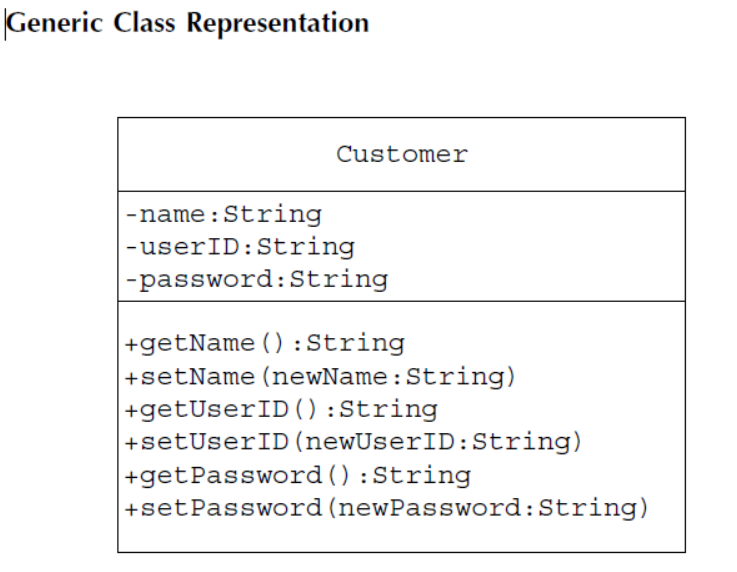
<https://dzone.com/articles/design-patterns-strategy>

**A design pattern is a documented best practice or core of a solution that has been applied successfully in multiple environments to solve a problem.**

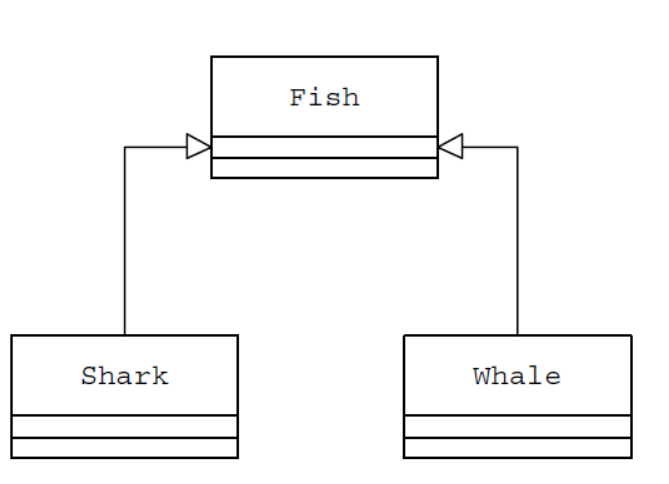
**Design pattern has many sections:**

* **Creational pattern: Deal with one of the most commonly performed tasks in an OO application, the creation of objects. support a uniform, simple, and controlled mechanism to create objects.**
* **Collectional patterns: Deal with groups or collections of objects.**
* **Behavioral Patterns: Deal with the details of assigning responsibilities between different objects, describe the communication mechanism between objects.**
* **Structural patterns: Deal with objects delegating responsibilities to other objects.**
* **Concurrency patterns: Ways to lock class code and an order of locking objects to prevent the occurrence of race conditions and deadlocks**

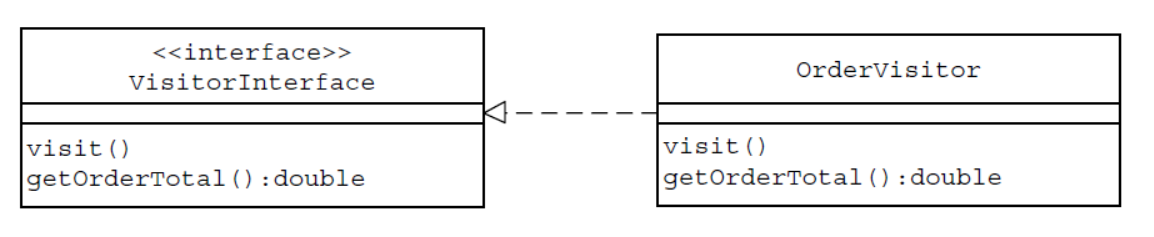
**Introduction toUML(unified modelling language):**



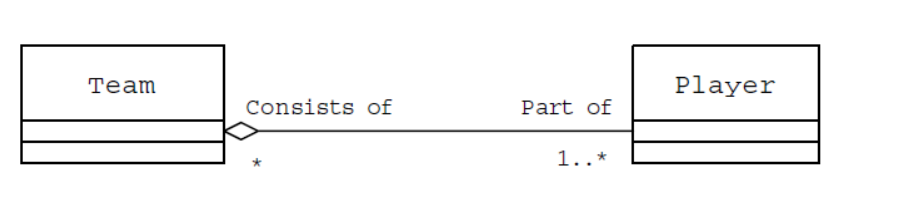
**Inheritance relationship:**



**Interface and it’s implementer:**



**Composition:**

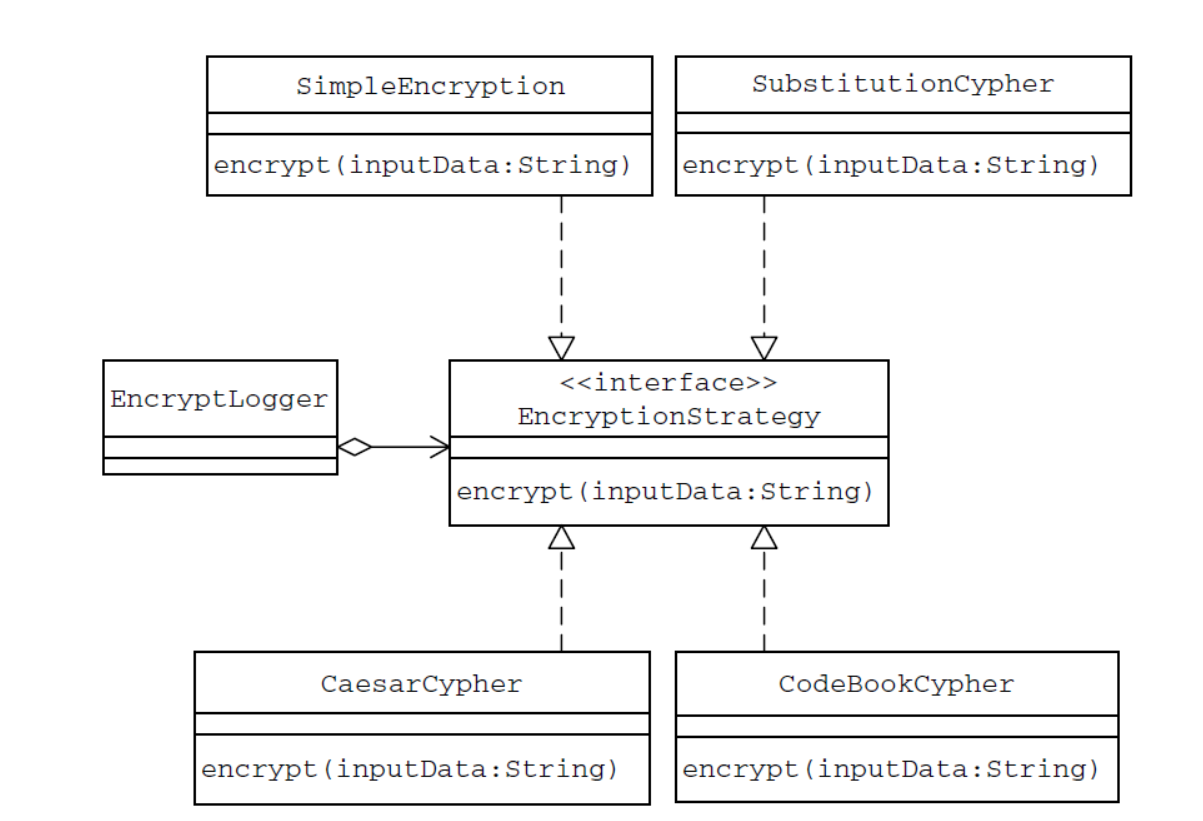


Strategy Pattern ( behavioral pattern)

The Strategy pattern is useful when there is a set of related algorithms and a

client object needs to be able to dynamically pick and choose an algorithm from this set that suits its current need.

The Strategy pattern suggests keeping the implementation of each of the algorithms in a separate class. Each such algorithm encapsulated in a separate class is referred to as a *strategy*. An object that uses a Strategy object is often referred to as a *context object*.



Another example:

//Strategy Interface

public interface Compression Strategy {

public void compressFiles(Array List<File> files);

}

And we'll need to provide our two implementations, one for zip and one for rar

public class ZipCompressionStrategy implements CompressionStrategy {

public void compressFiles(ArrayList<File> files) {

//using ZIP approach

}

}

public class RarCompressionStrategy implements CompressionStrategy {

public void compressFiles(ArrayList<File> files) {

//using RAR approach

}

}

Our context will provide a way for the client to compress the files. Let's say that there is a preferences setting in our application that sets which compression algorithm to use. We can change our strategy using the setCompressionStrategy method in the Context.

public class CompressionContext {

private CompressionStrategy contextstrategy;

//this can be set at runtime by the application preferences

public void setCompressionStrategy(CompressionStrategy strategy) {

contextstrategy = strategy;

}

//use the strategy

public void createArchive(ArrayList<File> files) {

contextstrategy.compressFiles(files);

}

}

It's obvious that all the client has to do now is pass through the files to the CompressionContext

public class Client {

public static void main(String[] args) {

CompressionContext ctx = new CompressionContext();

//we could assume context is already set by preferences

ctx.setCompressionStrategy(new ZipCompressionStrategy());

//get a list of files...

ctx.createArchive(fileList);

}

}