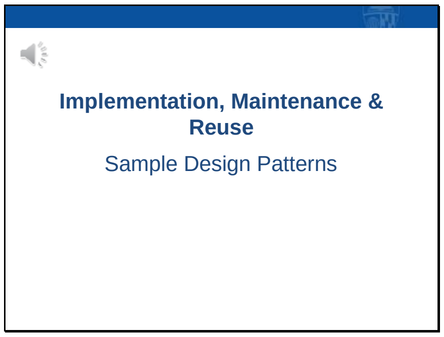
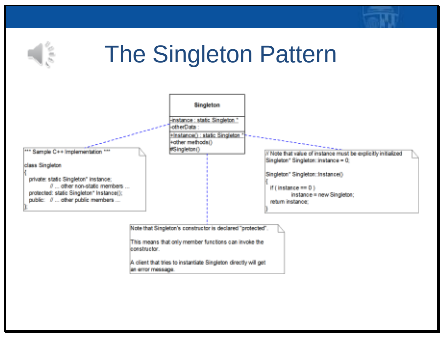
1 

Implementation, Maintenance & Reuse

Sample Design Patterns

In this lecture, we're going to introduce three widely used design patterns. The first one, called the Singleton pattern, is an example of a creational pattern. The second one, the Façade pattern, is an example of a structural pattern. And the third pattern, the Observer pattern, is an example of a behavioral pattern.

2 

The Singleton Pattern

```
Singleton
-instance : static Singleton *
-otherData :
+Instance() : static Singleton *
+other methods()
#Singleton()
```

```
*** Sample C++ Implementation ***

class Singleton
{
  private: static Singleton* instance;
             // ... other non-static members ...
  protected: static Singleton* Instance();
  public:   // ... other public members ...
};
```

```
// Note that value of instance must be explicitly initialized
Singleton* Singleton::instance = 0;

Singleton* Singleton::Instance()
{
  if ( instance == 0 )
             instance = new Singleton;
  return instance;
}
```

Note that Singleton's constructor is declared "protected".

This means that only member functions can invoke the constructor.

A client that tries to instantiate Singleton directly will get an error message.

The Singleton pattern is used when we want to ensure that only a single instance from a class is instantiated. It's one of the original "gang of four" patterns.

So…the Singleton pattern ensures that only a single instance from a class is permitted to be instantiated, and it also provides a single point of access to that object.

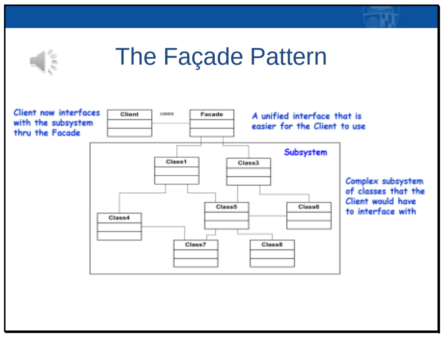
The Singleton pattern solution is pretty simple. I'm showing a Singleton implementation using the C++ language here. A similar solution would exist in the Java language, except that references instead of pointers would be used.

The key components of the pattern are first…the class constructor is not made public. In this example, the Singleton class constructor is given protected access. This means that only methods of the Singleton class can invoke it.

The second key feature of this pattern is the static variable "instance", which in this case is a pointer to a Singleton object. Because instance is static, only one instance will exist.

And the third key feature is the static method named Instance, that returns a pointer to the unique Singleton object. If you look at the code for this method you can see how things work. When the Instance method is called, it checks to see if the instance class member is null. If it is null, that means a Singleton hasn't been created yet, so a Singleton pointer is created and returned. If a Singleton has already been created, its address is returned.

3



The Façade Pattern

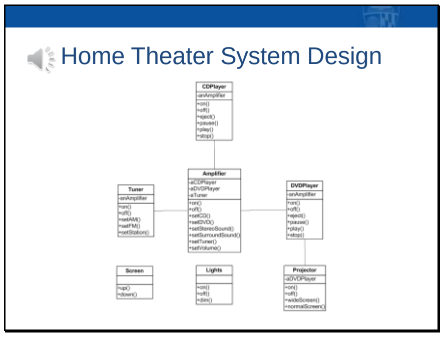
Client now interfaces with the subsystem thru the Facade
Client
uses
Facade
A unified interface that is easier for the Client to use
Subsystem
Class1
Class3
Complex subsystem of classes that the Client would have to interface with
Class5
Class6
Class4
Class7
Class8

Another useful structural pattern is the Façade pattern. This pattern can be used to provide a unified, more simplified interface that hides any complex relationships that might exist.

For example, we might have a subsystem of classes that interface and interact with each other in a complicated way, and we might have a client class that doesn't want or need to know about those complicated interactions.

In situations like this, a Façade class can be defined to simplify the interface that the client uses.

4



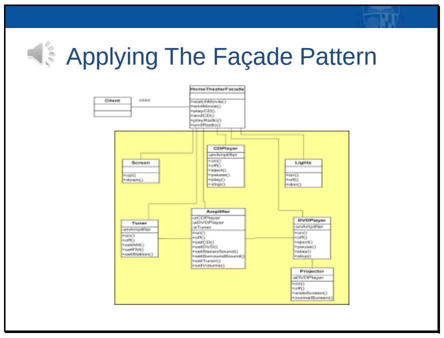
Home Theater System Design

Suppose we have a bunch of classes that comprise a home theater system. There's a lot of different classes and interfaces that exist. There may be clients of this system that don't really need to know how everything is related in order to get certain things done.

So, we want to hide the unnecessary relationships and interfaces from those clients to make things simpler for them to use.

And, we also want to minimize any maintenance that might have to be done to clients if some of the details of the subsystem components are changed.

5



Applying The Façade Pattern

Let's assume a client wants to be able to use the home theatre system to play movies and CDs, and listen to the radio.

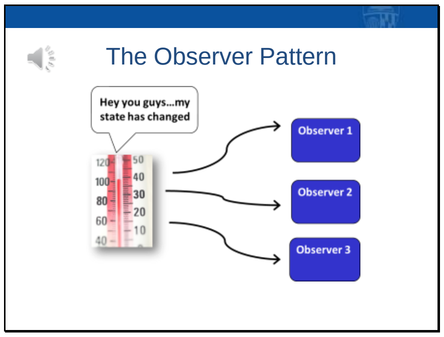
A HomeTheaterFacade class could be defined that provided the client with a simple way to do this…and hide the complicated low level details.

For example, to play a DVD, the following activities have to be done. Dim the room lights, lower the projection screen, turn the projector on, set projector input mode to DVD, set projector to display wide screen, turn the amplifier on and set it to surround sound, turn the DVD player on, and have it play.

That's a lot of steps. Using the façade class all those steps could be delegated to a watchMovie() method that the client could call to watch a movie.

And…if any components or interfaces were changed, the façade class methods may need to be changed, but clients of the façade class wouldn't be impacted at all.

6

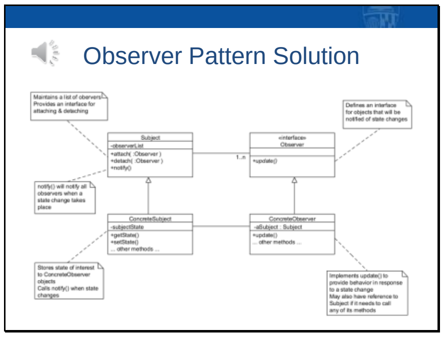


The Observer Pattern

The Observer pattern, yet another "gang of four pattern", which also goes by the name publish-subscribe pattern, is a very useful pattern in situations where an object needs to notify other objects whenever its state changes.

For example, maybe we have a weather application in which it is necessary for a temperature object to notify other objects whenever there is a change in temperature.

7



Observer Pattern Solution

Maintains a list of obervers Provides an interface for attaching & detaching
Defines an interface for objects that will be notified of state changes
Subject
-observerList
+attach(:Observer)
+detach(:Observer)
+notify()
1..n
«interface» Observer
+update()
notify() will notify all observers when a state change takes place
ConcreteSubject
-subjectState
+getState()
+setState()
... other methods ...
ConcreteObserver
-aSubject : Subject
+update()
... other methods ...
Stores state of interest to ConcreteObserver objects Calls notify() when state changes
Implements update() to provide behavior in response to a state change May also have reference to Subject if it needs to call any of its methods

Here's one design solution to the Observer pattern. There are a number of others, but this is the classic solution.

The key elements of this design pattern involve a Subject class and an Observer class. The Subject class is the class that needs to notify others whenever a state change takes place. The Observer class corresponds to a class that wants to be notified of state changes.

Let's start with the Subject class. The Subject class maintains a list of Observers. It also contains an attach() method that an Observer can call to register itself to be notified of state changes…and a detach() method that an Observer can call to unregister itself from the list. It also has a notify() method that notifies observers when a state change occurs.

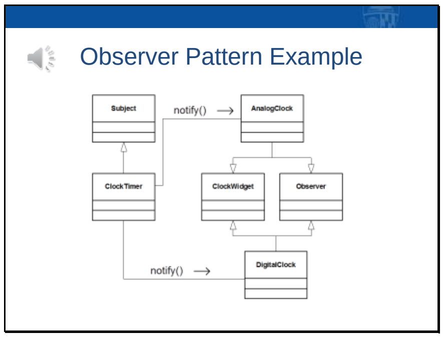
The Observer class is typically implemented as an interface, since its key component is the signature for a method called update(). The update() method will be the method called by the Subject's notify() method when there is a state change.

A ConcreteSubject class inherits from the Subject superclass. It implements the notion of state in one or more of its attributes, and may provide a getState() and setState() method, in addition to other methods specific to its behavior.

A ConcreteObserver class implements the update() method to provide whatever behavior is appropriate for it when a subject's state changes. The class may also have a reference to Subject if it needs to call any Subject methods such as getState().

That's all there is to it. Sometimes Subject is defined as an abstract class and delegates implementation of attach(), detach(), and notify() to its concrete subclasses.

8


Observer Pattern Example

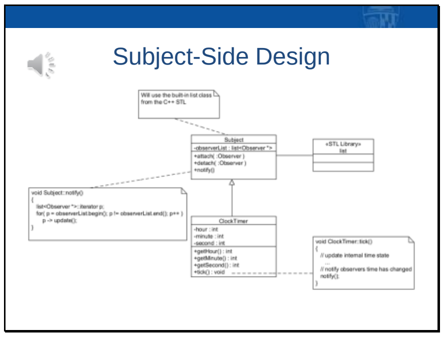
Let's take an example to see how the Observer pattern works in practice.

Assume an application needs to provide the capability to display both digital clocks and analog clocks. Assume also that we have a class called ClockWidget that provides graphical functionality for displaying clocks.

We will define a class, called ClockTimer, that maintains the current time of day and notifies any listeners of the current time every second. ClockTimer will inherit from the Subject class.

The AnalogClock and DigitalClock classes are the concrete Observer classes in this example. So they will inherit from the ClockWidget and Observer classes.

9



Subject-Side Design

```
void Subject::notify()
{
  list<Observer *>::iterator p;
  for( p = observerList.begin(); p != observerList.end(); p++ )
     p -> update();
}
```

```
void ClockTimer::tick()
{
  // update internal time state
     ...
  // notify observers time has changed
  notify();
}
```

We'll start on the subject side of the design, and just to show some real code snippets we'll demonstrate using the C++ language.

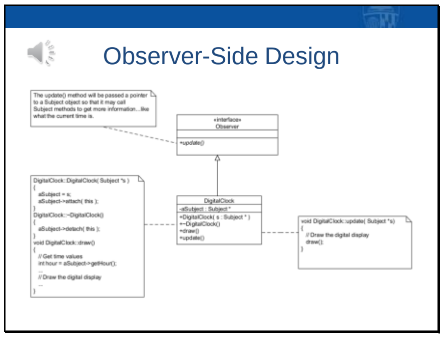
The Subject class will use a built-in list class from the C++ STL..or Software Template Library. The list will contain pointers to the Observer objects that are attached to the Subject. These Observer objects will be notified of state changes.

The notify() method simply iterates through the list of Observers and invokes each observer's update method.

The ClockTimer class will be the concrete subject class. It will maintain the current time, and it will notify its observers whenever the current time changes. The ClockTimer class also has a method named tick(), which gets called by an internal timer at regular intervals so that the current time is updated. All tick() does is to update the current hour, minute and second, and then calls the notify method which notifies the list of observers.

That's all there is to it on the Subject side.

10



Observer-Side Design

The update() method will be passed a pointer to a Subject object so that it may call Subject methods to get more information…like what the current time is.
«interface»
Observer
+update()
DigitalClock
-aSubject : Subject *
+DigitalClock(s : Subject *)
+~DigitalClock()
+draw()
+update()

```
DigitalClock::DigitalClock( Subject *s )
{
  aSubject = s;
  aSubject->attach( this );
}
DigitalClock::~DigitalClock()
{
  aSubject->detach( this );
}
void DigitalClock::draw()
{
  // Get time values
  int hour = aSubject->getHour();
  ...
  // Draw the digital display
  ...
}
```

```
void DigitalClock::update( Subject *s)
{
  // Draw the digital display
  draw();
}
```

Now let's look at the Observer side of the design. The DigitalClock class serves as the concrete observer object. It will maintain a pointer to the Subject it is attached to so that it can invoke the Subject object's methods to get additional information.

When a DigitalClock is instantiated, its constructor attaches the object to the Subject. When a DigitalClock is destroyed, its destructor detaches it from the Subject.

The update() method simply calls the draw() method to update the digital display. The draw method gets the current time from the Subject object and updates the display.

Now, the DigitalClock also inherits from the ClockWidget class, but it is not shown here because our discussion is focused on implementing the Observer-related functionality.

A similar design solution would exist for the AnalogClock concrete observer object.