Software Design using “Design Patterns”

Part 1 Introduction and Creational Patterns
Gang of Four

- GOF: "Gang of Four", ie Gamma, Helm, Johnson, Vlissides, authors of "Design Patterns: Elements of Reusable Object-Oriented Software"

- Classic text on OO Design

- Two main considerations
  - Program to an 'interface', not an 'implementation'.
  - Favour ‘object composition’ over ‘class inheritance’
Design Patterns

- Patterns are highly dependent on the language
- C++ is very limited – forcing us to use those
- Some API’s are designed around patterns (C# and Objective C use them a lot)
- We can break patterns into the following taxonomy
“The “Design Patterns” book is one of the worst programming books ever. Yes, really. I’m 100% dead serious when I say that I think it has set (and will continue to set) the progress of software development back by decades”

http://realtimecollisiondetection.net/blog/?p=44
Critique

“Design patterns are spoon-fed material for brainless programmers incapable of independent thought, who will be resolved to producing code as mediocre as the design patterns they use to create it.

Design patterns really are from hell!”

http://realtimecollisiondetection.net/blog/?p=81
Pattern Types

• Creational
• Structural
• Behavioural
• Concurrency pattern (for threading)
The singleton Pattern

• The singleton pattern defines an object that can only exist once

• This is done by implementing the code in a particular way with the object knowing if it has been created.

• If it has not it will create an instance of itself

• If it has been created it will return the instance.

• This is sometimes thought of an an “anti-pattern” should be used with care.
The singleton pattern

- Deletion can be difficult
- Modern C++ gives answers to all of this through a complex template framework
- In doubt, use a static member
- Problematic with multi-threading, must use the “double check lock paradigm”
Singleton* Singleton::instance()
{
    // using the Double Check locking pattern to make thread safe
    if (s_instance == 0)
    {
        // create a mutex to stop other threads accessing
        QMutex m;
        // the locker will auto unlock when out of scope
        QMutexLocker locker(&m);
        if (s_instance == 0)
        {
            s_instance = new Singleton();
        }
    }

    return s_instance;
}
extern "C++" int atexit( void (*func)() );

- Another way to handle singleton deletion is to use the `std::atexit()` signal handeller
- Registers the function pointed to by `func` to be called on normal program termination (via `std::exit()` or returning from the cpp/language/main function)
- Can use this to clear singleton resources (see ngl/Singleton.h template for example)
C++ 11 singleton

```cpp
CollisionShape *CollisionShape::instance()
{
    // in C++ 11 we use this to avoid having to do a double check lock pattern
    static CollisionShape s_instance;

    return &s_instance;
}
```

C++ 11 removes the need for manual locking. Concurrent execution shall wait if a static local variable is already being initialised.
The monostate pattern

• A monostate is a "conceptual singleton" is uses static data members and functions

• This means that all instances of the class are accessing the same static data

• The following example reads a config file and stores the values in static class attributes
#ifndef CONFIG_H__
#define CONFIG_H__
#include <string>

class Config
{
    public:
        Config();
        inline int getPointSize() const {return m_pointSize;}
        inline std::string getColour() const {return m_colour;}
        inline std::string getDrawMode() const {return m_drawMode;}

    private:
        static bool m_configRead;
        static int m_pointSize;
        static std::string m_colour;
        static std::string m_drawMode;

};

#endif
bool Config::m_configRead=false;
std::string Config::m_colour="";
std::string Config::m_drawMode="";
int Config::m_pointSize=1;

Config::Config()
{
    if (m_configRead == false)
    {
        m_configRead=true;
        std::fstream fileIn;
        fileIn.open("config.txt",std::ios::in);
        fileIn >> m_colour;
        fileIn >> m_pointSize;
        fileIn >> m_drawMode;
        fileIn.close();
    }
}
```cpp
#include "Config.h"
#include <iostream>

void draw()
{
    Config m;
    std::cout<<"in_draw_"<<m.getPointSize()<<"\n";
}

int main()
{
    Config m;
    std::cout<<"in_main_"<<m.getColour()<<"\n";
    draw();
    std::cout<<"draw_mode"<<m.getDrawMode()<<"\n";
}```
Factory Methods

- A factory method is a creational design pattern that allows you to create an object without having to specify the specific C++ type of the object.
- It can be thought of as a generalisation of a constructor.
- This helps to overcome some of the issues with C++ constructors.
The problem with ctors

• ctors have no return type, it makes it difficult to signal an error by returning a NULL type

• we have constrained naming. It always has the same name as the class, and we can only have one set of parameter lists (i.e. can’t have two ctors with single int param lists that do different things)

• No dynamic runtime binding of ctors, i.e. we must know the concrete class at compile time.

• No virtual constructors.
Factory methods

• At the basic level a factory method is a normal method that can return an instance of a class.

• However if we use it in conjunction with inheritance we can override the default class behaviour and return related class types.

• This is usually done in conjunction with abstract base classes.
Abstract Base Class

• An ABC is a class that contains one or more “pure virtual” member functions.

• This is not a “concrete” class and means that it can’t be instantiated using the new operator

• Instead it is a base class where derived classes provide the implementation of the pure virtual methods

• The following examples are from API Design for C++ by Martin Reddy and show a simple renderer interface.
class Renderer
{
public:
    virtual ~Renderer() {}
    virtual bool loadScene(const std::string &filename) = 0;
    virtual void setViewportSize(int w, int h) = 0;
    virtual void setCameraPos(double x, double y, double z) = 0;
    virtual void setLookAt(double x, double y, double z) = 0;
    virtual void render() = 0;
};
```cpp
class RenderFactory
{
public:
    Renderer *createRenderer(const std::string &_type);
};

Renderer *RenderFactory::createRenderer(const std::string &_type)
{
    if(_type=="OpenGL")
    {
        return new OpenGLRenderer();
    }

    else if(_type=="DirectX")
    {
        return new DirectXRenderer();
    }
    else return 0;
}
```
```cpp
#include <iostream>
#include "RenderFactory.h"

int main()
{
    // create the factory object
    RenderFactory *f = new RenderFactory;

    // create an OpenGL renderer
    Renderer *ogl = f->createRenderer("OpenGL");
    ogl->render();
    delete ogl;

    Renderer *directX = f->createRenderer("DirectX");
    DirectX->render();
    delete DirectX;

    delete f;

    Renderer *unknown = f->createRenderer("raytracer");
    if (unknown == 0)
    {
        std::cout<<"don't know how to create a raytracer\n";
    }

    return EXIT_SUCCESS;
}
```
Extensible Factories

- This further decouples the concrete classes from the methods
- We need to maintain a map of registered creation methods within the creator
- This now means that the extensible factory holds state data
- Usually this means they are created using a singleton so only one can ever exist
```cpp
#include "Renderer.h"
#include <string>
#include <map>

class RendererFactory
{
public:
    // The type for the callback that creates an IRenderer instance
    typedef Renderer *(const createCallback cb());

    // Add a new 3D renderer to the system
    static void registerRenderer(const std::string &type, createCallback cb);

    // Remove an existing 3D renderer from the system
    static void unregisterRenderer(const std::string &type);

    // Create an instance of a named 3D renderer
    static Renderer *createRenderer(const std::string &type);

private:
    static std::map<std::string, createCallback> m_renderers;
};
```
#include "RenderFactory.h"
#include "DirectXRenderer.h"
#include "OpenGLRenderer.h"

// instantiate the static variable in RendererFactory
std::map<std::string, RendererFactory::createCallback> RendererFactory::_renderers;

void RendererFactory::registerRenderer(const std::string &type, createCallback cb)
{
    _renderers[type] = cb;
}

void RendererFactory::unregisterRenderer(const std::string &type)
{
    _renderers.erase(type);
}

Renderer *RendererFactory::createRenderer(const std::string &type)
{
    std::map<std::string, createCallback>::iterator it = _renderers.find(type);
    if (it != _renderers.end())
    {
        // call the creation callback to construct this derived type
        return (it->second)();
    }
    return NULL;
}
#include <iostream>
#include "RenderFactory.h"
#include "OpenGLRenderer.h"
#include "DirectXRenderer.h"

int main()
{
    // register the various 3D renderers with the factory object
    RendererFactory::registerRenderer("opengl", OpenGLRenderer::create);
    RendererFactory::registerRenderer("DirectX", DirectXRenderer::create);
    // create an OpenGL renderer
    Renderer *ogl = RendererFactory::createRenderer("opengl");
    ogl->render();
    delete ogl;
    // create a DirectX software renderer
    Renderer *DirectX = RendererFactory::createRenderer("DirectX");
    DirectX->render();
    delete DirectX;
    // unregister the DirectX renderer
    RendererFactory::unregisterRenderer("DirectX");
    DirectX = RendererFactory::createRenderer("DirectX");
    if (!DirectX)
    {
        std::cout << "DirectX renderer unregistered" << std::endl;
    }
    return EXIT_SUCCESS;
}
Lazy initialisation

- Saves memory
- Delay object creation until needed
  - Textures
  - Geometries
- Can be combined with a Factory to create “Lazy Factories”
- Can be implemented as the “Multiton pattern”

<table>
<thead>
<tr>
<th>Multiton</th>
</tr>
</thead>
<tbody>
<tr>
<td>- instances : std::map&lt;Key,Multiton&gt;</td>
</tr>
<tr>
<td>- Multiton()</td>
</tr>
<tr>
<td>+ getInstance() : Multiton</td>
</tr>
</tbody>
</table>
Simple Texture Factory

- In this case we will use static attributes to contain a map of std::string to Texture * objects.
- If the name key exists the instance will be returned.
- If not a new object will be created and stored in the map.
class Texture
{
public:
    static Texture* getTexture(const std::string &_type);
    // just to demo the process
    static void printCurrentTexture();

private:
    static std::map<std::string, Texture*> m_textures;
    // the type of this texture (i.e. the name)
    std::string m_name;

    // note: constructor private forcing one to use static getTexture()
    Texture(const std::string &_t) : m_name(_t) {}
};
/now we can use the static member variable
std::map<std::string, Texture*> Texture::m_textures;

Texture* Texture::getTexture(const std::string& _type)
{
    // try to find an existing instance; if not found std::map will return
    // types.end()

    std::map<std::string, Texture*>::iterator it = m_textures.find(_type);
    Texture* t;
    if (it == m_textures.end())
    {
        // if no instance with the proper type was found, make one
        t = new Texture(_type); // lazy initialization part
        m_textures[_type] = t;
    }
    else
    {
        // if already had an instance
        t = it->second; // The return value will be the found texture
    }
    return t;
}
#include "Texture.h"
#include <cstdlib>

int main()
{
    Texture::getTexture("diffuse.tga");
    Texture::printCurrentTexture();
    Texture::getTexture("specular.tga");
    Texture::printCurrentTexture();
    Texture::getTexture("diffuse.tga");
    Texture::printCurrentTexture();

    return EXIT_SUCCESS;
}
The prototype pattern

- This pattern uses a prototypical instance to clone new instances of an object.

- This helps to
  - avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
  - avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.
// Prototype

class Prototype
{
    public:
        virtual ~Prototype() {}

        virtual Prototype* clone() const = 0;
};
Concrete Prototype

// Concrete prototype
class ConcretePrototype : public Prototype
{
public:
    ConcretePrototype(int _x) : m_x(_x) { }

    ConcretePrototype(const ConcretePrototype& _p) : m_x(_p.m_x) { }

    virtual Prototype* clone() const { return new ConcretePrototype(*this); }

    void setX(int _x) { m_x = _x; }

    int getX() const { return m_x; }

    void printX() const { std::cout << "Value:" << m_x << std::endl; }

private:
    int m_x;
};
Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself.

```cpp
#include "Prototype.h"
int main()
{
    Prototype* prototype = new ConcretePrototype(1000);
    for (int i = 1; i < 10; ++i)
    {
        ConcretePrototype* tempotype = dynamic_cast<ConcretePrototype*>(prototype->clone());
        tempotype->setX(tempotype->getX() * i);
        tempotype->printX();
        delete tempotype;
    }
    delete prototype;

    return EXIT_SUCCESS;
}
```
Dependancy Injection

- Dependancy injection is a technique where an object is passed into a class (injected) instead of having a class create and store the object itself.

- This means the class containing the object doesn’t need to know how to create the object.

- This is know as “inversion of control”

- Makes it easier to implement “unit testing” as we can inject test cases into our classes

- Decouples “concrete” from “concrete”
Types of injection

• Type 1 or interface injection, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.

• Type 2 or setter injection, in which the dependent module exposes a setter method that the framework uses to inject the dependency.

• Type 3 or constructor injection, in which the dependencies are provided through the class constructor.
The Law of Demeter

- The Law of Demeter for functions (methods) attempts to minimise coupling between modules in any given program.
- It tries to prevent you from reaching into an object to gain access to a third object’s methods.
- By writing “shy” code that honours the Law of Demeter as much as possible, we can achieve the goal of minimising coupling between modules.
• “The pragmatic programmer” Ch. 5 has a good discussion of this

```cpp
class Demeter {
private:
    A *a;
    int func();
public:
    //...
    void example(B& b);
};
void Demeter::example(B& b) {
    C c;
    int f = func(); // itself
    b.invert(); // any parameters that were passed in to the method
    a = new A();
    a->setActive(); // any objects it created
    c.print(); // any directly held component objects
}
```
The Builder Pattern

- The Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.

- Helps to eliminate “telescoping constructors”

```java
Pizza(int size) { ... }
Pizza(int size, boolean cheese) { ... }
Pizza(int size, boolean pepperoni) { ... }
Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon, int price, int) { ... }
```

- Helps to eliminate the use of setters and getters after object construction

```java
Pizza pizza = new Pizza(12);
pizza.setCheese(true);
pizza.setPepperoni(true);
pizza.setBacon(true);
```
Builder Pattern

- Define an intermediate object whose member functions define the desired object part by part.
- Build Pattern lets us defer the construction of the object until all the options for creation have been specified.

```
for all object in structure
  builder->buildPart()
```
new concreteBuilder()

new Director(concreteBuilder)

Construct()

buildA()

buildB()

buildC()

getResult()
Pizza Example

• This is a classic Builder pattern example
• It is used to illustrate the key concepts in the Builder pattern
• We start with a basic Pizza class
class Pizza
{
  public:
  void setDough(const std::string &_dough)
  {
    m_dough = _dough;
  }
  void setSauce(const std::string &_sauce)
  {
    m_sauce = _sauce;
  }
  void setTopping(const std::string &_topping)
  {
    m_topping = _topping;
  }
  void open() const
  {
    std::cout << "Pizza with " << m_dough << " dough, " << m_sauce << " sauce and "
               << m_topping << " topping.\n";
  }
  private:
  std::string m_dough;
  std::string m_sauce;
  std::string m_topping;
};
Abstract Builder

```cpp
#include "Pizza.h"
// "Abstract Builder"

class PizzaBuilder
{
public:
    Pizza* getPizza()
    {
        return m_pizza;
    }
    void createNewPizzaProduct()
    {
        m_pizza = new Pizza;
    }
    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void buildTopping() = 0;
protected:
    Pizza* m_pizza;
};
```
Concrete Builders

```cpp
class HawaiianPizzaBuilder : public PizzaBuilder
{

public:
  virtual void buildDough()
  {
    m_pizza->setDough("cross");
  }
  virtual void buildSauce()
  {
    m_pizza->setSauce("mild");
  }
  virtual void buildTopping()
  {
    m_pizza->setTopping("ham+pineapple");
  }
};

class SpicyPizzaBuilder : public PizzaBuilder
{

public:
  virtual void buildDough()
  {
    m_pizza->setDough("pan_baked");
  }
  virtual void buildSauce()
  {
    m_pizza->setSauce("hot");
  }
  virtual void buildTopping()
  {
    m_pizza->setTopping("pepperoni+salami");
  }
};
```
class Cook
{
public:
    void setPizzaBuilder(PizzaBuilder* pb)
    {
        m_pizzaBuilder = pb;
    }
    Pizza* getPizza()
    {
        return m_pizzaBuilder->getPizza();
    }
    void constructPizza()
    {
        m_pizzaBuilder->createNewPizzaProduct();
        m_pizzaBuilder->buildDough();
        m_pizzaBuilder->buildSauce();
        m_pizzaBuilder->buildTopping();
    }
private:
    PizzaBuilder* m_pizzaBuilder;
};
Using the classes

```cpp
#include "PizzaBuilder.h"
#include "Cook.h"
#include "HawaiianPizzaBuilder.h"
#include "SpicyPizzaBuilder.h"
int main()
{
    Cook cook;
    PizzaBuilder* hawaiianPizzaBuilder = new HawaiianPizzaBuilder;
    PizzaBuilder* spicyPizzaBuilder = new SpicyPizzaBuilder;

    cook.setPizzaBuilder(hawaiianPizzaBuilder);
    cook.constructPizza();

    Pizza* hawaiian = cook.getPizza();
    hawaiian->open();

    cook.setPizzaBuilder(spicyPizzaBuilder);
    cook.constructPizza();

    Pizza* spicy = cook.getPizza();
    spicy->open();

    delete hawaiianPizzaBuilder;
    delete spicyPizzaBuilder;
    delete hawaiian;
    delete spicy;
}
```
Object Pools

• This is a creation / optimisation pattern

• Intent
  • “Improve performance and memory use by reusing objects from a fixed pool instead of allocating and freeing them individually.”
  • Allows use to use contiguous memory on the heap and reduce allocations which may be expensive.
  • Reduces memory fragmentation
Fragmentation

- Fragmentation means the free space in our heap is broken into smaller pieces of memory instead of one large open block.

http://gameprogrammingpatterns.com/object-pool.html
Object Pool

• Define a pool of re-usable objects that you need to frequently create and destroy

• Objects are similar in size.

• Allocating objects on the heap is slow or could lead to memory fragmentation.

• Each object encapsulates a resource such as a database or network connection that is expensive to acquire and could be reused.

• For creating threads (we will cover thread pools next year)
Object Pool

- The pool may waste memory on unneeded objects
- We can only have a fixed amount of objects at any one time
- Memory size for objects is fixed (however with pointers we can do other tricks but will defeat the object of the pool)
- Reused objects are not cleared and will remain in memory.
Particle

class Particle
{
public:
    Particle() : m_framesLeft(0) {}

    void init(double _x, double _y, double _xVel, double _yVel, int _lifetime);
    void animate();
    bool inUse() const { return m_framesLeft > 0; }

private:
    int m_framesLeft;
    double m_x, m_y;
    double m_xVel, m_yVel;
};

void Particle::init(double _x, double _y, double _xVel, double _yVel, int _lifetime)
{
    m_x = _x;
    m_y = _y;
    m_xVel = _xVel;
    m_yVel = _yVel;
    m_framesLeft = _lifetime;
}

void Particle::animate()
{
    if (!inUse())
    {
        return;
    }
    m_framesLeft--;
    m_x += m_xVel;
    m_y += m_yVel;
    std::cout << "Particle\"" << m_x << "," << m_y << std::endl;
}
#include "Particle.h"

class ParticlePool
{
public:
    void create(double _x, double _y, double _xVel, double _yVel, int _lifetime);
    void animate();

private:
    static const int POOL_SIZE = 100;
    Particle m_particles[POOL_SIZE];
};

void ParticlePool::animate()
{
    for (int i = 0; i < POOL_SIZE; i++)
    {
        m_particles[i].animate();
    }
}

void ParticlePool::create(double _x, double _y, double _xVel, double _yVel, int _lifetime)
{
    // Find an available particle.
    for (int i = 0; i < POOL_SIZE; i++)
    {
        if (!m_particles[i].inUse())
        {
            m_particles[i].init(_x, _y, _xVel, _yVel, _lifetime);
            return;
        }
    }
}
This works ok, however we do need to iterate through the pool to find the first free particle.

This can be improved with a free store for more discussion see the game programming patterns book.
The PIMPL Idiom

• Introduced by Jeff Sumner as shorthand for “Pointer to Implementation”

• It is used to avoid exposing private details in an API header

• Whilst it is not a design pattern may be considered a special case of the “Bridge pattern”

• Main reason for use is to overcome C++ limitations
In C++, we can forward declare a pointer to a type, i.e., we declare a name but not the details. This is sometimes referred to as an “opaque pointer” as you can’t see the detail.
Example “Autotimer”

- This example is from C++ API Design (Reddy 2011)
- This first example shows a design where implementation detail is visible in the headers
- It has OS specific defines and includes
- All of the code is in the header
- Is hard to maintain and read
/** modified from Marting Reddy example **

```cpp
#ifndef AUTOTIMER_H
#define AUTOTIMER_H

#include <string>
// Bad: only need iostream because of implementation code
#include <iostream>

// Bad: exposes platform specifics in your header file
#ifdef WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

class AutoTimer
{
public:
    // Bad: exposes (and inlines) implementation details
    explicit AutoTimer(const std::string &name) :
        mName(name)
    {
        #ifdef _WIN32
            mStartTime = GetTickCount();
        #else
            gettimeofday(&mStartTime, NULL);
        #endif
    }

    ~AutoTimer()
    {
        std::cout << mName << " took " << GetElapsed() << " secs " << std::endl;
    }

    // Bad: no need to expose this function publicly
    double GetElapsed() const
    {
        #ifdef _WIN32
            return (GetTickCount() - mStartTime) / 1e3;
        #else
            struct timeval end_time;
            gettimeofday(&end_time, NULL);
            double t1 = mStartTime.tv_usec / 1e6 + mStartTime.tv_sec;
            double t2 = end_time.tv_usec / 1e6 + end_time.tv_sec;
            return t2 - t1;
        #endif
    }

    // Bad: data members should always be private
    std::string mName;
    #ifdef _WIN32
        DWORD mStartTime;
    #else
        struct timeval mStartTime;
    #endif
};
#endif
```
PIMPL Version

- In this version we pre-declare a class called Impl which we will declare in the header
- There will be no detail of this just the declaration
- We also create a pointer to the Impl class and use this
```cpp
/// modified from the martin Reddy book
///
#
define AUTOTIMER_H
#include <string>

/// An object that reports how long it was alive for when it
/// is destroyed.
///
class AutoTimer
{
    public:
    /// Create a new timer object with a human-readable name
    explicit AutoTimer(const std::string &name);
    /// On destruction, the timer reports how long it was alive
    ~AutoTimer();

    private:
    /// Make this object be noncopyable because it holds a pointer
    AutoTimer(const AutoTimer &);
    const AutoTimer &operator =(const AutoTimer &);

    class Impl;
    Impl *m_impl;
};

#endif
```
```cpp
class AutoTimer::Impl
{
public:
    // Return how long the object has been alive
    double GetElapsed() const
    {
        #ifdef _WIN32
            return (GetTickCount() - mStartTime) / 1e3;
        #else
            struct timeval end_time;
            gettimeofday(&end_time, NULL);
            double t1 = mStartTime.tv_usec / 1e6 + mStartTime.tv_sec;
            double t2 = end_time.tv_usec / 1e6 + end_time.tv_sec;
            return t2 - t1;
        #endif
    }

    std::string mName;
    #ifdef _WIN32
        DWORD mStartTime;
    #else
        struct timeval mStartTime;
    #endif
};
```
Now the API is cleaner, but the C++ code is more complex.

The ctor must allocate an Impl object and destroy it in the dtor.

Impl is declared as a private nested class, so it doesn’t add to global namespace and visibility.

This means it must be accessed by member functions of Impl, however could be moved to public to allow easier access.
AutoTimer::AutoTimer(const std::string &name) :
    m_impl(new AutoTimer::Impl())
{
    m_impl->mName = name;
#ifdef _WIN32
    m_impl->mStartTime = GetTickCount();
#else
    gettimeofday(&m_impl->mStartTime, NULL);
#endif
}

AutoTimer::~AutoTimer()
{
    std::cout << m_impl->mName << " took " << m_impl->GetElapsedTime()
              << " secs " << std::endl;
    delete m_impl;
    m_impl = NULL;
}
Copy Semantics

• By default the compiler will create a copy ctor and assignment operator

• As we have a pointer in the class we must ensure that a deep copy is done.

• If the class is non-copyable ensure that the copy and assignment operators are private

• Can also use either boost::noncopyable or Qt version
Pointers

• As the PIMPL idiom uses a pointer there can be trouble due to the allocation and de-allocation of the pointer.

• One way to overcome this problem is to use a smart pointer.

• There are a number of smart pointers available in the new C++ 11 specification however as we are not using this we need to use the boost ones.
Smart Pointers

• Smart pointers manage the life of a pointer and attempt to ensure proper de-allocation of resources

• C++ has std::auto_ptr good basic smart pointer but deprecated under C++ 11

• boost has a number of smart pointers which replicate the new C++ 11 smart pointers

• Fall into two main areas, single pointer or shared pointer
In the previous example we can use a scoped_ptr to store our m_impl pointer.

When the class is destroyed (goes out of scope) the pointer will be automatically destroyed.

class AutoTimer
{
public:
    /// Create a new timer object with a human-readable name
    explicit AutoTimer(const std::string &name);
    /// On destruction, the timer reports how long it was alive
    ~AutoTimer();

private:
    class Impl;
    boost::scoped_ptr<Impl> m_impl;
};
RAII

- The scoped pointer is a form of “resource acquisition is initialisation”
- In C++ the only code that can be guaranteed to be executed after an exception is thrown are the dtors of objects residing on the stack.
- Resources are acquired during initialisation as there is no chance of them being used before they are available
- They are released during the destruction of the object
API Wrapping

• A number of design patterns are used to aid wrapping of existing code or legacy API’s

• This is also good for wrapping C api’s to C++ or visa versa

• There are several patterns that allow this using different approaches

• In most cases there will be a performance issue as we are passing data from one class to another
The Observer Pattern

• In this pattern an object called the subject maintains a list of dependent classes called observers

• This is used to allow communication between classes without adding compile time dependancies

• It is used in various guises in most large software systems (for example Qt uses a version of this in signals and slots)
The Observer Pattern

Observer
+ notify()

ConcreteObserverA
+ notify()

ConcreteObserverB
+ notify()

Subject
+ observerCollection
+ registerObserver(observer)
+ unregisterObserver(observer)
+ notifyObservers()

Subject.notifyObservers() for observer in observerCollection
    call observer.notify()
class Observer;

class Subject
{
    private:
        std::vector<class Observer*> m_views;
        int m_value;
    public:
        void attach(Observer * _obs)
        {
            m_views.push_back(_obs);
        }
        void setVal(int _val)
        {
            m_value = _val;
            notify();
        }
        int getVal() const { return m_value; }
        void notify();
};
• In this case the registered `notify()` function in the dependents is the update method
class Observer
{
    Subject *m_subject;
    int m_denom;
public:
    Observer(Subject *_mod, int _div)
    {
        m_subject = _mod;
        m_denom = _div;
        m_subject->attach(this);
    }
    virtual void update() = 0;
protected:
    Subject *getSubject() { return m_subject; }
    int getDivisor() const { return m_denom; }
};
Concrete Classes

```cpp
class DivObserver: public Observer {
    public:
        DivObserver(Subject *_mod, int _div): Observer(_mod, _div) {}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        std::cout << v << " \_div\_" << d << " \_is\_" << v / d << '\n';
    }
};

class ModObserver: public Observer {
    public:
        ModObserver(Subject *_mod, int _div): Observer(_mod, _div) {}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        std::cout << v << " \_mod\_" << d << " \_is\_" << v % d << '\n';
    }
};
```
int main()
{
    Subject subj;
    DivObserver divObs1(&subj, 4);
    DivObserver divObs2(&subj, 3);
    ModObserver modObs3(&subj, 3);
    subj.setVal(14);
    std::cout<<"now set the value again\n";
    subj.setVal(22);
}

Every time the subject changes the values will be updated in the observing classes.

14 div 4 is 3
14 div 3 is 4
14 mod 3 is 2
now set the value again
22 div 4 is 5
22 div 3 is 7
22 mod 3 is 1
MVC Pattern

- The Model View Controller pattern is an extension of the Observer pattern.
- The model consists of application data and rules.
- The controller mediates input converting it to commands for the model or view.
MVC

• Many Graphical API’s use the MVC model to enable the separation of UI elements from the data

• As we split the Model and the View it makes it possible to implement several UI’s based on the same data model

• Reduces duplication

• A more modular approach to API development. Different developers can work on different elements
MVP

• Model View Presenter usually used in UI API’s

• The model is an interface defining the data to be displayed or modified by the user

• The view is a passive interface that displays the model and routes user events to the presenter to act upon that data.

• The presenter acts upon the model and the view. It retrieves data from the model, and formats it for display in the view.
The Memento Pattern

• This pattern allows an object to restore itself to a previous state

• Can be used for do / undo

• It is implemented using two objects the “originator” and the “caretaker”

• The originator is some object that has an internal state.

• The caretaker is going to do something to the originator, but wants to be able to undo the change
The Memento Pattern

**Originator**
- state
+ setMemento()
+ createMemento()

**Memento**
- state
+ getState()
+ setState()

**Caretaker**

return new Memento(state)

```
class Memento {
{
public:
    Memento(int _val)
    {
        m_state = _val;
    }

private:
    friend class Number;
    int m_state;
};
```
class Number
{
    public:
        Number(int _value) {
            m_value = _value;
        }
        void makeDouble() {
            m_value = 2 * m_value;
        }
        void half() {
            m_value = m_value / 2;
        }
        int getValue() {
            return m_value;
        }
        Memento *createMemento() {
            return new Memento(m_value);
        }
        void reinstateMemento(Memento *mem) {
            m_value = mem->m_state;
        }
    private:
        int m_value;
};
class Command
{
public:
    typedef void (Number::*Action)();
    Command(Number* receiver, Action action)
    {
        m_receiver = receiver;
        m_action = action;
    }
    virtual void execute()
    {
        m_mementoList[m_numCommands] = m_receiver->createMemento();
        m_commandList[m_numCommands] = this;
        if (m_numCommands > m_limitGuard)
        {m_limitGuard = m_numCommands;
        m_numCommands++;
        (m_receiver->*m_action)();
    }
    static void undo()
    {
        if (m_numCommands == 0)
        {std::cout << "Attempt to run off the end!!" << std::endl;
        return ;
    }
        m_commandList[m_numCommands - 1]->m_receiver->reinstateMemento
        (m_mementoList[m_numCommands - 1]);
        m_numCommands--;
    }
    void static redo()
    {
        if (m_numCommands > m_limitGuard)
        {std::cout << "Attempt to run off the end!!" << std::endl;
        return ;
    }
        (m_commandList[m_numCommands]->m_receiver->*m_commandList[m_numCommands] ->m_action)();
        m_numCommands++;
    }
protected:
    Number* m_receiver;
    Action m_action;
    static Command* m_commandList[20];
    static Memento* m_mementoList[20];
    static int m_numCommands;
    static int m_limitGuard;
};
int main()
{
    int i;
    std::cout << "Integer: ";
    std::cin >> i;
    Number *object = new Number(i);

    Command *commands[3];
    commands[1] = new Command(object, &Number::makeDouble);
    commands[2] = new Command(object, &Number::half);

    std::cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
    std::cin >> i;

    while (i)
    {
        if (i == 3)
            Command::undo();
        else if (i == 4)
            Command::redo();
        else
            commands[i]->execute();
        std::cout << "\n" << object->getValue() << std::endl;
        std::cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
        std::cin >> i;
    }
}
The Visitor Pattern

- Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Visitor implements “double dispatch”
- In “double dispatch”, the operation executed depends on: the name of the request, and the type of two receivers (the type of the Visitor and the type of the element it visits).
The Visitor Pattern

Client

<<interface>>
Element

ElementOne
+accept(v :Visitor)

ElementTwo
+accept(v :Visitor)

<<interface>>
Visitor

+visit(e : ElementOne)
+visit(e : ElementTwo)

<interface>VisitorOne

VisitorTwo

+visit(e : ElementOne)
+visit(e : ElementTwo)

The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types
```cpp
class Visitor;

class Element
{
  public:
    virtual void accept(class Visitor &v) = 0;
};

class A: public Element
{
  public:
    void accept(Visitor &v);
    std::string a()
    {
      return "class_A";
    }
};

class B: public Element
{
  public:
    void accept(Visitor &v);
    std::string b()
    {
      return "class_B";
    }
};

class C: public Element
{
  public:
    void accept(Visitor &v);
    std::string c()
    {
      return "class_C";
    }
};

void A::accept(Visitor &v)
{
  v.visit(this);
}
```
```cpp
class Visitor {
    public:
        virtual void visit(A *e) = 0;
        virtual void visit(B *e) = 0;
        virtual void visit(C *e) = 0;
};

class SomeOperation: public Visitor {
    void visit(A *e)
    {
        std::cout << "someOperation_on_A" + e->a() << '\n';
    }
    void visit(B *e)
    {
        std::cout << "someOperation_on_B" + e->b() << '\n';
    }
    void visit(C *e)
    {
        std::cout << "someOperation_on_C" + e->c() << '\n';
    }
};
```
#include "Visitors.h"

int main()
{
    Element *list[] = {
        new A(), new B(), new C()
    };
    SomeOperation op;
    SomeOtherOp other;
    for (int i = 0; i < 3; i++)
        list[i]->accept(op);
    for (int i = 0; i < 3; i++)
        list[i]->accept(other);
}
Command Pattern

• This basically allows us to wrap a method call into a new object

• Useful for abstracting / translating method calls into smaller and simpler code segments

• Can hide implementation detail as well as allowing user configuration

```cpp
void ProcessInput (enum Key)
{
    if(key == key::jump) jump();
    else if(key == key::walk) walk();
    ....
    else if(key == key::run) run();
}
```
• First we will define a base class that represents a triggerable command.

```cpp
class Agent;

class Command
{
public:
    Command();
    virtual ~Command() {}  
    virtual void execute(Agent& _agent) = 0;
};
```
Command Pattern

- next we create sub classes for each of the triggerable actions.

```cpp
class RunCommand : public Command {
    public:
        virtual void execute(Agent& _agent);
};
class WalkCommand : public Command {
    public:
        virtual void execute(Agent& _agent);
};
class LeftCommand : public Command {
    public:
        virtual void execute(Agent& _agent);
};
class RightCommand : public Command {
    public:
        virtual void execute(Agent& _agent);
};
class UpCommand : public Command {
    public:
        virtual void execute(Agent& _agent);
};
class DownCommand : public Command {
    public:
        virtual void execute(Agent& _agent);
};

void RunCommand::execute(Agent& _agent) {
    _agent.run();
}
void WalkCommand::execute(Agent& _agent) {
    _agent.walk();
}
void LeftCommand::execute(Agent& _agent) {
    _agent.left();
}
void RightCommand::execute(Agent& _agent) {
    _agent.right();
}
void UpCommand::execute(Agent& _agent) {
    _agent.up();
}
void DownCommand::execute(Agent& _agent) {
    _agent.down();
}
```
Command Pattern

- The input processor is going to define a pointer for each command we are going to use.

- In this case I’m going to use a smart pointer to store the different commands so we don’t need to worry about the dtor.

- Now when this is being used we have added a level of indirection to the method call
#include <memory>

// a simple enum for our agent modes
enum class Moves{RUN, WALK, UP, DOWN, LEFT, RIGHT};

class InputProcessor
{
public:
    InputProcessor()
    :
        m_walk(new WalkCommand),
        m_run(new RunCommand),
        m_left(new LeftCommand),
        m_right(new RightCommand),
        m_up(new UpCommand),
        m_down(new DownCommand)
    {}

    Command* handleInput(Moves _m);

private:
    // as these are pointers use a smart pointer to store
    std::unique_ptr<Command> m_walk;
    std::unique_ptr<Command> m_run;
    std::unique_ptr<Command> m_left;
    std::unique_ptr<Command> m_right;
    std::unique_ptr<Command> m_up;
    std::unique_ptr<Command> m_down;
};
Command* InputProcessor::handleInput(Moves _m)  
{
    Command *ret;
    // return the concrete type for the move
    switch(_m)
    {
    case Moves::WALK : ret=m_walk.get(); break;
    case Moves::RUN : ret=m_run.get(); break;
    case Moves::LEFT : ret=m_left.get(); break;
    case Moves::UP : ret=m_up.get(); break;
    case Moves::DOWN : ret=m_down.get(); break;
    case Moves::RIGHT : ret=m_right.get(); break;
    default: ret=nullptr; break;
    }

    return ret;
}
```cpp
#include "Command.h"
#include "InputProcessor.h"
#include "Agent.h"

int main()
{
  // create an array of moves to demonstrate
  // this could actually be an interactive key loop,
  // moves generated from an AI or playback from a file

  std::array<Moves, 10> input = {
      {Moves::WALK, Moves::RIGHT,
       Moves::RUN, Moves::UP,
       Moves::RIGHT, Moves::DOWN,
       Moves::WALK, Moves::DOWN,
       Moves::LEFT, Moves::LEFT
    };

  // create an agent
  std::vector<Agent> agents;
  agents.push_back(Agent(Point2(0.0f, 0.0f), "agent_1"));
  agents.push_back(Agent(Point2(0.0f, 2.0f), "agent_2"));
  agents.push_back(Agent(Point2(3.0f, 0.0f), "agent_3"));

  // create an input processor
  InputProcessor processor;
  // process data
  for(auto move : input)
  {
    Command* command = processor.handleInput(move);
    if (command != nullptr)
    {
      for(auto a : agents)
      {
        command->execute(a);
        a.debug();
      }
    }
  }
  return EXIT_SUCCESS;
}
```

agent 1 position 0 0
agent 2 position 0 2
agent 3 position 3 0
agent 1 position 1 0
agent 2 position 1 2
agent 3 position 4 0
agent 1 position 0 0
agent 2 position 0 2
agent 3 position 3 0
agent 1 position 0 1
agent 2 position 0 3
agent 3 position 3 1
agent 1 position 1 0
agent 2 position 1 2
agent 3 position 4 0
agent 1 position 0 -1
agent 2 position 0 1
agent 3 position 3 -1
agent 1 position 0 0
agent 2 position 0 2
agent 3 position 3 0
agent 1 position 0 -1
agent 2 position 0 1
agent 3 position 3 -1
agent 1 position -1 0
agent 2 position -1 2
agent 3 position 2 0
agent 1 position -1 0
agent 2 position -1 0
agent 3 position 2 2
agent 1 position -1 2
agent 2 position -1 2
agent 3 position 2 0
Command Pattern

- In the previous example I used an array of data to playback moves.
- This could be user input, file serialisation or even network data used to drive the movement of the agent.
- It is also possible to add state to the concrete commands and implement undo redo in a very simple way.
- Sometimes this may be better than using the memento pattern as it requires less memory.
flyweight pattern

• As the name implies this pattern is a light weight pattern

• It is designed to reduce the duplication of similar data and methods from similar classes

• The following example demonstrates this
Example

- In this case each instance of the tree would have a copy of both Mesh and two textures

- This (especially in Games) will be the same mesh and texture for each instance of the tree

- If we are creating a Forest of 1000’s of trees there is a large duplication of data

- We can use the flyweight to gain a smaller memory footprint.
• Now we have moved the “intrinsic state” or “context free” data to the Model class

• This will be created once (or perhaps via inheritance to individual Tree types via a factory method)

• The actual Tree now has only it’s specific state (extrinsic)
flyweight example 2

• In the following example we use a simple flyweight terrain class to hold different instances of terrain with unique state data

• We can then hold an array of pointers to this in our world (3 types in this case)

• Whilst the lookup may incur some cost it still has a smaller memory footprint

• Some of this could be done by storing an enum however this is fairly easy to extend.
enum class TextureID {GRASS, HILL, RIVER};

class Terrain {
 public:
  Terrain(int _moveCost, bool _isWater, TextureID _tex) :
    m_moveCost(_moveCost),
    m_isWater(_isWater),
    m_tex(_tex) {}

  int getMoveCost() const {return m_moveCost;}  
  bool isWater() const {return m_isWater;}       
  const TextureID & getTexture() const {
    return m_tex;
  }

 private:
  int m_moveCost;
  bool m_isWater;
  TextureID m_tex;
};
// based on discussion here
// http://gameprogrammingpatterns.com/flyweight.html
#include <cassert>
#include <random>
#include "Terrain.h"
constexpr int WIDTH=80;
constexpr int HEIGHT=20;

class World
{
public:
    World():
        m_grassTerrain(1,false,TextureID::GRASS),
        m_hillTerrain(3,false,TextureID::HILL),
        m_riverTerrain(2,true,TextureID::RIVER){}

    void generateTerrain();
    const Terrain & getTile(int _x, int _y) const
    {
        assert(_x<WIDTH);
        assert(_y<HEIGHT);
        return *m_tiles[_x][_y];
    }

private:
    Terrain m_grassTerrain;
    Terrain m_hillTerrain;
    Terrain m_riverTerrain;
    // simple 2d array for now but could use
    // other container
    Terrain *m_tiles[WIDTH][HEIGHT];
};
void World::generateTerrain()
{
    // Seed with a real random value, if available
    std::random_device rd;
    // create a mersenne twister generator
    std::mt19937 gen(rd());
    // create an int distribution
    std::uniform_int_distribution<> dis(0, 10);

    for(int x=0; x<WIDTH; ++x)
    {
        for(int y=0; y<HEIGHT; ++y)
        {
            if(dis(gen) == 0)
            {
                m_tiles[x][y]=&m_hillTerrain;
            }
            else
            {
                m_tiles[x][y]=&m_grassTerrain;
            }
        }
    }
    std::uniform_int_distribution<> dis2(0, WIDTH);

    int x=dis2(gen);
    for(int y=0; y<HEIGHT; ++y)
    {
        m_tiles[x][y]=&m_riverTerrain;
    }
}
#include "World.h"

int main()
{
World world;
world.generateTerrain();
for(int y=0; y<HEIGHT; ++y)
{
    for(int x=0; x<WIDTH; ++x)
    {
        std::cout<<world.getTile(x,y).getMoveCost();
    }
    std::cout<<"\n";
}
The Composite Pattern

- Compose objects into tree structures to represent whole-part hierarchies
- 1-to-many "has a" up the "is a" hierarchy
// The Component

class DrawingElement
{
public:
    DrawingElement(std::string _name) : m_name(_name) {}
    virtual void Add(DrawingElement* _d) = 0;
    virtual void Remove(DrawingElement* _d) = 0;
    virtual void Display(int _indent) = 0;
    virtual ~DrawingElement() {};

protected:
    std::string m_name;

private:
    // no default ctor
    DrawingElement();
};
class PrimitiveElement : public DrawingElement
{
public:

    PrimitiveElement(std::string _name) : DrawingElement(_name)
    {
    
    }

    void Add(DrawingElement* _d)
    {
        std::cout << "Cannot add to a PrimitiveElement" << std::endl;
    }

    void Remove(DrawingElement* _d)
    {
        std::cout << "Cannot remove from a PrimitiveElement" << std::endl;
    }

    void Display(int _indent)
    {
        std::string newStr(_indent, '-');
        std::cout << newStr << "_" << m_name << std::endl;
    }

    virtual ~PrimitiveElement();
private:

    // no default ctor
    PrimitiveElement();
};
// The 'Composite' class

class CompositeElement : public DrawingElement
{
public:
    CompositeElement(std::string _name) : DrawingElement(_name) {};
    void Add(DrawingElement* _d)
    {
        m_elements.push_back(_d);
    }
    void Remove(DrawingElement* _d)
    {
        std::vector<DrawingElement*>::iterator it = m_elements.begin();
        while(it != m_elements.end())
        {
            if(*it == _d)
            {
                delete _d;
                m_elements.erase(it);
                break;
            }
            ++it;
        }
    }
    void Display(int _indent)
    {
        std::string newStr(_indent, '-');
        std::cout << newStr << '+' << m_name << std::endl;
        std::vector<DrawingElement*>::iterator it = m_elements.begin();
        while(it != m_elements.end())
        {
            (*it)->Display(_indent + 2);
            ++it;
        }
    }
    virtual ~CompositeElement()
    {
        while(!m_elements.empty())
        {
            std::vector<DrawingElement*>::iterator it = m_elements.begin();
            delete *it;
            m_elements.erase(it);
        }
    }
private:
    // no default allocation
    CompositeElement();
    std::vector<DrawingElement*> m_elements;
};
```cpp
#include "Composite.h"

int main()
{
    // Create a Tree Structure
    CompositeElement* root = new CompositeElement("Picture");
    root->Add(new PrimitiveElement("Red Line"));
    root->Add(new PrimitiveElement("Blue Circle"));
    root->Add(new PrimitiveElement("Green Box"));
    
    // Create a Branch
    CompositeElement* comp = new CompositeElement("Two Circles");
    comp->Add(new PrimitiveElement("Black Circle"));
    comp->Add(new PrimitiveElement("White Circle"));
    root->Add(comp);
    
    // Add and remove a primitive elements
    PrimitiveElement* pe1 = new PrimitiveElement("Yellow Line");
    root->Add(pe1);
    PrimitiveElement* pe2 = new PrimitiveElement("Orange Triangle");
    root->Add(pe2);
    root->Remove(pe1);
    
    // Recursively display nodes
    root->Display(1);
    
    // delete the allocated memory
    delete root;
    return 0;
}
The State Pattern

- The state pattern (and the Strategy pattern) are behavioural patterns.
- They allows an object to cleanly change it’s state at runtime
- They can also be useful for implementing Finite state machines
Game State Machine

Game
- m_state : GameState
+ draw()
+ update()
+ setState()

GameState <<abstract>>
- m_gameState : GameState
  # m_highScore : int
  # m_score : int
  + init()
  + draw()
  + nextState()
  + setState(_s : GameState *)

MainMenu
GamePlay
GameOver
HighScore
// abstract game state

class GameState
{

public:
    virtual void draw() = 0;
    virtual void init() = 0;
    virtual void nextState() = 0;
    void setState(GameState *s) { m_state = s; }
    inline void setScore(int i) { m_score = i; }
    inline int getScore() const { return m_score; }
    inline void setHighScore(int i) { m_highScore = i; }
    inline int getHighScore() const { return m_highScore; }

private:
    GameState *m_state;

protected:
    static int m_score;
    static int m_highScore;
};

int GameState::m_highScore = 0;
int GameState::m_score = 0;
class Game
{
    public:
    Game(int score);
    void draw()
    {
        m_state->draw();
    }
    void update()
    {
        m_state->nextState();
    }
    inline void setScore(int _i)
    {
        m_state->setScore(_i);
    }
    ~Game()
    {
        std::cout<<"game_dtor\n";
    }
    private:
    GameState *m_state;
};

Game::Game(int _score)
{
    m_state = new MainMenu;
    m_state->setScore(0);
    m_state->setHighScore(_score);
}
class MainMenu : public GameState
{
    public:
        MainMenu(){ this->init();}
        virtual ~MainMenu(){std::cout<<"Main_menu_dtor\n";}
        void init() { std::cout<<"main_Menu_init\n"; }
        void draw() { std::cout<<"main_menu_draw\n"; }
        void nextState();
};

void MainMenu::nextState()
{
    setState(new Gameplay);
    delete this;
}
class GamePlay : public GameState
{
    public:
    GamePlay() { this->init(); }
    virtual ~GamePlay() { std::cout << "GamePlay_dtor\n"; }
    void init() { std::cout << "GamePlay_init\n"; }
    void draw() { std::cout << "GamePlay_draw\n"; }
    void nextState();
};

void GamePlay::nextState()
{
    setState(newGameOver);
    delete this;
}
class GameOver : public GameState
{
    public:
    GameOver() { this->init(); }
    virtual ~GameOver() { std::cout << "GameOver\ndtor\n"; }

    void init() { std::cout << "GameOver_init\n"; }
    void draw() { std::cout << "GameOver_draw\n"; }
    void nextState();
};

class HighScore : public GameState
{
    public:
    HighScore() { this->init(); }
    virtual ~HighScore() { std::cout << "HighScore\ndtor\n"; }

    void init() { std::cout << "//////HighScore_init///////\n"; }
    void draw() { std::cout << "//////HighScore_draw///////\n"; }
    void nextState();
};
void GameOver::nextState()
{
    std::cout << "score is " << m_score << "\n";
    if (m_score > m_highScore)
    {
        m_highScore = m_score;
        setState(new HighScore);
    }
    else
    {
        setState(new MainMenu);
    }
    delete this;
}
#include <iostream>
#include "State.h"

int main()
{
    // create game with high score of 0
    Game g(0);
    std::cout<<"**START**\n";
    for(int i=0; i<4; ++i)
    {
        g.draw();
        std::cout<<"**now update the state**\n";
        g.update();
    }
    std::cout<<"setting high score\n";
    g.setScore(200);
    g.update();
    g.update();
    g.update();
    std::cout<<"***EXIT_GAME_AND_TIDY_UP***\n";
}

main Menu init
** START **
main menu draw
** now update the state**
GamePlay init
Main menu dtor
main menu draw
** now update the state**
GamePlay init
GamePlay dtor
GamePlay draw
** now update the state**
GameOver init
GameOver dtor
GameOver draw
** now update the state**
score is 0
main Menu init
Main menu dtor
setting high score
GamePlay init
GamePlay dtor
GameOver init
GameOver dtor
score is 200
/////// HighScore init /////
HighScore dtor
***EXIT GAME AND TIDY UP***
game dtor
References

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.


References

References

• http://en.wikipedia.org/wiki/Prototype_pattern

• http://en.wikipedia.org/wiki/Liskov_substitution_principle

• “The Pragmatic Programmer” Andrew Hunt, David Thomas

• http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.3681&rep=rep1&type=pdf
References

- [http://sourcemaking.com/](http://sourcemaking.com/)
- [http://sourcemaking.com/design_patterns/observer/cpp/3](http://sourcemaking.com/design_patterns/observer/cpp/3)
References

- [http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller](http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)
- [http://sourcemaking.com/design_patterns/memento](http://sourcemaking.com/design_patterns/memento)
- [http://advancedcppwithexamples.blogspot.co.uk/2010/09/c-example-for-composite-design-pattern.html](http://advancedcppwithexamples.blogspot.co.uk/2010/09/c-example-for-composite-design-pattern.html)