DESIGN PATTERNS TUTORIAL 1

DISCLAIMER:

I am not a design patterns wizard – I'm learning just like everyone else, and teaching is an ironically excellent way to learn. For a more in depth look into design patterns I have to recommend the book "Design Patterns: Elements of Reusable Object-Oriented Software" by the famous gang of four. Definitely check that out if you're interested in the subject.

Simply put, design patterns are reusable solutions that have been developed to tackle common programming problems. Having a library of these tried and tested solutions can make designing complex, modular applications easier. Each tutorial in this series will look at an individual design pattern, break down its concepts and provide an example of how it could be applied. The examples will have a games slant towards them, showing how each pattern works in a game development context.

OBSERVER PATTERN

An Observer pattern defines a one-to-many interface for objects. That is, a single object that can provide multiple objects with information, without explicitly knowing what the objects are. This promotes loose coupling between classes, reducing class dependencies and making everything more modular – always a good thing.

Anyone familiar with Events and Delegates in C# should recognise this one-to-many feature. You create an Event, and delegates can subscribe/unsubscribe to it. When you call the event it calls all the delegates that have subscribed to it, allowing the objects that containing the delegates to make decisions based on the information.

In a game environment this can reduce wasteful polling of a variable every frame to check their values e.g. having a Player object that contains an integer representing its health. Instead of the HUD constantly checking the health to see if it needs to change what is being drawn, the element that displays the health could "observe" the player object and be notified of when the health changes. This means the Player and the HUD element never need to know the internal workings of each other.

More than one object can also watch for changes to the Player's health. Say there was an animation system that changed how the player moved based on its health. We would still want to keep the workings of the animation abstracted from the Player object (see Model-View-Controller pattern) so the animation system used could also "observe" the Player object. The animation system is updated alongside the HUD element, while the Player object still doesn't need to know anything about either of them.

If anyone has used a Listener system in a game engine like OGRE 3D then this is similar to an Observer pattern, except calling it a "Listener" gives a much more accurate impression of what it does. Objects wait to be told about changes, they don't see the changes as they happen. They rely on the owner of the important

information to tell them of any changes. With calling something an Observer it suggests that it actively watches for any changes, which isn't strictly true, but that's just me being pedantic. :D

PRACTICAL EXAMPLE: PLAYER AND ITS HEALTH

This example is an implementation of the previously discussed Player object notifying other objects of changes made to its health. I'm using C++, so some background knowledge of this language would be beneficial in understanding what's going on. The concept of an Observer pattern is language independent, so even if you don't understand this example hopefully you can still apply the ideas to what you do know.

So, first things first, we need to define an interface that classes can inherit to allow them to observe things. The following abstract class, or interface, defines a simple pure virtual method that takes in a pointer to some user data of any data type. This method will be overridden in all inheriting classes, defining something that will be done with the supplied user data.

```
class IObserver
{
public:

	virtual void Update(void * pUserData) = 0;

};
```

With an observer interface defined we can make a HudElement class that inherits from it. This is what will watch the Player for changes in its health.

```
class HudElement : public IObserver
{
public:

	void Update(void * pUserData)
	{
		int * pHealth = (int*)pUserData;
		cout << "Health is currently at " << *pHealth << "\n";
		Draw();
	}

	void Draw()
	{
		// important drawing stuff here
	}

};
```

Next we need a way of keeping track of all the attached observers to a specific object. This can be encapsulated into a Subject class. A Subject manages the attached observers and triggers their Update methods when Notify is called.

```
class Subject
{
public:

	virtual ~Subject() {}

	// subscribe the observer to the subject
	void Attach(IObserver * pWatcher)
	{
		cout << "\n\n\tAttached new observer!\n\n";
		m_ObserverList.push_back(pWatcher);
	}
	// unsubscribe the observer from the subject
	void Remove(IObserver * pWatcher)
	{
		cout << "\n\n\tRemoved an observer\n\n";
		m_ObserverList.push_back(pWatcher);
	}

protected:

	Subject(){}

	void Notify(void * pUserData)
	{
		list<IObserver*>::iterator Itr = m_ObserverList.begin();

		while (Itr != m_ObserverList.end())
		{
			(*Itr)->Update(pUserData);
			Itr++;
		}
	}

private:

	list<IObserver*> m_ObserverList;


};
```

The Player class should inherit the Subject class. This now means the Player class can have objects observing its contents. When the Health value is set through the accessor method the Notify method from the Subject class is called, supplying the health parameter as the user data.

```
class Player : public Subject
{
public:

	void SetHealth(int iHealth)
	{
		m_iHealth = iHealth;
		Notify(&m_iHealth);
	}

	int GetHealth() { return m_iHealth; }

private:
	int m_iHealth;

};
```

We can see this being used in practice in the following Main function. A player object and healthHudElement object are created, and the first time the health is set nothing happens. After attaching the healthHudElement to the player we can see that when the health is changed the healthHudElement outputs the new health value.

```
int main()
{
	Player player;
	HudElement healthHudElement;

	player.SetHealth(100);
	player.SetHealth(90);

	player.Attach(&healthHudElement);

	player.SetHealth(80);
	player.SetHealth(50);

	cout << "\n";

	system("PAUSE");

	return 1;

}
```

OUTPUT: