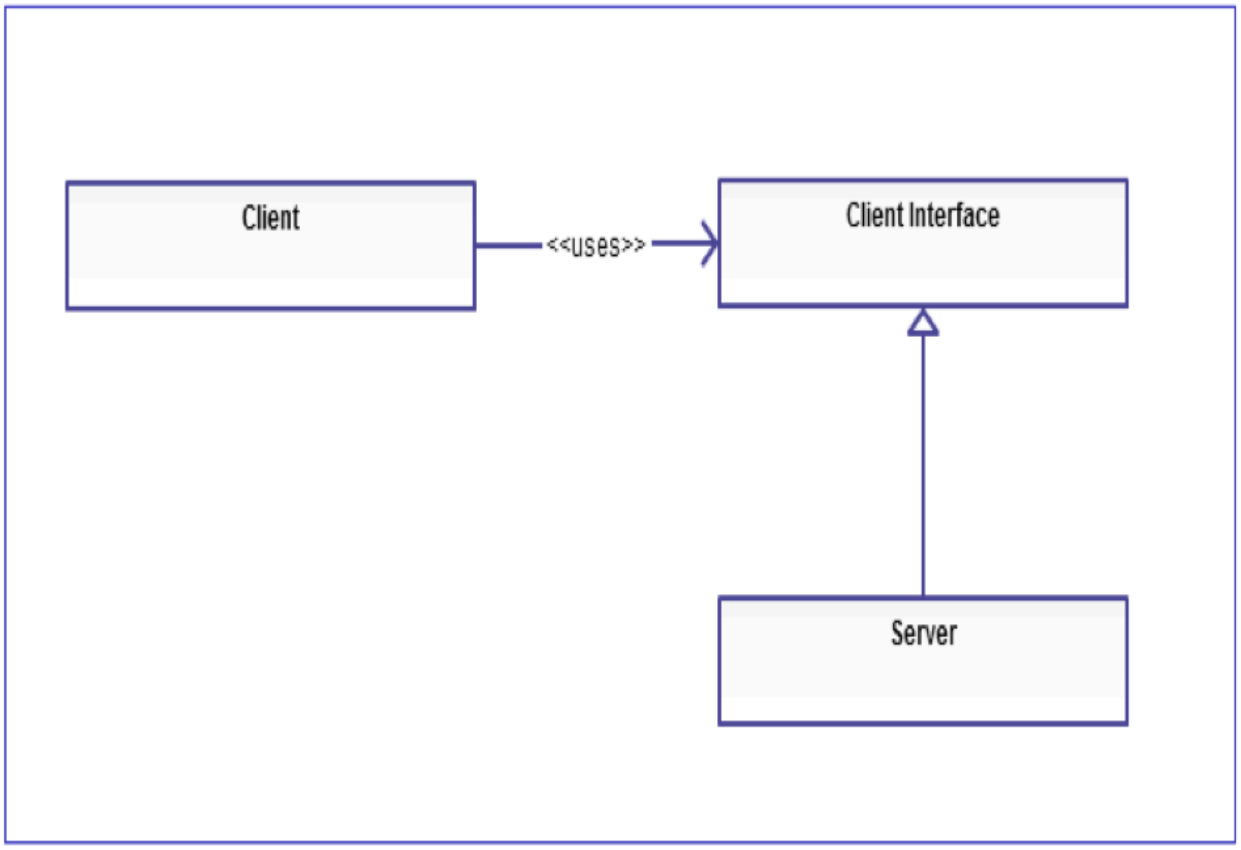

Figure 3. Example observing the OCP 1 principle

A particular aspect in this example is the way we named the abstract class ClientInterface and not ServerInterface, for example. The reason for this choice is the fact that abstract classes are more closely associated to their clients than to the classes that implement them.

The Open-Closed principle is also used in the Strategy and Plugin design patterns (3). For instance, Fig.4 presents the corresponding design, which observes the open-closed principle.

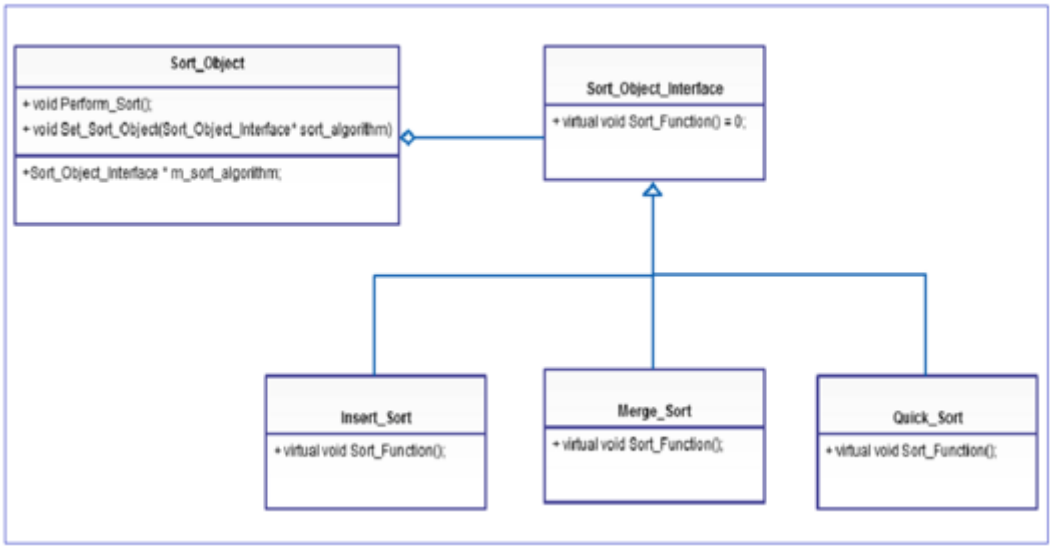

Figure 4

The Sort_Object class performs a function of sorting objects, function which can be described in the abstract interface Sort_Object_Interface. The classes derived from the abstract class Sort_Object_Interface are forced to implement the method Sort_Function(), but, at the same time, they have the freedom to offer any implementation for this interface. Thus, the behaviour specified in the interface of the method void Sort_Function(), can be extended and modified by creating new subtypes of the abstract class Sort_Object_Interface.

In the definition of the class Sort_Object we will have the following methods:

```
void Sort_Object::Sort_Function()
{
    m_sort_algorithm->sortFunction();
}
void Sort_Object::Set_Sort_Algorithm(const Sort_Object_Interface* sort_algorithm)
{
   std::cout << "Setting a new sorting algorithm..." << std::endl;
   m_sort_algorithm = sort_algorithm;
}
```

Conclusions

The main mechanisms behind this principle are abstraction and polymorphism. Whenever the code has to be modified in order to implement some new functionality, one must also take into consideration the creation of an abstraction which can provide an interface for the desired behaviour and offer at the same time the possibility to add new behaviours for the same interface in the future. Of course, the creation of an abstraction is not always necessary. This method is generally useful where there are frequent changes to be made.

Conformance to this open-closed principle is costly. It requires time to develop and effort to create the necessary abstractions. These abstractions increment the complexity of the software design.

In exchange, the Open/Closed Principle is, in many ways, at the heart of object-oriented programming. Conformance to this principle is what yields the greatest benefits claimed for object-oriented technology: code flexibility, reusability and maintainability.

The Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

In languages such as C++ or Java, the main mechanism through which abstraction and polymorphism is done is inheritance. In order to create a correct inheritance hierarchy we must make sure that the derived classes extend, without replacing, the functionality of the base classes. In other words, the functions using pointers or references to the base classes should be able to use instances of the derived classes without being aware of this. Contrary, the new classes may produce undesired outcomes when they are used in the entities of the already existing program. The importance of the LSP principle becomes obvious the moment it is violated.

Example:

Suppose we have a Shape class, whose objects are already used somewhere in the application and which has a SetSize method, containing the mSize property which can be used as a side or diameter, depending on the represented figure.

```
class Shape
{
 public:
      void SetSize(double size);
      void GetSize(double& size);
 private:
      double mSize;
};
```

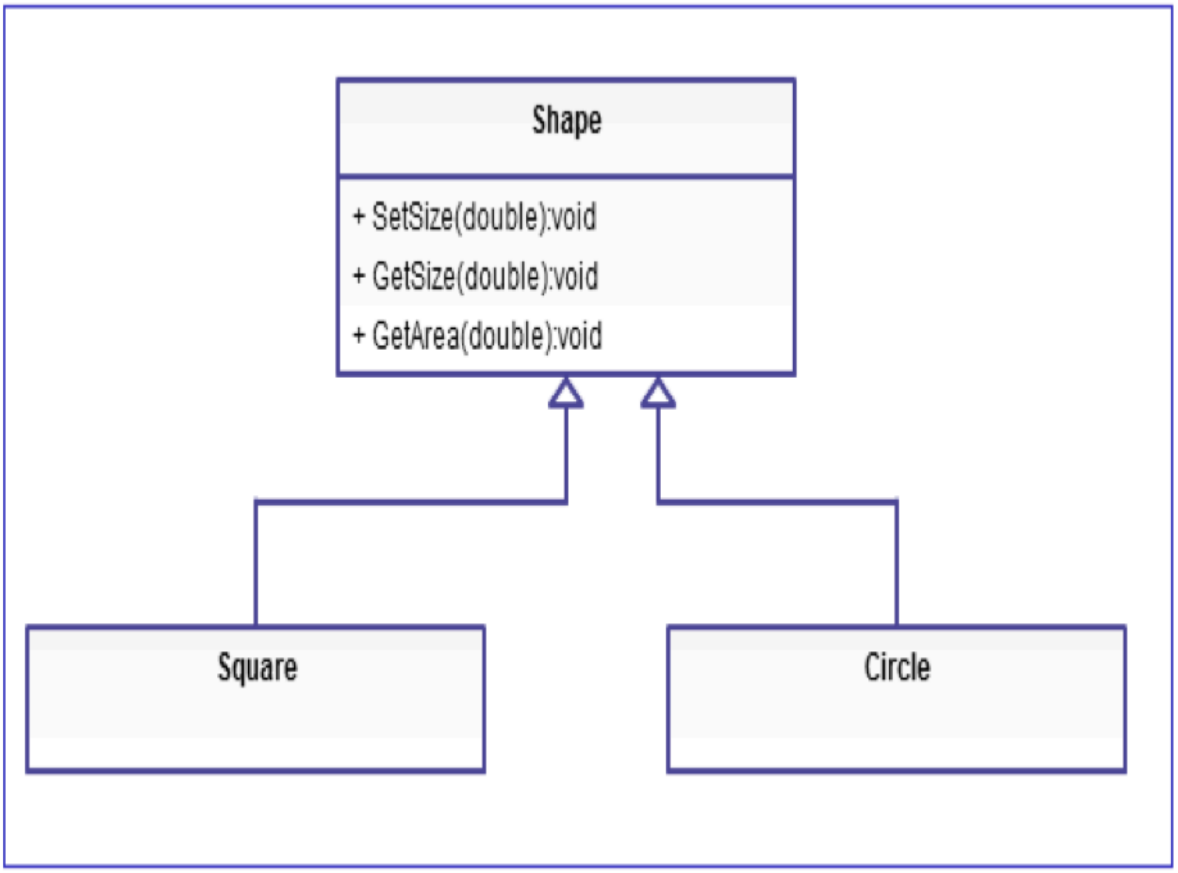

Figure 5

Later on, we will extend the application by adding the Square and Circle classes. Taking into consideration the fact that the inheritance models an IS_A relationship, the new Square and Circle classes can be derived from the Shape class.

Let"s suppose hereafter that the Shape objects are returned by a factory method, based on some conditions established at run time, so that we do not know exactly the type of the returned object. But we do know it is Shape. We get the Shape object, we set its size property to 10 units and we compute its surface. For a Square object, the area will be 100.

```
void f(Shape& shape,  float& area)
{
  shape.SetSize(10);
  shape.GetArea(area);
  assert(area == 100);  // Oups!
// for circle area = 314.15927!
}
// for circle area = 314.15927!
}
```

In this example, when the f function gets r as a parameter, an instance of the Circle class will have a wrong behaviour. Since, in function f, the Square type objects cannot substitute the

Rectangle type objects, the LSP principle is violated. The f function is fragile in relation to the Square/Circle hierarchy.

Design by Contract
Many developers may feel uncomfortable with the notion of behavior that is "reasonably assumed." How could you know what our users/ clients will really expect from the classes we are implementing?

To our help comes the design by contract technique (DBC). The contract of a method informs the author of a class about the behaviors that he can safely rely on. The contract is specified by declaring preconditions and postconditions for each method. The preconditions must be true in order for the method to execute. On completion, after executing the method, it guarantees that the postconditions are true.

Certain languages, such as Eiffel, have direct support for preconditions and postconditions. They only have to be declared, and during runtime they are automatically verified. In C++ or Java, this functionality is missing. Contracts can instead be specified by writing unit tests. By thoroughly testing the behavior of a class, the unit tests make the behavior of the class clear. Authors of client code will want to review the unit tests in order to know what to reasonably assume about the classes they are using.

Conclusions
The LSP principle is a mere extension of the Open-Closed principle and it means that, when we add a new class derived in an inheritance hierarchy, we must make sure that the newly added class extends the behaviour of the base class, without modifying it.

Dependency Inversion Principle (DIP)

A. High-level modules should not depend on low-level modules. Both should depend on abstract modules.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

This principle enunciates the fact that the high-level modules must be independent of those on lower levels. This decoupling is done by introducing an abstraction level between the classes forming a high hierarchy level and those forming lower hierarchy levels. In addition, the principle states that the abstraction should not depend upon details, but the details should depend upon the abstraction. This principle is very important for the reusing of software components. Moreover, the correct implementation of this principle makes it much easier to maintain the code.

Fig. 6 presents a diagram of classes organised on three levels. Thus, the PolicyLayer class represents the high-level layer and it accesses the functionality in the MechanismLayer class, situated on a lower level. In its turn, the MechanismLayer class accesses the functionality in the UtilityLayer class, which is also on a low-level layer. In conclusion, it is obvious that, in this class diagram, the high-levels depend upon the low-levels. This means that, if there is a modification on one of the low levels, chances are rather high that the modification propagates upwards, towards the high level layers, which means that the more abstract higher levels depend on the more concrete lower levels. So, the Dependency Inversion principle is violated.

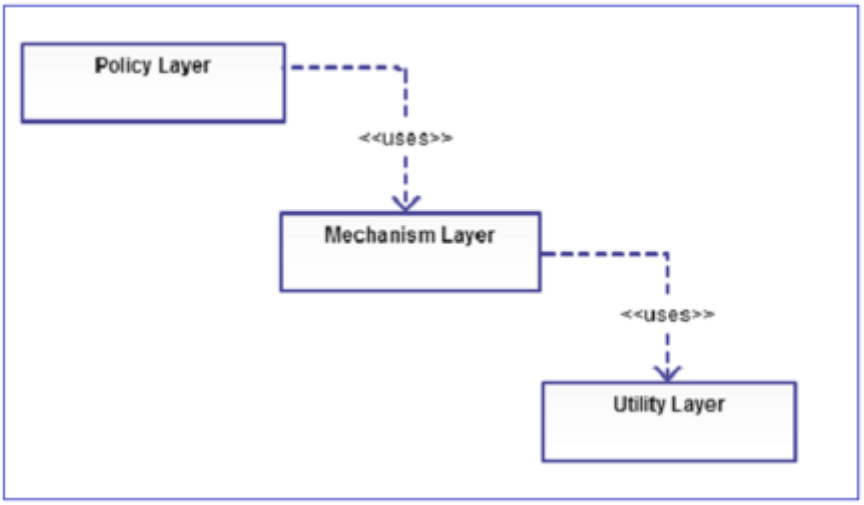

Figure 6

Figure 7 presents the same class diagram as in Fig.6, but this time the dependency inversion principle is observed. Thus, to each level accessing the functionality of a lower level, we added an interface which will be implemented by the lower level. This way, the interface through which the two levels communicate is defined in the higher hierarchical level, so that the dependency was reversed, namely the low level depends on the high level. Modifications carried out on the low levels no longer affect the high levels, but it happens backwards. In conclusion, the class diagram in Fig. 7 complies with the dependency inversion principle.

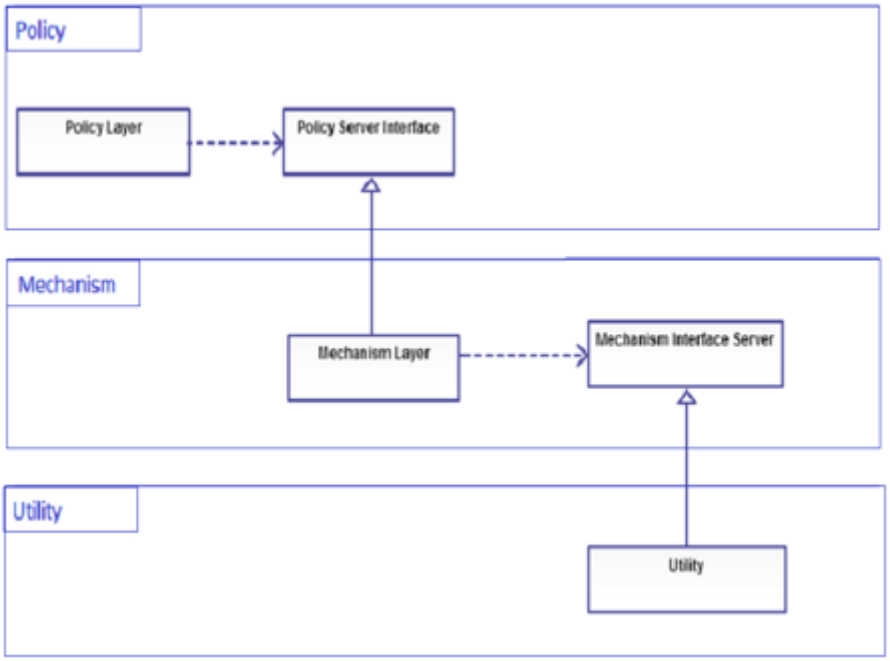

Figure 7

Conclusions

Procedural traditional programming creates dependency policies where high level modules depend on the details of low level modules. This programming method is inefficient, since

modifications of the details lead to modifications in the high level modules also. Object oriented programming reverses this dependency mechanism, so that both the details and the high levels depend upon abstractions, and the services often belong to the clients.

No matter the programming language used, if the dependencies are inverted, then the code design is object oriented. If the dependencies are not inversed, then the design is procedural. The dependency inversion principle represents the fundamental low level mechanism at the origin of many benefits offered by the object oriented programming. Complying with this principle is fundamental for the creation of reusable modules. It is also essential for writing code that can stand modifications. As long as the abstractions and the details are mutually isolated, the code is much easier to maintain.

The Interface Segregation Principle (ISP)

Clients should not depend on interfaces they do not use.

This principle stresses the fact that when an interface is being defined, one must be careful to put only those methods which are specific to the client in the interface. If in an interface one adds methods which do not belong there, then the classes implementing the interface will have to implement those methods, too. For instance, if we consider the interface Employee, which has the method Eat, then all the classes implementing this interface will also have to implement the Eat method. However, what happens if the Employee is a robot? Interfaces containing unspecific methods are called "polluted" or "fat" interfaces.

Figure 8 presents a class diagram containing: the TimerClient interface, the Door interface and the TimedDoor class. The TimerClient interface should be implemented by any class that needs to intercept events generated by a Timer. The Door interface should be implemented by any class that implements a door. Taking into consideration that we needed to model a door that closes automatically after a period of time, Fig.3.7 presents a solution in which we have introduced the TimedDoor class derived from the Door interface, and in order to also dispose of the functionality from TimerClient, the Door interface was also modified so as to inherit the TimerClient interface. However, this solution pollutes the Door interface, since all the classes that will inherit this interface will have to implement the TimerClient functionality (4), too.

```
class Timer
{
public:
    void Register(int timeout, TimerClient* client);
} ;
class TimerClient
{
public:
   virtual void TimeOut();
};

class Door
{
 public:
            virtual void Lock() = 0;
   virtual void Unlock() = 0;
```

```
    virtual bool IsDoorOpen() = 0;
} ;
```

The separation of interfaces can be done through the mechanism of multiple inheritance. In Fig. 9, we can see how multiple inheritance can be used to comply with the Interface Segregation principle in design. In this model, the TimeDoor interface inherits from both Door and TimerClient interfaces.

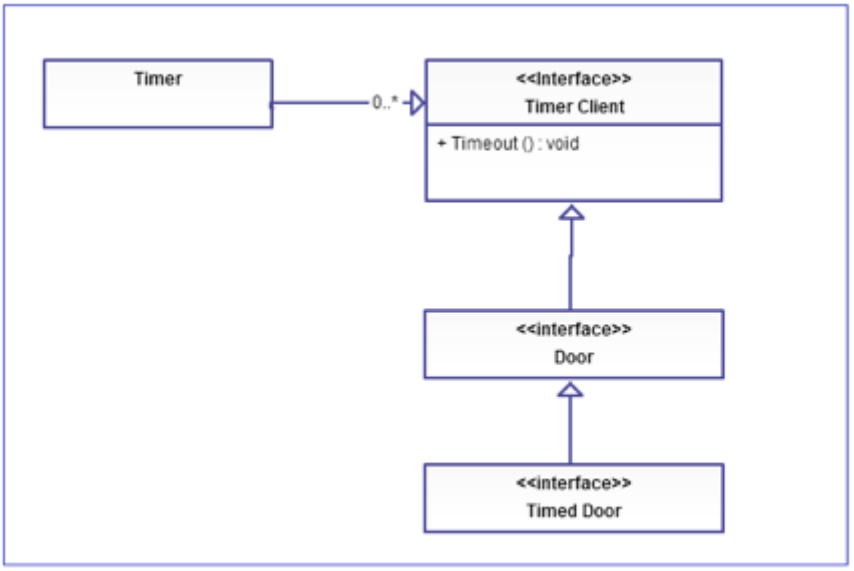

Figure 8

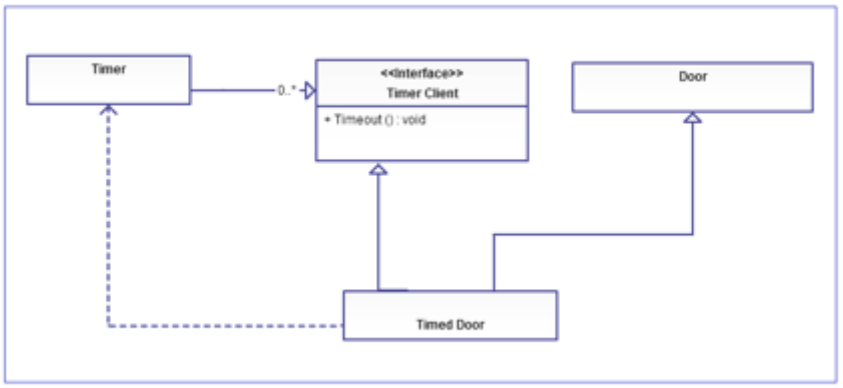

Figure 9

Conclusions

Polluted or fat classes cause harmful couplings between their clients. When one client forces a change on the fat class, all the other clients of the polluted class are affected. Thus, clients should have to depend only on methods that they actually call. This can be achieved by breaking the interface of the fat class into many client-specific interfaces. Each client-specific interface declares only those functions that its particular client or client group invoke. The fat class can then inherit all the client-specific interfaces and implement them. This breaks the dependence of the clients on methods that they don"t invoke and allows the clients to be independent of one another.