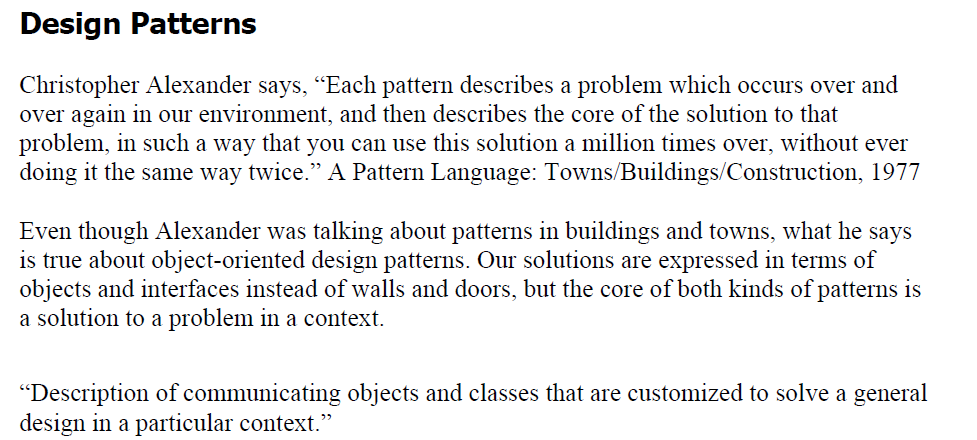
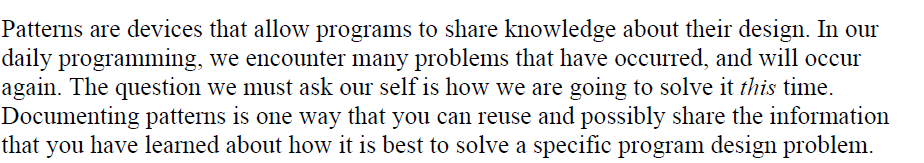
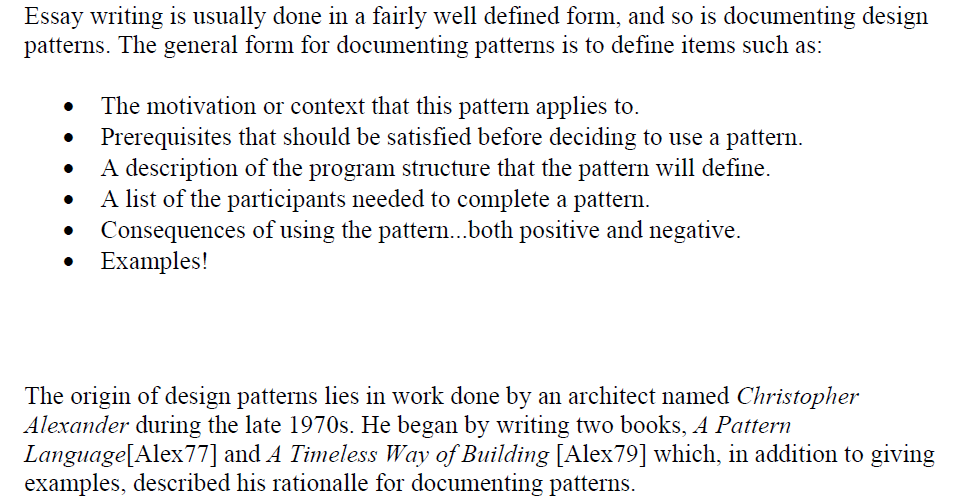
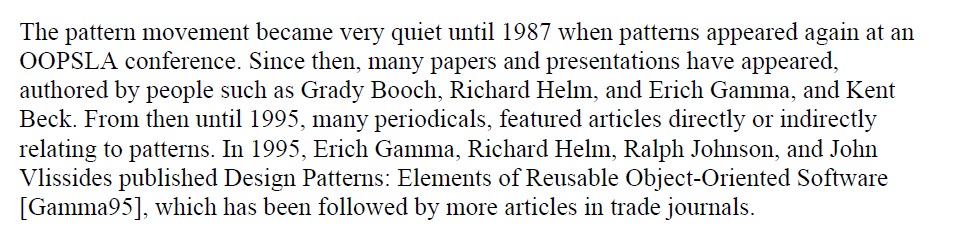
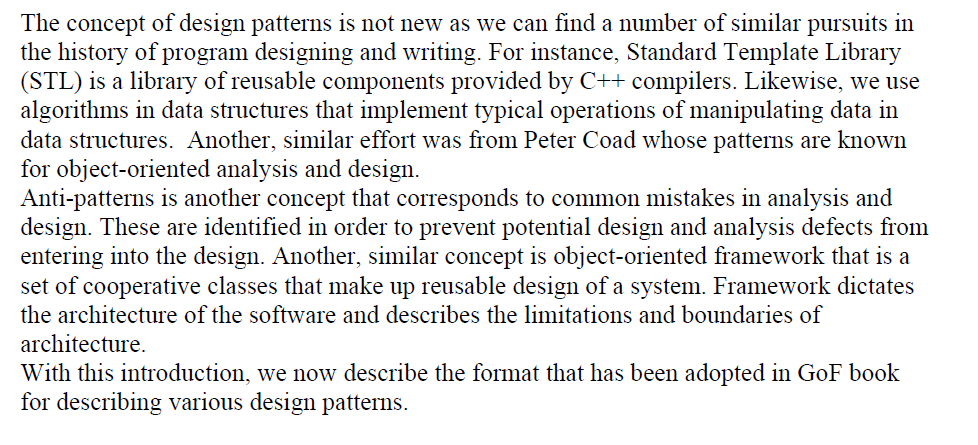
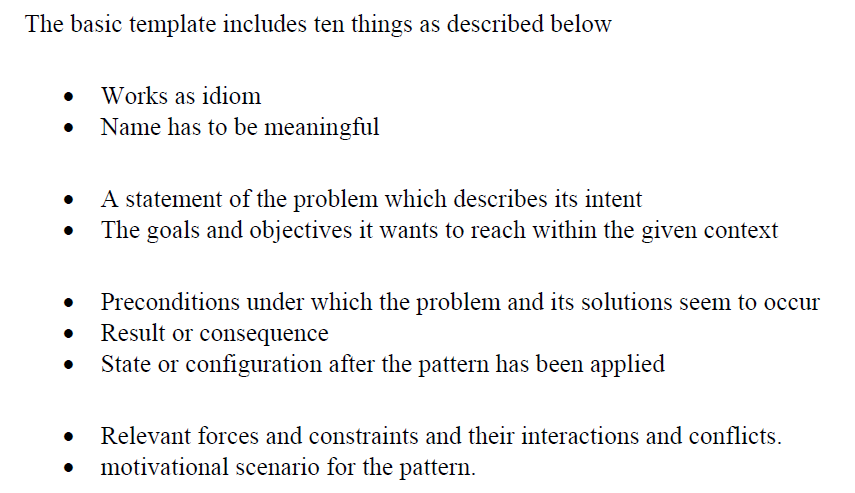
0 

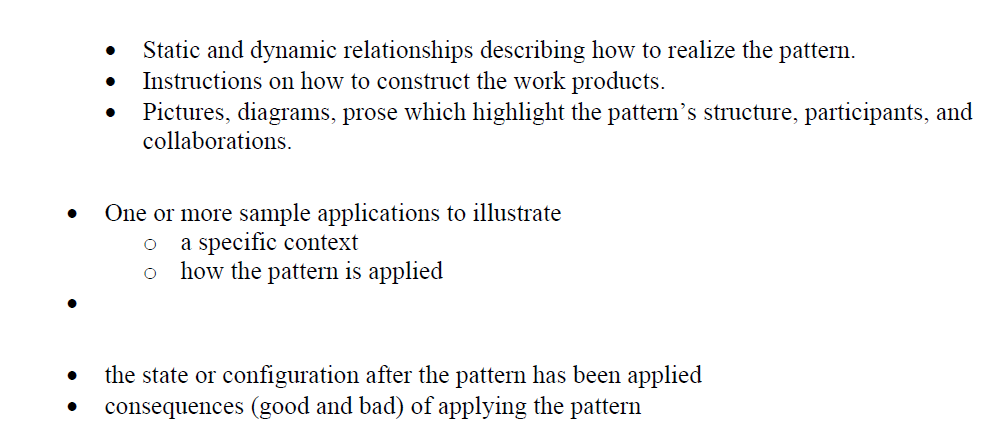


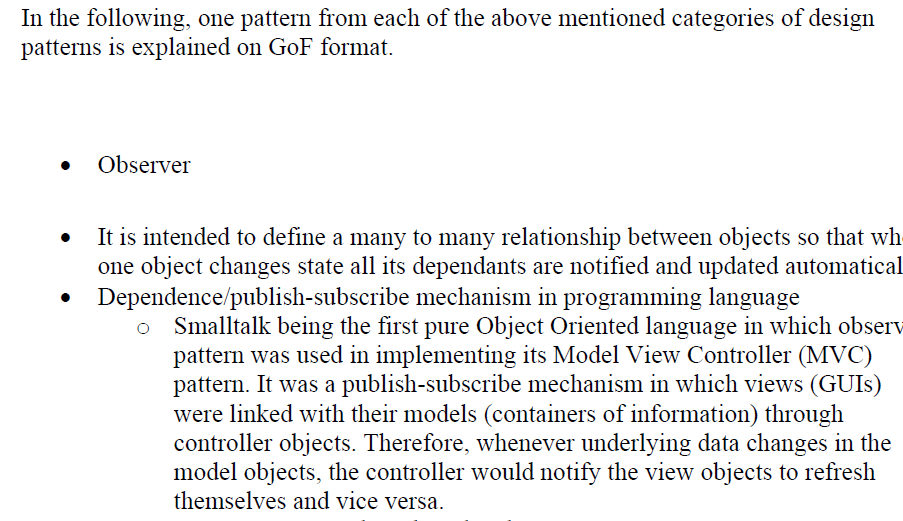


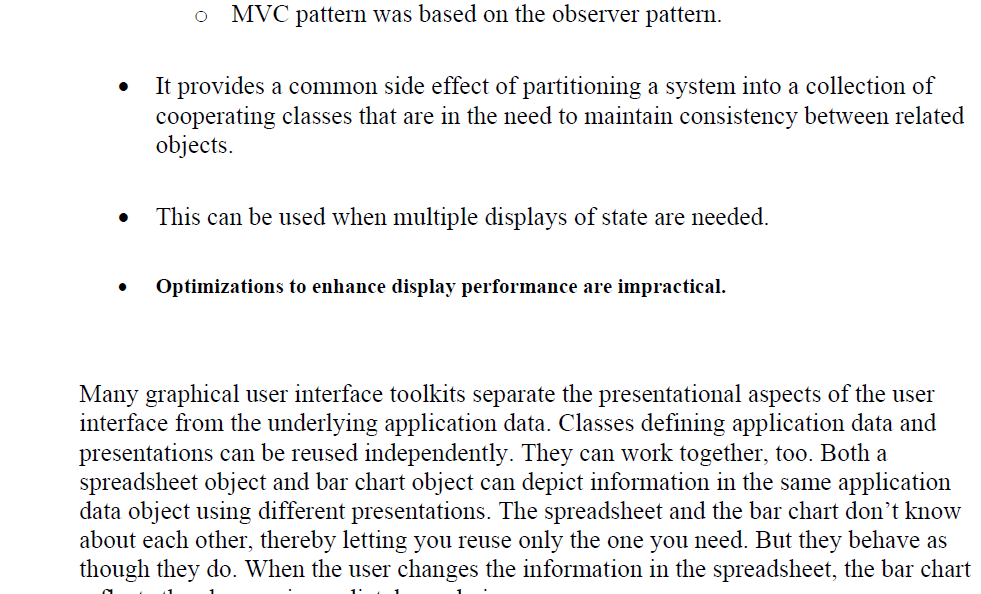




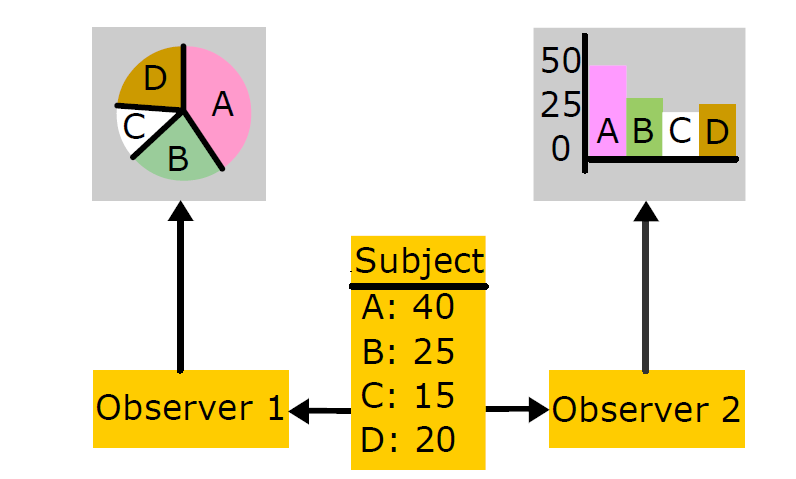


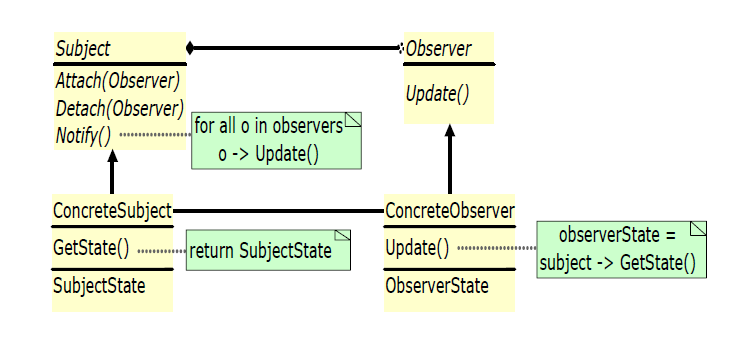


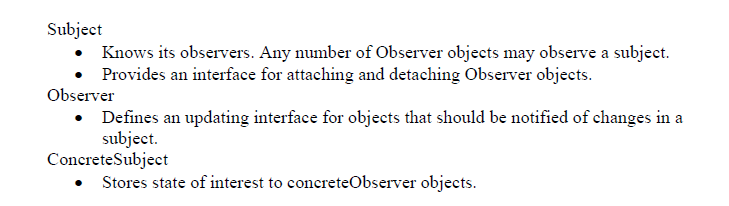


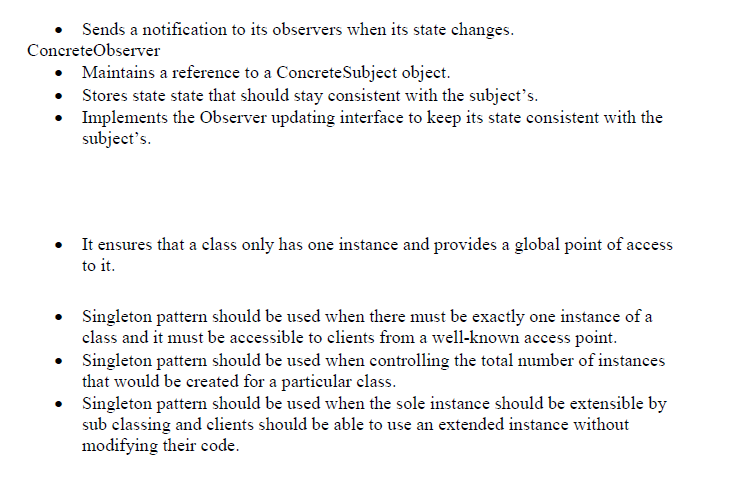












Please visit the following design pattern repository

<http://www.oodesign.com>

Example

Singleton Pattern

Motivation

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

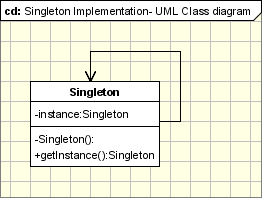
The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

Intent

* Ensure that only one instance of a class is created.
* Provide a global point of access to the object.

Implementation

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.

|  |
| --- |
| class Singleton {  private static Singleton instance;  private Singleton()  {  ...  }   public static synchronized Singleton getInstance()  {  if (instance == null)  instance = new Singleton();   return instance;  }  ...  public void doSomething()  {  ...   } } |

You can notice in the above code that getInstance method ensures that only one instance of the class is created. The constructor should not be accessible from the outside of the class to ensure the only way of instantiating the class would be only through the getInstance method.

The getInstance method is used also to provide a global point of access to the object and it can be used in the following manner:

|  |
| --- |
| Singleton.getInstance().doSomething(); |

Applicability & Examples

According to the definition the singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point. Here are some real situations where the singleton is used:

Example 1 - Logger Classes

The Singleton pattern is used in the design of logger classes. This classes are ussualy implemented as a singletons, and provides a global logging access point in all the application components without being necessary to create an object each time a logging operations is performed.

Example 2 - Configuration Classes

The Singleton pattern is used to design the classes which provides the configuration settings for an application. By implementing configuration classes as Singleton not only that we provide a global access point, but we also keep the instance we use as a cache object. When the class is instantiated( or when a value is read ) the singleton will keep the values in its internal structure. If the values are read from the database or from files this avoids the reloading the values each time the configuration parameters are used.

Example 3 - Accesing resources in shared mode

It can be used in the design of an application that needs to work with the serial port. Let's say that there are many classes in the application, working in an multi-threading environment, which needs to operate actions on the serial port. In this case a singleton with synchronized methods could be used to be used to manage all the operations on the serial port.

Example 4 - Factories implemented as Singletons

Let's assume that we design an application with a factory to generate new objects(Acount, Customer, Site, Address objects) with their ids, in an multithreading environment. If the factory is instantiated twice in 2 different threads then is possible to have 2 overlapping ids for 2 different objects. If we implement the Factory as a singleton we avoid this problem. Combining Abstract Factory or Factory Method and Singleton design patterns is a common practice.