Design Patterns; Uniform Initializers

ITP 435

Week 9, Lecture 2

Definition of a Design Pattern

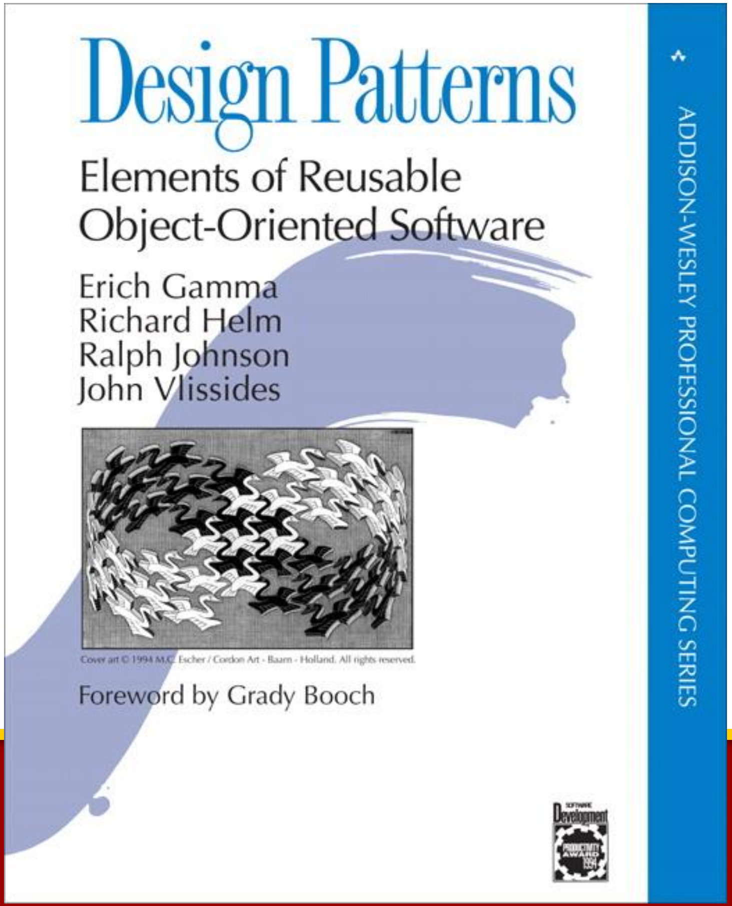
- Short definition:
- "Descriptions of communication objects and classes that are customized to solve a general design problem in a particular context." – Gang of Four

- 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* gFileSystem = nullptr;

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

Awesome Singleton Implementation w/ UniquePtr

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

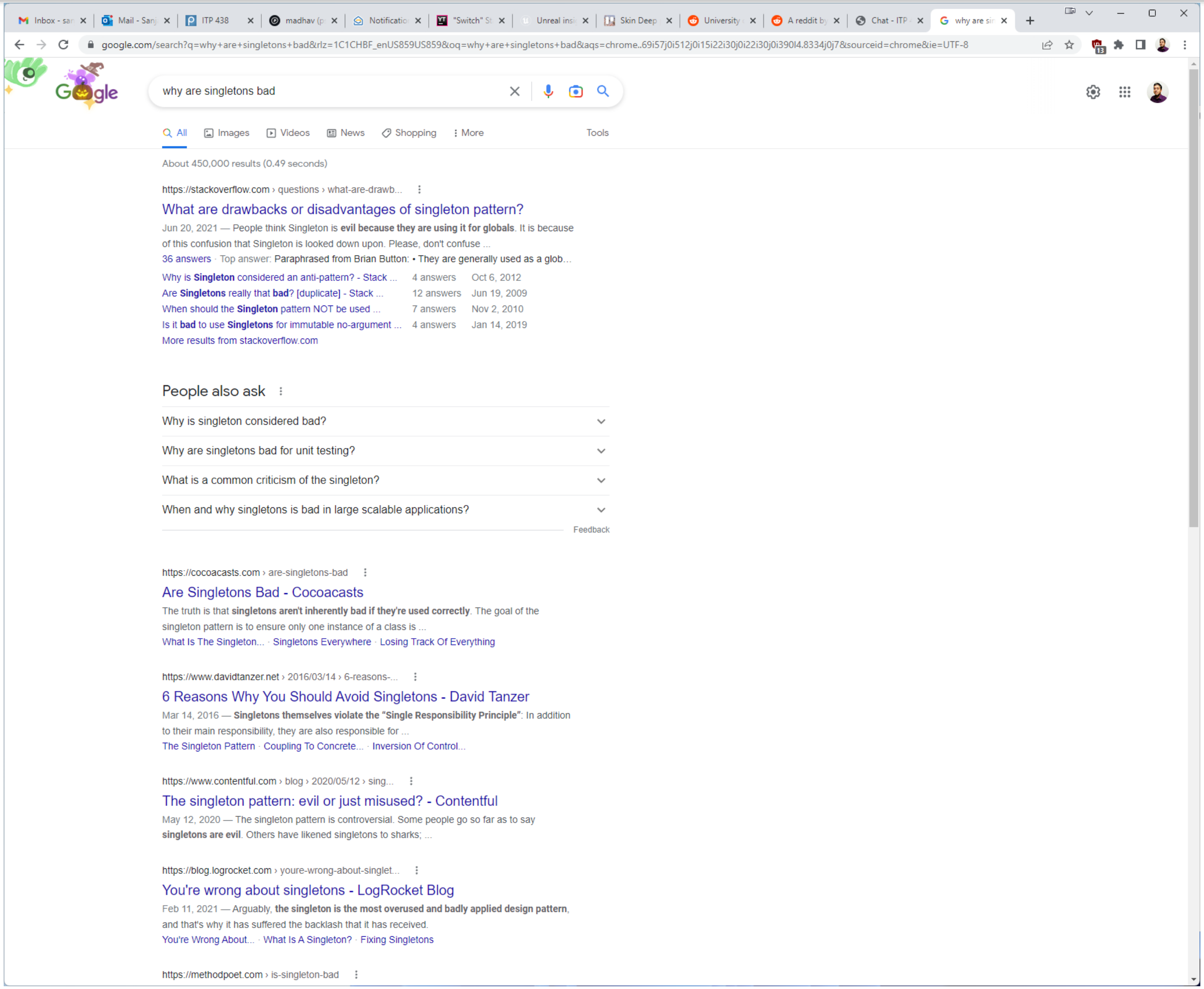
Singleton Usage Example

- 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();
  ```

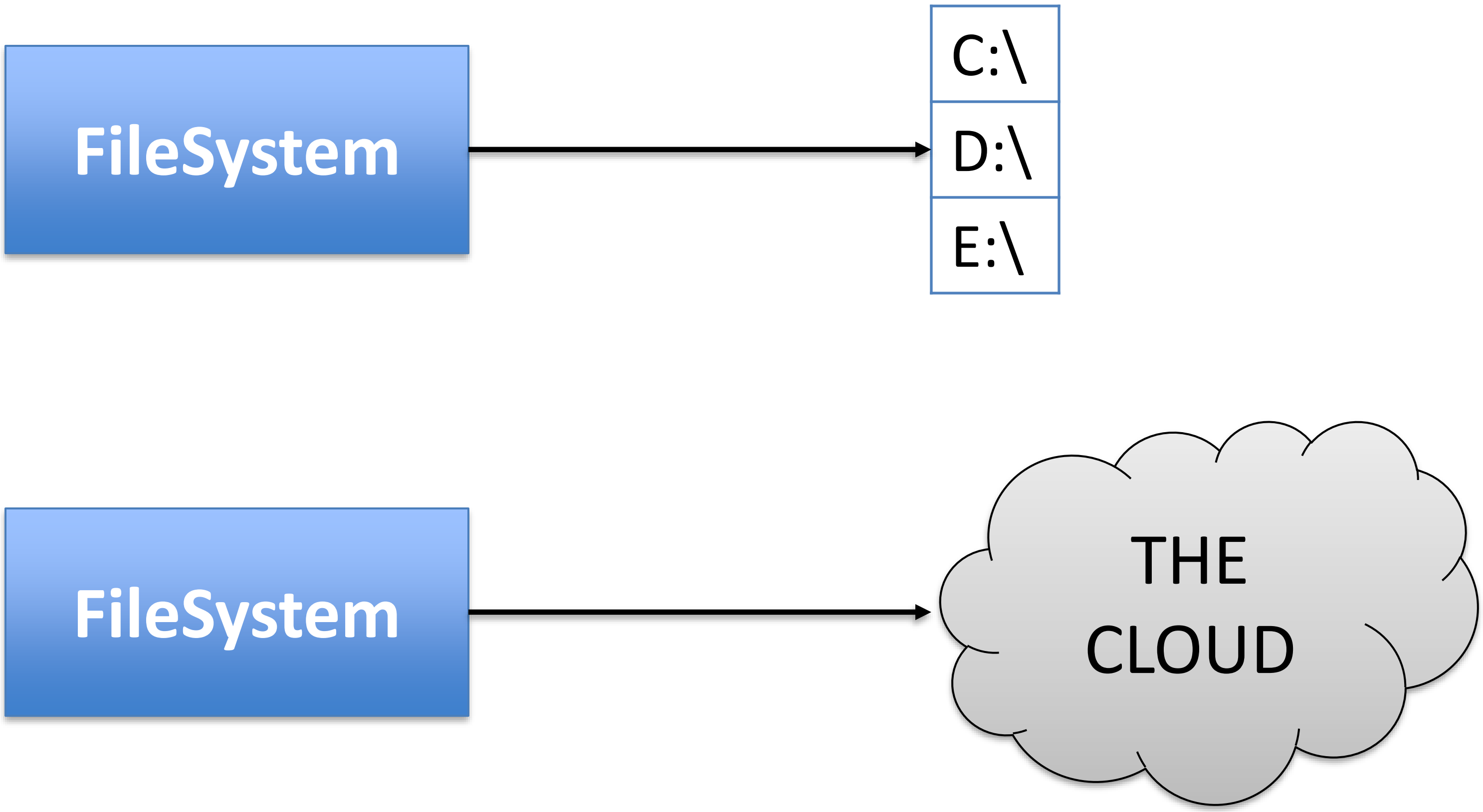
- This will work anywhere, in any file
- If you make FileSystem have a protected default constructor, and make Singleton<FileSystem> a friend, you can prevent anyone else from constructing a FileSystem

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
{


};

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

Singleton Alternatives

- Static methods?

```
class FileSystem
{
public:
    static bool Start();
    static void CreateFile(const std::string& fileName);
    static void Stop();
};
```

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

Dependency Injection

- Make dependencies explicit

```cpp
// MyAwesomeClass.h
#pragma once
class MyAwesomeClass
{
public:
    MyAwesomeClass(class FileSystem& f);
};
```

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!

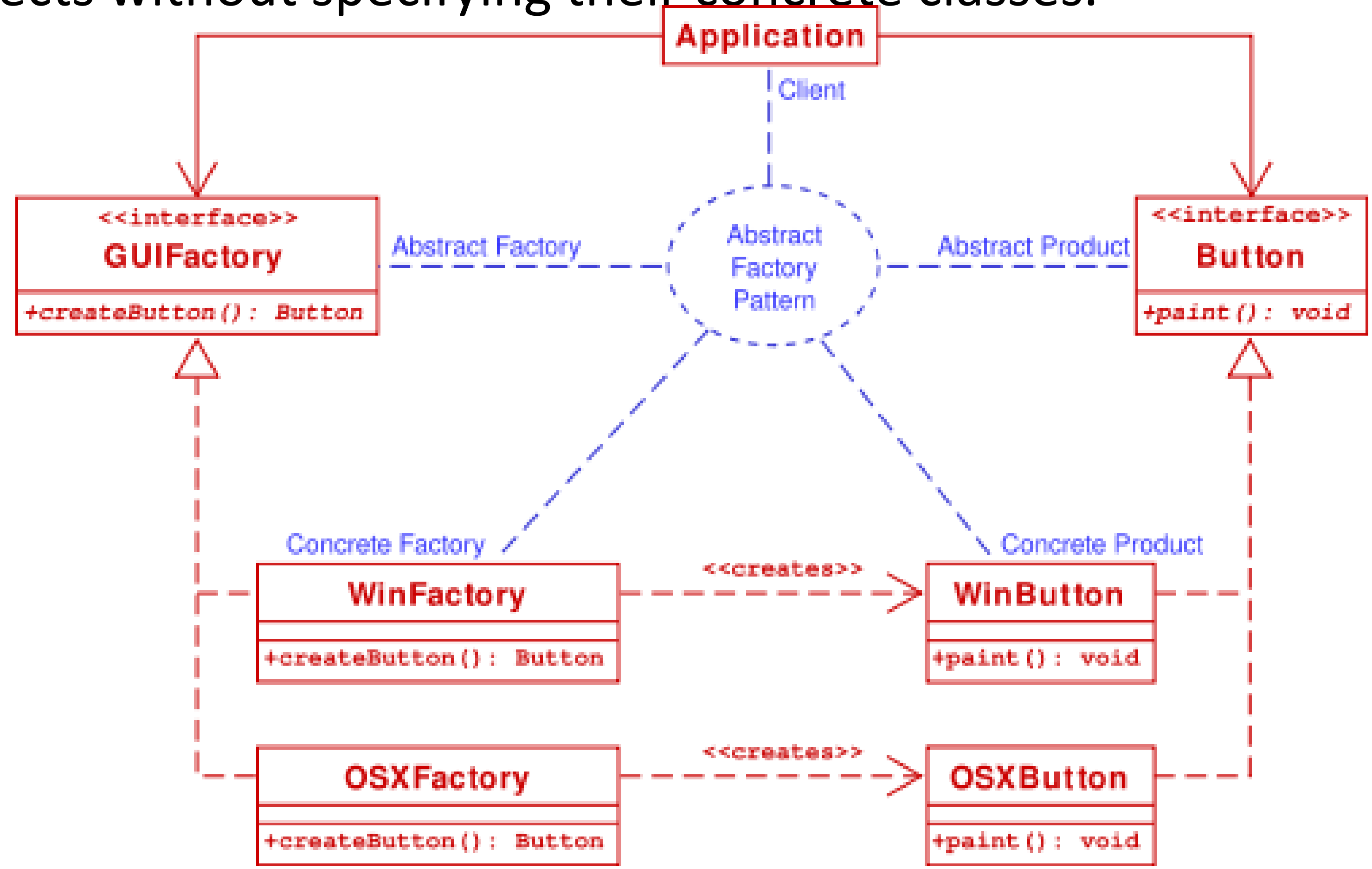
Factory Method

- "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)
    {
        // Imagine there's complicated init code here
        // ...

        if (st == TRIANGLE) return new Triangle( /*stuff*/ );
        if (st == SQUARE) return new Square( /*stuff*/ );
        // 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 = 0;
};

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

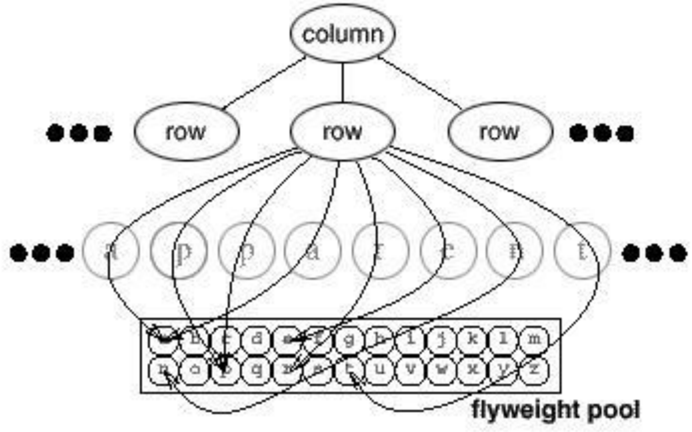
Adapter (Sample Code), cont'd

```
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;
    }
};
```

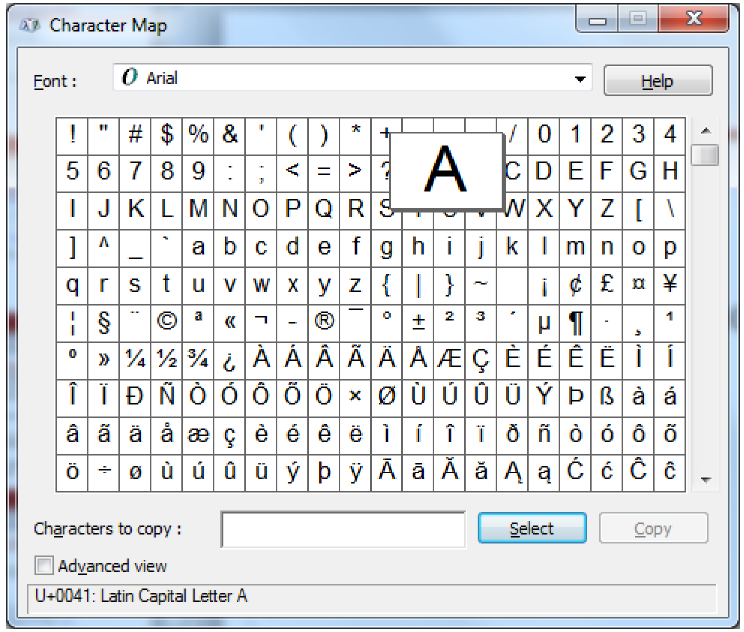
Flyweight

- "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:

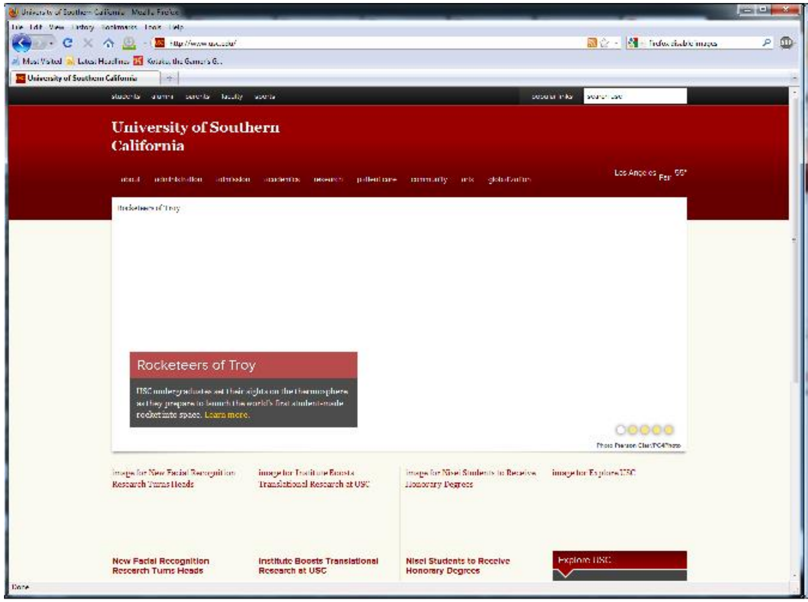
Glyph	H	E	L	L	O
Code	0x48	0x45	0x4C	0x4C	0x4F

Flyweight Code

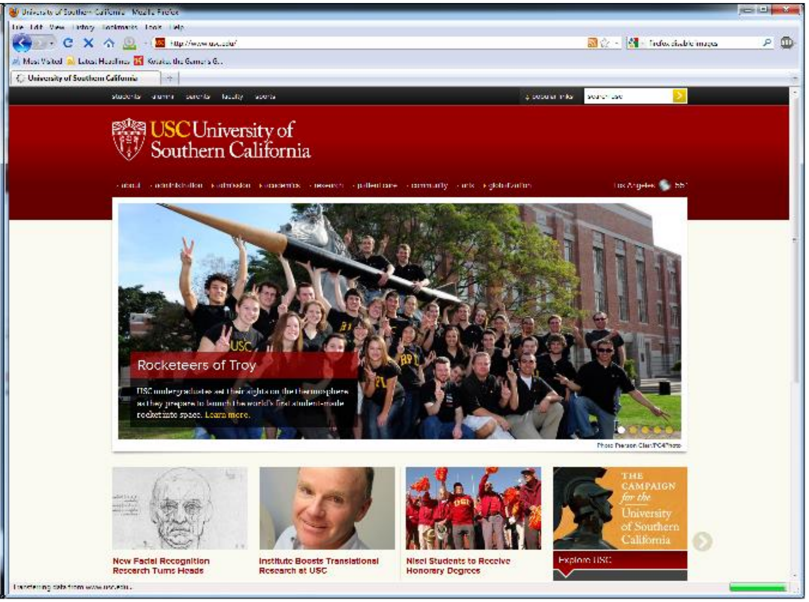
```
struct Glyph
{
    virtual void Draw(Window*, Context&);
private:
    short number;
    Image* data;
};

// 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

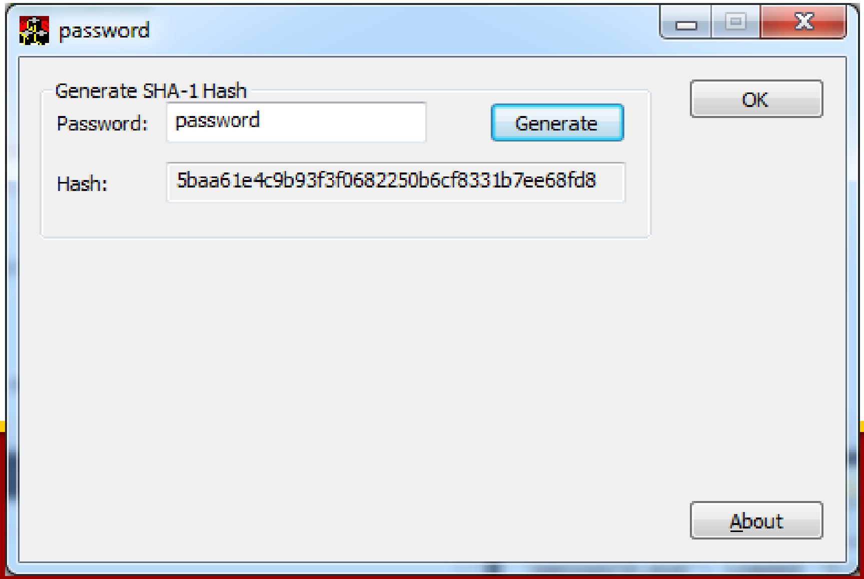
```
struct Image
{
    bool IsLoaded();
    void LoadAsync(std::string fileName, BoundingBox& bounds)
    {
        // Load image on separate thread (without blocking)
    }
    void Draw()
    {
        // Draws this image at location
    }
private:
    BoundingBox mBounds;
};
```

Proxy Code, cont'd

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

Mediator

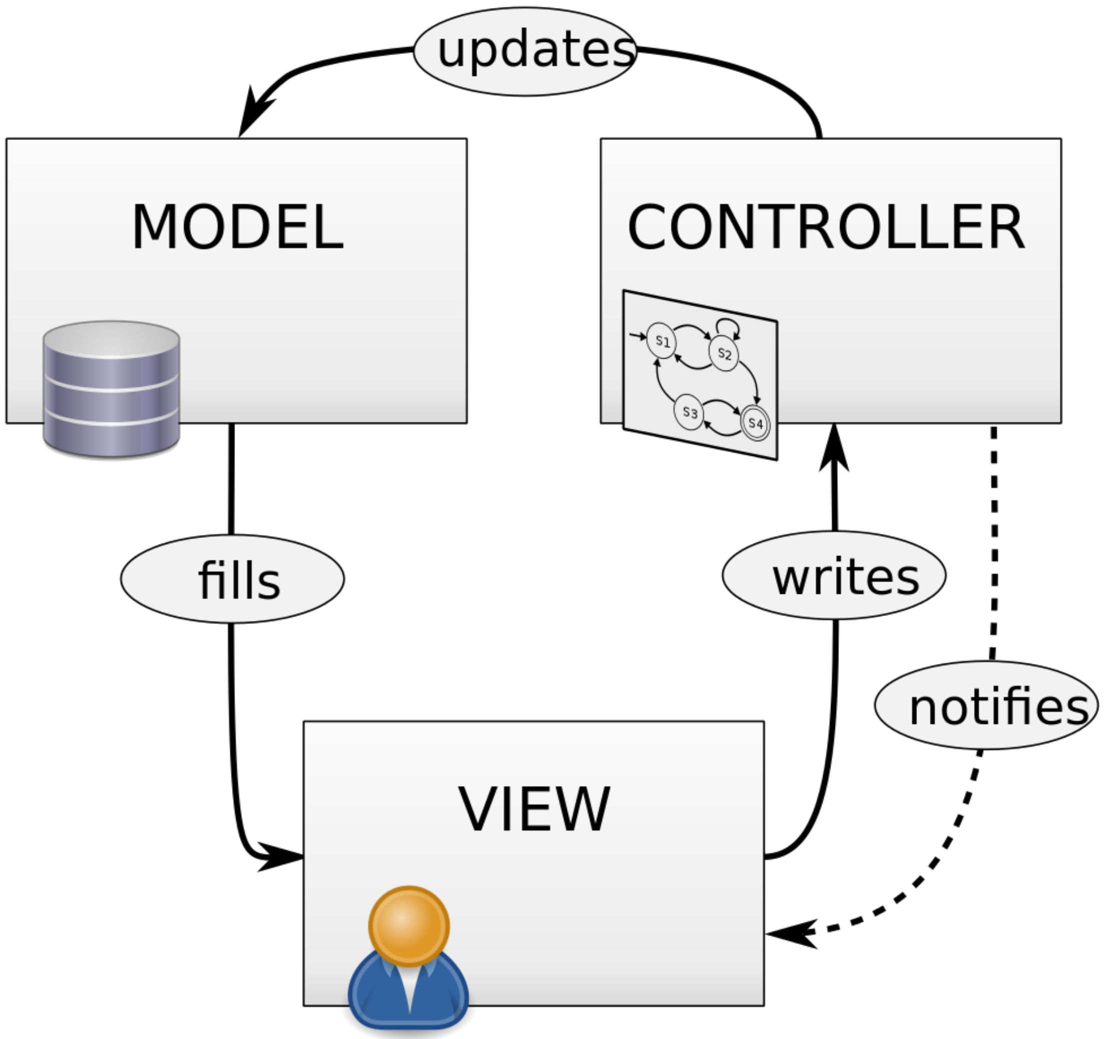
- "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."
- Simple example: Interactions between dialog box widgets.
- Rather than having each 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* mOk;
    Button* mCancel;
    ListBox* mFontList;
    EntryField* mFontName;
};
```

Model/view/controller

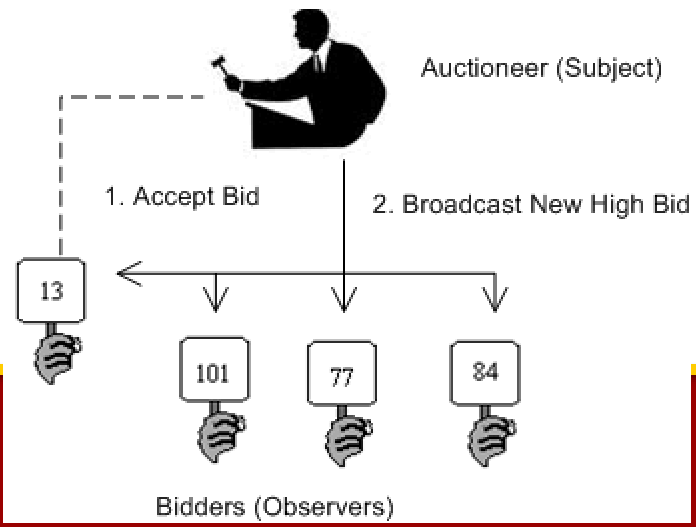


Observer

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

In-class Activity

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?)

Poltergeist Anti-Pattern

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

Uniform Initialization

Basic Initialization in C++11

- There are way too many ways to initialize variables in C++11:

```
int w(0);         // Initializer in parenthesis

int x = 0;        // Initializer after =

int y{ 0 };       // Initializer in braces

int z = { 0 }; // Equals and braces!
```

Problems w/ Basic Initialization

- Not all forms of initialization work in all cases – for example, if the copy constructor is deleted…

```
// Clang error: Copying invokes deleted constructor
std::atomic<int> a1 = 5;

std::atomic<int> a2(5);       // Okay

std::atomic<int> a3{ 5 };     // Okay

std::atomic<int> a4 = { 5 }; // Okay
```

Initializing Non-Static Member Data

- In C++98, the best way to initialize member data is in the initializer list:

```
class Test {
public:
    Test()
        : x(20)
    { }
private:
    int x;
};
```

Initializing Non-Static Member Data, Cont'd

- In C++11, you are allowed to initialize non-static data at the point of declaration – provided you use either equals ***or*** braces (or both):

```
class Test {
public:
private:
   int x = 5;         // Works
   int y = { 25 }; // Works
   int z{ 125 };    // Works
};
```

Initializing Non-Static Member Data, Cont'd

- ***However***, parenthesis syntax is rejected:

```
class Test {
public:
private:
   int x = 5;       // Works
   int y = { 25 }; // Works
   int z{ 125 };   // Works
   int a(125);     // ERROR!
};
```

Uniform Initialization

- ***Uniform initialization*** is the idea of one initialization syntax that is valid wherever initialization is valid

- In order to maintain backwards compatibility in old code, the ***braces initialization*** was selected as the uniform initialization:

```
// This is a uniform initialization
int y{ 0 };

// This is (almost always) also a uniform initialization
int z = { 0 };
```

Difference Between Brace Syntaxes

- = { } will ignore explicit constructors…

```
class Test {
public:
    explicit Test(int i) : x(i) { }
private:
    int x;
};
```

- Then:

```
// Uniform Initialization - Works!
Test t1{ 5 };

// = { } ignores explicit constructors - ERROR! :(
Test t2 = { 5 };
```

Narrowing

- A novel aspect of uniform initialization is that it does not allow ***narrowing*** from a less constrained to a more constrained type

- Examples:

```
char c1{ 50 }; // Fine, 50 can fit in a char
char c2{ 1234 }; // ERROR - Can't narrow 1234 -> char


int i1{ 10 }; // Fine
int t2{ 10.0 }; // ERROR - Can't narrow float -> int
```

Aggregate Initialization

- You can also use the uniform initialization to initialize a class, struct, or union, provided that:
  - There's no private/protected members
  - No constructors (other than default/delete)
  - No base class
  - No virtual functions
  - No brace/equal initializers for non-static members

Aggregate Initialization, Cont'd

```
struct Test1 {
   int x;
   int y;
   int z;
};
```

- You can initialize each member via the uniform initialization (in the order in which they are declared)

```
// Initializes
// x = 50
// y = -50
// z = 25
Test1 t{ 50, -50, 25 };
```

Aggregate Initialization, Cont'd

- Also works when nesting aggregates that satisfy the conditions!

<table>
<tr><th>Declaration</th><th>Initialization</th></tr>
<tr><td>

```
struct Point {
   int x;
   int y;
   int z;
};

struct Test2 {
   Point topLeft;
   Point botRight;
};
```

</td><td>

```
Test2 t2{
   {5, 10, 15}, // Top left
   {2, 4, 6}, // Bottom right
};
```

</td></tr>
</table>

Uniform Initialization and STL Collections

- The great thing about uniform initialization is it works to initialize STL collections!

```
// Initialize a vector of even numbers
std::vector<int> v{ 2, 4, 6, 8, 10 };

// Initialize a list of odd numbers
std::list<int> l{ 1, 3, 5, 7, 9 };

// Create a pair
std::pair<std::string, int> p{ "Hello", 5 };
```

Vector of Pairs

- Works even with nesting!

```
// Vector of pairs
std::vector<std::pair<int, std::string>> months{
   { 1, "January" },
   { 2, "February" },
   { 3, "March" },
   // ...
};
```

- ***Q: How does this actually work?***

std::initializer_list

- Included in the header <initializer_list>
- It's a templated and lightweight proxy class to a temporary array
- Only supports the following operations:
  - Size
  - Begin and end iterators
  - That's it!
- This can be used to create an initializer_list constructor

Example

- Suppose we have a standardish dynamic array:

```
template <typename T>
class DynArray {
public:
   // Functions here ...
private:
   size_t mSize;
   T* mData;
};
```

Example, Cont'd

- Some standard functions:

```
// Default Constructor
DynArray()
: mSize(10)
{
   mData = new T[mSize];
}
// Destructor
~DynArray() {
   delete[] mData;
}
// Constructor to specify initial size
DynArray(size_t size)
: mSize(size)
{
   mData = new T[mSize];
}
```

Example, Cont'd

- The initializer list constructor:

```
// Initializer list constructor
DynArray(const std::initializer_list<T>& list) {
   mSize = list.size();
   mData = new T[mSize];
   int i = 0;
   for (const T& val : list) {
      mData[i] = val;
      i++;
   }
}
```

Example, Cont'd

- Now we can call the initializer_list constructor using the uniform initialization syntax:

```
// Calls the initializer list constructor
DynArray<int> test{ 5, 10, 15, 20, 25 };
```

- A side effect is that if there is an initializer_list constructor, it'll always be preferred when using uniform initialization

Example, Cont'd

- A side effect is that if there is an initializer_list constructor, it'll ***always*** be preferred when using uniform initialization:

```
// This STILL calls the initializer list constructor
// (Creates a list of size 1 with the value 5)
DynArray<int> test2{ 5 };

// To call the constructor that takes the initial size
// only, you have to use the old syntax...
// (Creates a list of size 5 with no data)
DynArray<int> test3(5);
```

- This is the one thing to watch out for when using uniform initialization!

Something that’s annoying…

- Why can’t I just do:

```
// Test 2
someFunctionThatTakesAList({1, 2, 3});
```

- “In Python you could totally do this!” – Random Python programmer

std::initializer_list

- Great way to make a function that can take an arbitrary number of arguments, provided they are all the same type.

```
#include <initializer_list>
int addList(const std::initializer_list<int>& list)
{
   int retVal = 0;
   for (auto i : list)
   {
      retVal += i;
   }

   return retVal;
}
```

- Then later this function can be called like this:

```
// Outputs 33
std::cout << addList({1, 1, 2, 3, 5, 8, 13}) << std::endl;
```

std::initializer_list Constraint

- Remember, it only works if all elements in the list are the same type
- But this should usually be the case…
- And you can use pairs/tuples if you want to pack them in further

std::initializer_list with std::pair

```
void printMonths(const std::initializer_list<std::pair<int,
                    std::string>> & list)
{
   for (auto i : list) {
      std::cout << i.first << ":" << i.second << std::endl;
   }
}

// Later...
printMonths({
   { 1, "January" },
   { 2, "February" },
   { 3, "March" },
   // ...
});
```

A more real use

- I use initializer lists in some of the code in ITP 439:

```
// Returns true if the current token matches one of the tokens
// in the list.
bool Parser::peekIsOneOf(const std::initializer_list<Token::Tokens>& list)
   noexcept
{
   for (Token::Tokens t : list)
   {
      if (t == peekToken())
      {
         return true;
      }
   }
   return false;
}
```