Design Patterns

What are Design Patterns?

- Design patterns are well tried out solutions to common design problems.

- A design pattern has the following:
  - Name: Obviously, this is to identify the pattern.
    - Example:
      - *Iterator*
      - *Smart Pointer*

  - Problem: This tells you when to use the pattern, i.e., what problem to solve using this pattern.

  - Solution: This tells you what classes make up the pattern, what their responsibilities are and how they interact (i.e., collaboration).

  - Consequences: Results, space or time trade-offs

Creational design patterns

- Some examples are:
  - Singleton
  - Proxy
  - Abstract factory
  - Factory method
  - Builder

Structural design patterns

- Some examples are:
  - Proxy
  - Facade
  - Decorator
  - Composite
  - Flyweight
  - Adapter

Behavioral design patterns

- Some examples are:
  - Template method
  - Strategy
  - Visitor
  - Iterator
  - Observer
  - Command
  - Interpreter

Concurrency patterns

- Some examples are:
  - Double checked locking
  - Monitor
  - Reader writer lock
  - Thread specific storage

Singleton

Singleton: A Creational Pattern

- *Singleton* is an example of a creational pattern.
- This can be used when you want to ensure only one instance of a class is created in the application.
- This one instance should be accessible by the client code.
- You could have one global object of this class, and everyone could access it, but that does not stop someone from creating another instance of this class.
- Hence, the intelligence that the class is a Singleton is built into the class, and it ensures that no more than one instance of it is created.

Singleton

```
class MySingletonClass   {
  public:
    static  MySingletonClass *  Instance();
  protected:
    MySingletonClass();        ← Protected, so no one can instantiate this directly.
private:
    // copy constructor and assignment operator are private, and not implemented.
    MySingletonClass( const MySingletonClass & other);
    MySingletonClass & operator = ( const MySingletonClass & other);

    static  MySingletonClass  *  m_Instance;
};

MySingletonClass  *  MySingletonClass::m_Instance = NULL;
MySingletonClass  *  MySingletonClass::Instance()
{
    if (m_Instance == NULL)
            m_Instance = new MySingletonClass();          ← Instantiated only ONCE.

    return   m_Instance;
}
```

Singleton

- Examples of singleton could be:
  - Database object
  - Window manager
  - Print spooler

Singleton

- In multi threaded environments, you have to use locking to prevent multiple objects being created.
- Given a multi threaded application, what is a potential problem in code below:

```
MySingletonClass *  MySingletonClass::Instance()
{
    if (m_Instance == NULL)
            m_Instance = new MySingletonClass();

    return   m_Instance;
}
```

Singleton

- Now, you would want to add a lock to prevent multiple threads from allocating a SingletonClass object and writing to m_Instance

```
MySingletonClass  *  MySingletonClass::Instance()
{
     Lock                    lock;                      // this will lock

     if (m_Instance == NULL)
               m_Instance = new MySingletonClass();

     return   m_Instance;
}
```

- However, the code above still has a problem. Why ?

Singleton

```
MySingletonClass  *  MySingletonClass::Instance()
{
      if (m_Instance == NULL)
      {
            Lock                      lock;                     // this will lock

            m_Instance = new MySingletonClass();
      }

      return   m_Instance;
}
```

- But the code above still has a problem. Why?

Singleton

```
MySingletonClass  *  MySingletonClass::Instance()
{
    if (m_Instance == NULL)
    {
        Lock                    lock;                   // this will lock

        if (m_Instance == NULL)
        {
                m_Instance = new MySingletonClass();
        }
    }

    return   m_Instance;
}
```

- This should work now.

Proxy

Proxy: A Structural Pattern

- *Proxy* is an example of a Structural pattern.

- This pattern is used when you want to
  - control access to an object (Protection proxy)
  - local representative of a remote object (Remote proxy)
  - when you want to encapsulate some intelligence in the proxy (e.g. Smart Pointer, which is a special case of the Proxy pattern).

Proxy

- So, the proxy does the following:
  - Keeps a pointer (or reference) to the original object (the one that it is a proxy for)
  - The original object is accessed via this proxy object.
  - And for that reason, proxy has to implement the same interface as the original object.
  - Proxy may also be responsible for deleting / releasing the original object/resource.

Proxy

- Protection proxy:
  - An example could be the proxy web server that is used behind a firewall. It serves as a proxy to the web servers outside the firewall of a company.

- Remote proxy:
  - RMI (Remote Method Invocation) in Java is an example of a remote proxy, as you can think of it as a representation(API) of the remote resource(some application/service) running on some computer, perhaps in a different geographical region.
  - Above is an example specific to Java, but you can think of RPC (Remote Procedure Call) in general.

Proxy

- Could possibly also use it to ensure locks are used (assuming the proxied object is a shared resource requiring lock before access).
- I have not used it this way, but the following scenario seems possible:
  - Lets say there is a shared resource that requires write locks.
  - SmartPointer->read(int index) is a proxy to the object for reading.
  - SmartPointer->write(int index) is a proxy to the object that, on a write operation, would do the following:
    - Acquire write lock
      - wait if lock not available
    - Write to object.
    - Release write lock.

Iterator: A Behavioral Pattern

- *Iterator* pattern is a Behavioral pattern.
- It is also known as *cursor.*

- This pattern is used when you want to provide the capability to “walk” through the elements of an aggregation (collection).

- In other words, you want to iterate over a collection (also called containers in STL).

- Advantage of having an iterator capability is that the client code does not need to understand the underlying representation of the collection, i.e., it does not need to figure out how to go from one element to the next. It simply uses the iterator.

Iterator

- In other words, this pattern solves the problem of "moving through a collection of objects, when you don't know how the collection is implemented".

Iterator: Examples

```
vector < string >  strVector;

vector < string >::iterator  start = strVector.begin();        ← start points to first element
vector < string >::iterator  end  = strVector.end();           ← end points to one past last element

while ( start != end )
{
    … // processing code
    ++ start;                                  ← Go to next element. Note:  prefer pre increment
}
```

```
list  < string >  strList;

list < string >::iterator  start = strList.begin();            ← start points to first element
list < string >::iterator  end  = strList.end();               ← end points to one past last element

while ( start != end )
{
    … // processing code
    ++ start;                                  ← Go to next element. Note:  prefer pre increment
}
```

Template method pattern

- This is a fairly common design pattern, and has nothing to do with C++ templates.
- In this, you have a base class that lays down the "algorithm"… i.e., it has the calls to the required methods, but these methods may not have an implementation (or have a default implementation).
- The derived classes will have an implementation for these methods that are being called in the "algorithm" in the base class.
- Since the derived classes will have their own customized implementation of the methods, the actual actions taken will be different for different classes, but the higher level algorithm or steps will be the same.
- The derived class' methods get called by the base class (as part of the algorithm in the base class).

Template method pattern

- For example:

- Consider a distributed cache, where different types of entries are stored, and on a cache access, there is some post compute that gets done on the entries BEFORE the results are returned.

```
RetrieveFromCache()
{
		CacheData  cData = ReadCache( key );

		cData.PostCompute();

		return cData;
}
```

Template method pattern

```
CacheData::PostCompute()
{
            PostComputeStep1();
            …
            PostComputeStepN();
}

CacheData::PostComputeStep1()
{
            /// empty, no actions here
}

CacheData::PostComputeStep2()
{
            DefaultActionCall();    // Default actions here
}

CacheDataType1::PostComputeStep1()   ← This method of derived class is called by
{                                      base class PostCompute
            // customized steps here
}
```

Prototype pattern

- This is used when you want to copy objects (polymorphic).
- The code copying an object does not need to know the exact type of the object, because the object implements a clonable interface (or copyable interface).
- In fact, many times, the code copying an object may not be able to know (at least not easily) the type of the object(s) that need to be copied.
- See code example

Observer pattern

- This pattern is used when an application (or part of an application) wants to know when some other application (or some other part of that same application) changes state.

- Example:
  - Consider a server cache, to which clients are connected to read or update cache data.
  - Lets say that the clients also cache some entries locally(that they have accessed previously).
  - The clients now would like to keep their local cache updated as and when their locally cached values change on the cache server.

  - To solve the above problem, we can implement the observer pattern.
  - The clients can "register interest" with the server for the values that the client has cached locally (this local cache will of course keep changing).

Observer pattern

- This is like a push based notification.
- You could think of solving this problem using a pull notification, where the clients would poll the server regularly for any changes (at a fixed, configurable interval).
- However, there are downsides to this approach:
  - The change on server may happen in between the polling window, and the client would have stale data up until the next polling window.
  - This would take up more processing resource on the clients.
  - Likely would cause more network traffic when all clients poll the server(s) for changes.
    - Note: There could be multiple servers, not just multiple clients