Design Patterns

ITP 435 – Spring 2016
Week 4, Lecture 1
Lecturer: Sanjay Madhav

USC Viterbi School of Engineering

University of Southern California

Definition of a Design Pattern

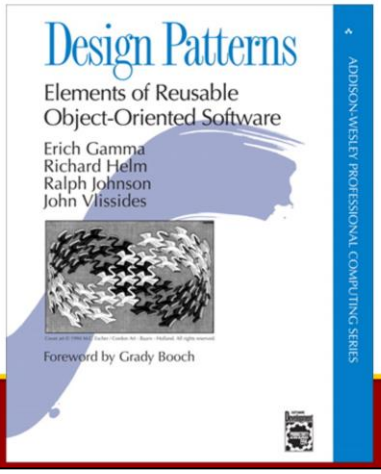
- Short definition:
- "Descriptions of communication objects and classes that are customized to solve a general design problem in a particular context." – Gang of Four

- But people like to quote Christopher Alexander (because he was quoted in Gang of Four, also):

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem."

"Gang of Four" Book

- *De-facto* book for anything you ever wanted to know about Design Patterns
- Describes 23 commonly used patterns. How they're used, how to implement them, and consequences



Singleton

- "Ensure a class only has one instance, and provide a global point of access to it."

- Eg. There would only be one "FileSystem" class in the entire program.
- Ideally, our implementation should stay away from relying on global pointers.
- (Don't do this):

```
FileSystem* g_FileSystem = nullptr;

// Then somewhere else…
g_FileSystem = new FileSystem();
```

Awesome Singleton Implementation


```
template <class T>
class Singleton
{
private:
   static T* _instance;
protected:
   Singleton() {}
public:
   static T& get()
   {
      if (_instance)
      {
         return *_instance;
      }
      else
      {
         _instance = new T();
         return *_instance;
      }
   }
};

template <class T> T* Singleton<T>::_instance = 0;
```

Singleton Usage Example

- Then our FileSystem singleton is declared as follows:

```
class FileSystem : public Singleton<FileSystem>
```

- When you want to use the FileSystem, you would say:

```
FileSystem::get().DoSomething();
```

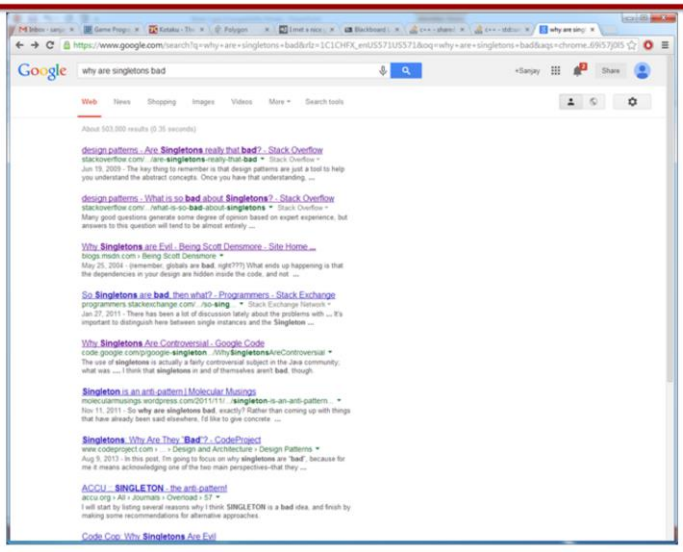
- This will work anywhere, in any file
- It's a roundabout way of having a protected Singleton "interface"

USC Viterbi School of Engineering

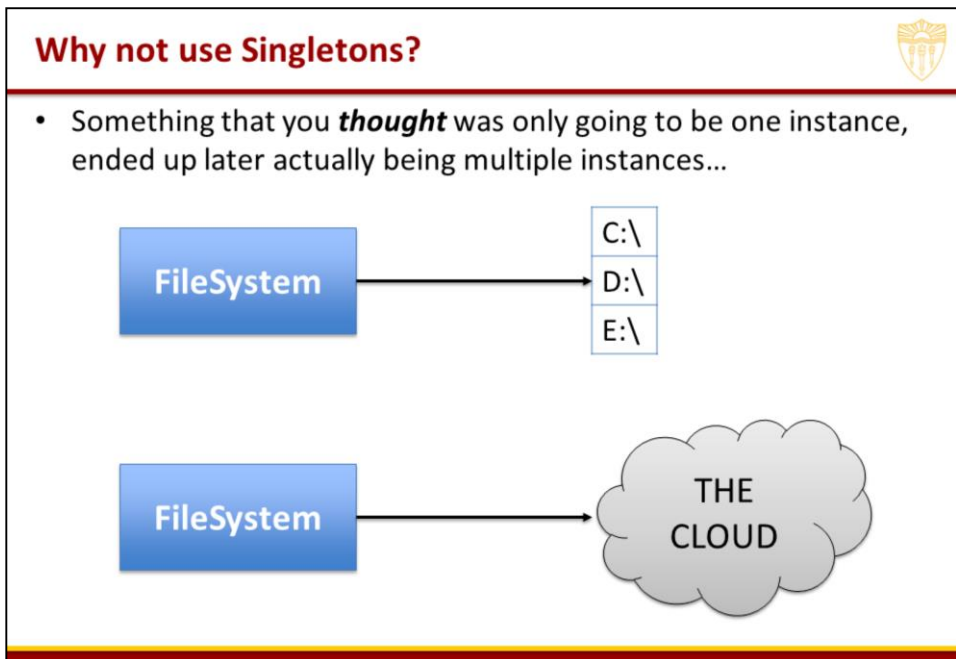
University of Southern California

The bug is that the singleton uses "new" to allocate the instance but never deallocates said instance. But we can live with that since our singleton probably never needs to be deallocated

Singleton Backlash



Why not use Singletons?



- Something that you ***thought*** was only going to be one instance, ended up later actually being multiple instances…

Why not use a Singleton? (cont'd)

- They can hide dependencies:

```
// MyAwesomeClass.h
#pragma once
// No include of FileSystem.h here!
// (no forward declaration either)
class MyAwesomeClass
{

};

// Secretly, in MyAwesomeClass.cpp...
FileSystem::get().GetFile("myawesomefile.txt");
```


This is particularly bad if you want to run automated unit tests on specific modules. It's very difficult because there is an unclear coupling between modules

Singleton Alternatives

- Static methods?

```
class GraphicsDevice
{
public:
   static bool Start();
   static void Draw(float fDeltaTime);
   static void Stop();
};
```

- Advantage: These are maybe a little bit more clear what you're doing
- Disadvantage: You lose the guaranteed single point of instantiation

Singleton Alternatives (cont'd)

- Instead of having LOTS of singletons, have one singleton
- A "service locator"
- "Hey I need a file system!"
  - Either returns a valid service provider OR
  - A "null" service
- A popular alternative because now all the services can be written in a manner that does not assume there's only one instance

My Take on Singletons

- Use singletons if:
  1. You are convinced there's no way you will ever have more than one instance
  2. You truly need a globally accessible instance

- And don't overuse them!

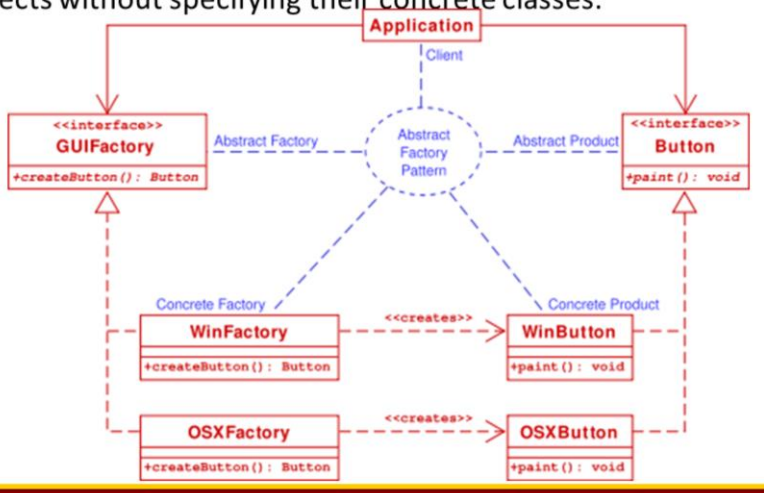
Factory Method

- "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."

```
// Class which makes shapes
class ShapeMaker
{
     static Shape* Create(ShapeType st)
     {
          if (st == TRIANGLE) return new Triangle();
          if (st == SQUARE) return new Square();
          // And so on...
     }
};
```

Abstract Factory

- "Provide an interface for creating families of related or dependent objects without specifying their concrete classes."



Abstract Product and Abstract Factory


```
struct Button
{
    virtual void paint() = 0;
};

struct GUIFactory
{
    virtual Button* createButton() = 0;
};
```

Concrete Product and Concrete Factory


```
struct WinButton : public Button
{
    void paint() { /* Do stuff */ }
};

struct WinFactory : public GUIFactory
{
    Button* createButton() { return new WinButton(); }
};
```

Adapter (aka Wrapper)

- "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."
- Suppose we have a "Shape" class we use in a drawing program
- We want to add a "Shape" for drawing text objects, but we want to use another library for text drawing (TextView)
- So we need an Adapter which takes TextView and makes it conform to Shape

Adapter (Sample Code)


```
struct Shape()
{
    // Shape class we want to use
    virtual void BoundingBox(Point& topLeft, Point& bottomRight) const;
};

struct TextView()
{
    // Text view class for external lib
    void GetOrigin(int x, int y) const;
    void GetExtent(int width, int height) const;
};
```

Adapter (Sample Code), cont'd

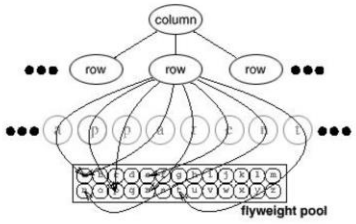

```
struct TextShape : public Shape, private TextView
{
    void BoundingBox(Point& topLeft, Point& bottomRight) const
    {
        int x, y, w, h;
        GetOrigin(x, y);
        GetExtent(w, h);

        topLeft.x = x;
        topLeft.y = y;
        bottomRight.x = x + w;
        bottomRight.y = y + h;
    }
};
```

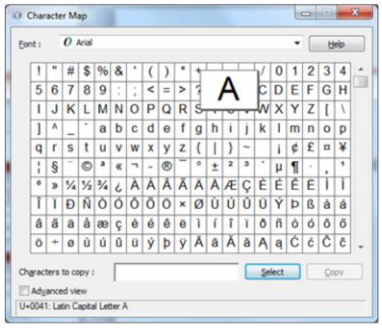
Flyweight

- "Use sharing to support large numbers of fine-grained objects efficiently."

- Example:
- You are writing a word processor and need to display characters over and over again.



Glyphs



- For a particular word, store the short representing each character code, rather than storing the actual glyph image data:

Glyph	H	E	L	L	O
Code	0x48	0x45	0x4C	0x4C	0x4F

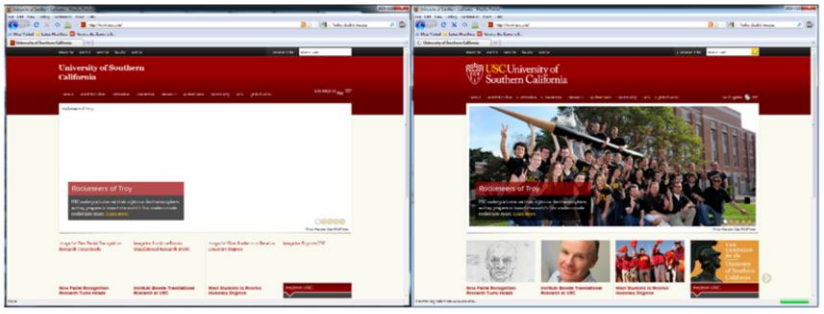
Flyweight Code


```
struct Glyph
{
    virtual void Draw(Window*, Context&);
private:
    short number;
    Image* data;
};

// Store in hash table where the key is the number,
// value is char data.
std::unordered_map<short, Glyphs*> glyphs;
```

Proxy

- "Provide a surrogate or placeholder for another object to control access to it."
- Example: Loading a web page on a slow connection. You use proxies for the image files until you can load them.



With Proxies | With Actual Images

Proxy Code

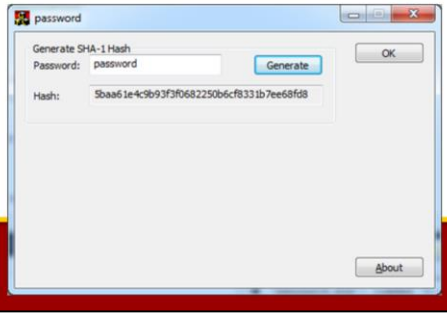

```
struct Image
{
    bool IsLoaded();
    void LoadAsync(std::string fileName, BoundingBox& bounds)
    {
        // Load image on separate thread (without blocking)
    }
    void Draw()
    {
        // Draws this image at location
    }
private:
    BoundingBox m_Bounds;
};
```

Proxy Code, cont'd


```
struct ImageProxy
{
    void Load(std::string fileName, BoundingBox& bounds)
    {
        m_Image.LoadAsync(fileName, bounds);
    }
    void Draw()
    {
        if (m_Image.IsLoaded())
        {
            m_Image.Draw();
        }
        else
        {
            // Draw blank image with appropriate bounding box
        }
    }
private:
    Image m_Image;
    BoundingBox m_Bounds;
};
```

Mediator

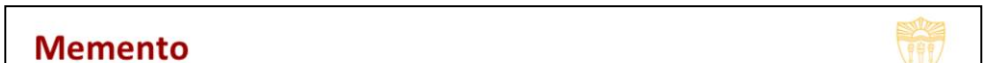
- "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."
- Simple example: Interactions between dialog box widgets.
- Rather than having each button/edit box talk to the others, you have a Mediator (MainDlg class) which converses between the different elements



Mediator (Code)


```
class FontDialog
{
public:
    void OnMessage(Message* msg);
private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

Memento



- "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."
- Commonly used for "Undo" feature

Memento (Code)


```
class State;

class Originator
{
public:
    Memento* CreateMemento();
    void SetMememnto(const Memento*);
private:
    State* _state; // Internal data
};
```

Memento (Code), cont'd

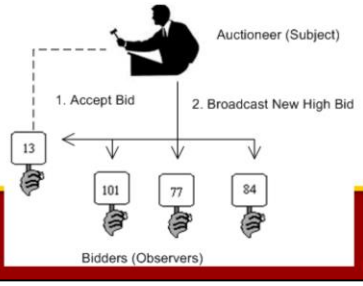

```
class Memento
{
private:
    friend class Originator; // WE ARE FRIENDS OH NOES
    void SetState(State*);
    State* GetState();
// ...
};
```

Observer

- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
- The "view" in model-view-controller

- Example:
- Data which can be displayed as a spreadsheet or chart. If the data changes, the spreadsheet and chart also need to change!

- Example 2:



Observer (Code)


```
class Subject;

class Observer
{
public:
    virtual void Update(Subject* updatedSubject) = 0;
};
```

Observer (Code), cont'd


```
class Subject
{
public:
    virtual void Attach(Observer* o)
    {
        _observers.push_back(o);
    }
    virtual void Detach(Observer* o)
    {
        _observers.remove(o);
    }
    virtual void NotifyObservers()
    {
        for (auto i = _observers.begin(); i != _observers.end(); ++i)
        { i->Update(this); }
    }
private:
    std::list<Observer*> _observers;
};
```

USC Viterbi School of Engineering

University of Southern California

We could also rewrite this to use lambda expressions

Iterator

- "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

- You should know what an iterator is by now

Strategy (aka Policy)

- "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."
- A policy defines multiple ways that a particular action can be done

- For example, suppose you wanted to have different "policies" to describe different ways to construct an object dynamically:
  - With regular new
  - With malloc and then placement new
  - Etc
- With policy classes, you can define multiple ways to do the above, and then choose to use one or the other.

"Creator" Policy


```
template <class T>
struct OpNewCreator
{
     static T* Create()
     {
          return new T;
     }
};

template <class T>
struct MallocCreator
{
     static T* Create()
     {
          void* buf = std::malloc(sizeof(T));
          if (!buf) return 0;
          return new (buf) T; // Placement new
     }
};
```

Using a Policy

- Then we could have a class which uses one or the other policy:

```
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{
    // Use Create function to create Widgets at some point!
};
```

- Then construct an instance of the class using a particular policy:

```
typedef WidgetManager<OpNewCreator<Widget>> MyWidgetMgr;
```

Full List of Official Gang of Four Patterns

Creational Patterns	Proxy
Abstract Factory	**Behavioral Patterns**
Builder	Chain of Responsibility
Factory Method	Command
Prototype	Interpreter
Singleton	Iterator
Structural Patterns	Mediator
Adapter	Memento
Bridge	Observer
Composite	State
Decorator	Strategy
Façade	Template Method
Flyweight	Visitor

Anti-Pattern

- A pattern which may be commonly used, but is ineffective and/or counterproductive in practice.

- Examples:
- Circular Dependency
- Accidental Complexity
- Spaghetti Code
- Copy and Paste Programming
- Poltergeists (wait, what?)

Some people also consider “Singleton” an anti-pattern

Poltergeist Anti-Pattern

- "A short-lived, typically stateless object used to perform initialization or to invoke methods in another, more permanent class."