6

Design Pattern for Games Development

In chapter, the following recipes will be covered:

* Using the singleton design pattern
* Using the factory method
* Using the abstract factory method
* Using the observer pattern
* Using the fly-weight pattern
* Using the strategy pattern
* Using the command pattern

# Introduction

Let us consider that we are faced with a certain problem. After sometime, we find a solution to that problem. Now if the problem reoccurs or a similar pattern of the problem reoccurs, we will know how to solve the problem by applying the same principle which solved the previous problem. Design patterns are similar to that. There are already 23 such solutions documented which provides subtle solutions to how to deal with problems which have a similar pattern to the ones that are documented. They are described by the authors more commonly referred as the Gangs of Four. They are not complete solutions but rather a template or a framework which can be applied to similar situations. One of the biggest drawback of the design pattern however is that if they are not applied correctly, they can prove to be disastrous. The design patterns can be classified into Structural, behavioral and creational. We will be looking at only few of them which are used often in games.

# Using the singleton design pattern

1. The singleton design pattern is the most used design pattern in games. And unfortunately, it is also the most overused and most incorrectly applied design pattern in games. There are a few advantages of the singleton design pattern which we will discuss. However, it has a lot of deadly consequences as well.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will see how easy it is to create a singleton design pattern. We will also see what are the common pitfalls of this design pattern.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

using namespace std;

class PhysicsManager

{

private:

static bool bCheckFlag;

static PhysicsManager \*s\_singleInstance;

PhysicsManager()

{

//private constructor

}

public:

static PhysicsManager\* getInstance();

void GetCurrentGravity()const;

~PhysicsManager()

{

bCheckFlag = false;

}

};

bool PhysicsManager::bCheckFlag = false;

PhysicsManager\* PhysicsManager::s\_singleInstance = NULL;

PhysicsManager\* PhysicsManager::getInstance()

{

if (!bCheckFlag)

{

s\_singleInstance = new PhysicsManager();

bCheckFlag = true;

return s\_singleInstance;

}

else

{

return s\_singleInstance;

}

}

void PhysicsManager::GetCurrentGravity() const

{

//Some calculations for finding the current gravity

//Probably a base variable which constantly gets updated with value

//based on the environemnt

cout << "Current gravity of the system is: " <<9.8<< endl;

}

int main()

{

PhysicsManager \*sc1, \*sc2;

sc1 = PhysicsManager::getInstance();

sc1->GetCurrentGravity();

sc2 = PhysicsManager::getInstance();

sc2->GetCurrentGravity();

\_getch();

return 0;

}

## How it works...

The main reason why anyone wants to use a singleton class, is when he wants to restrict to just one instance of the class. In our example, we have taken the Physics Manager class. We have the constructor as private and then assigned a static function to get the handle to the instance of the class and hence its methods. We also use a boolean to check if an instance is already created. If it is, we do not assign a new instance. If it is not, we assign a new instance and call the corresponding methods.

As intelligent as it may seem, this design pattern has many flaws and hence should be avoided as much as possible in game design. Firstly, it’s a global variable. This in itself is bad. Secondly this encourages bad coupling which may appear in the code. Third, it is not concurrent friendly. Imagine there are multiple threads, each thread can access this global variable. Hence it is a recipe for disaster for deadlock to happen. Finally, one of the most common mistakes made by new programmers is to create managers for everything. And then make the manager as singleton. Fact is, we can get away completely from creating manager by using OOPS and references in an effective manner.

The above code shows a lazy value of initialising singleton and hence can be improved. However, all the fundamental problems as described above will still remain.

# Using the factory method

A factory essentially is a warehouse for creating objects of other type. In a factory method design pattern, the creation of a new type of object, like an enemy or building, happens from an interface and the subclass decides which class it needs to instantiate. This is also a commonly used pattern in games and can be quite useful.

## Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will find out how easy it is to write a factory method design pattern

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

#include <vector>

using namespace std;

class IBuilding

{

public:

virtual void TotalHealth() = 0;

};

class Barracks : public IBuilding

{

public:

void TotalHealth()

{

cout << "Health of Barrack is :" << 100;

}

};

class Temple : public IBuilding

{

public:

void TotalHealth()

{

cout << "Health of Temple is :" << 75;

}

};

class Farmhouse : public IBuilding

{

public:

void TotalHealth()

{

cout << "Health of Farmhouse is :" << 50;

}

};

int main()

{

vector<IBuilding\*> BuildingTypes;

int choice;

cout << "Specify the different building types in your village" << endl;

while (true)

{

cout << "Barracks(1) Temple(2) Farmhouse(3) Go(0): ";

cin >> choice;

if (choice == 0)

break;

else if (choice == 1)

BuildingTypes.push\_back(new Barracks);

else if (choice == 2)

BuildingTypes.push\_back(new Temple);

else

BuildingTypes.push\_back(new Farmhouse);

}

cout << endl;

cout << "There are total " << BuildingTypes.size() << " buildings" << endl;

for (int i = 0; i < BuildingTypes.size(); i++)

{

BuildingTypes[i]->TotalHealth();

cout << endl;

}

for (int i = 0; i < BuildingTypes.size(); i++)

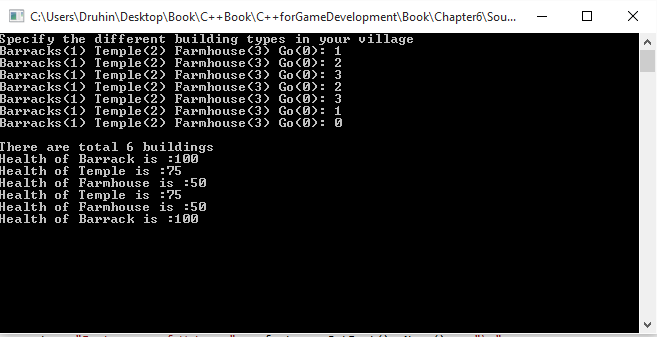
delete BuildingTypes[i];

\_getch();

}

## How it works...

In this example, we have created a Building interface which has a pure virtual function for totalhealth. This means all the derived classes must over ride this function. Hence we can guarantee that all our buildings will have this property. We can keep adding to this structure by having more properties like hit points, total storage capacity, rate of production of villagers and so on based on the nature and design of the game. The derived classes have their own implementation of totalhealth. They are also named to reflect the type of building they are. The biggest advantage of this design pattern is that all we need at the client side is a reference to the base interface. After that we can create the type of building we need at runtime. We store those building types in a vector list and finally use a loop to display the contents. Since we have the reference IBuilding\*, we can assign any new derived class we want at runtime. There is no need to create reference for all the derived class such as Temple\* and so. The image below shows the output we are likely to get for a user defined village.



Insert Image B04929\_06\_01.png

# Using the abstract factory method

An abstract factory is a part of the creational design pattern. It is one of the best ways to create an object and is a commonly repeated design pattern in games. It is like a factory of factories. It uses an interface to create a factory. The factory is responsible for creating objects without specifying their class type. The factory generates these objects based on the factory method design pattern. However, some can argue that the abstract factory method can also be implemented using the prototype design pattern.

## Getting ready

1. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will find out how easy it is to implement the abstract factory pattern.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

#include <string>

using namespace std;

//IFast interface

class IFast

{

public:

virtual std::string Name() = 0;

};

//ISlow interface

class ISlow

{

public:

virtual std::string Name() = 0;

};

class Rapter : public ISlow

{

public:

std::string Name()

{

return "Rapter";

}

};

class Cocumbi : public IFast

{

public:

std::string Name()

{

return "Cocumbi";

}

};

. . . . .// Similar classes can be written here

class AEnemyFactory

{

public:

enum Enemy\_Factories

{

Land,

Air,

Water

};

virtual IFast\* GetFast() = 0;

virtual ISlow\* GetSlow() = 0;

static AEnemyFactory\* CreateFactory(Enemy\_Factories factory);

};

class LandFactory : public AEnemyFactory

{

public:

IFast\* GetFast()

{

return new Cocumbi();

}

ISlow\* GetSlow()

{

return new Marzel();

}

};

class AirFactory : public AEnemyFactory

{

public:

IFast\* GetFast()

{

return new Zybgry();

}

ISlow\* GetSlow()

{

return new Bungindi();

}

};

class WaterFactory : public AEnemyFactory

{

public:

IFast\* GetFast()

{

return new Manama();

}

ISlow\* GetSlow()

{

return new Pokili();

}

};

//CPP File

AEnemyFactory\* AEnemyFactory::CreateFactory(Enemy\_Factories factory)

{

if (factory == Enemy\_Factories::Land)

{

return new LandFactory();

}

else if (factory == Enemy\_Factories::Air)

{

return new AirFactory();

}

else if (factory == Enemy\_Factories::Water)

{

return new WaterFactory();

}

}

int main(int argc, char\* argv[])

{

AEnemyFactory \*factory = AEnemyFactory::CreateFactory

(AEnemyFactory::Enemy\_Factories::Land);

cout << "Slow enemy of Land: " << factory->GetSlow()->Name() << "\n";

delete factory->GetSlow();

cout << "Fast enemy of Land: " << factory->GetFast()->Name() << "\n";

delete factory->GetFast();

delete factory;

getchar();

factory = AEnemyFactory::CreateFactory(AEnemyFactory::Enemy\_Factories::Air);

cout << "Slow enemy of Air: " << factory->GetSlow()->Name() << "\n";

delete factory->GetSlow();

cout << "Fast enemy of Air: " << factory->GetFast()->Name() << "\n";

delete factory->GetFast();

delete factory;

getchar();

factory = AEnemyFactory::CreateFactory(AEnemyFactory::Enemy\_Factories::Water);

cout << "Slow enemy of Water: " << factory->GetSlow()->Name() << "\n";

delete factory->GetSlow();

cout << "Fast enemy of Water: " << factory->GetFast()->Name() << "\n";

delete factory->GetFast();

getchar();

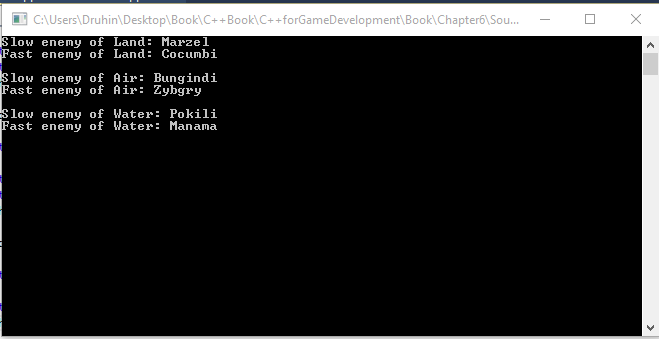
return 0;

}

## How it works...

In this example, we have created two interfaces namely IFast and ISlow. After that we have created several enemies which decide whether they are fast or slow. Finally, we create an abstract class with two virtual functions to get the fast enemy and the slow enemy. This means all the derived classes must override and have their own implementation of these functions. So in effect we have created a factory of factories. The land, air and water enemy factories which we have created from the abstract class has references of two interfaces for slow and fast. Hence the land, water and air serve as factories themselves too.

So from the client side, we can request a fast land enemy or a slow water enemy and we can get the appropriate enemy displayed to us. As the image shows, we can get the output as displayed below.



Insert Image B04929\_06\_02.png

# Using the observer pattern

Observer design pattern is one which is used quite less in games, but it should be used more often by game developers as it is a very smart way to handle notifications. In the observer design pattern, a component maintains a one to many relationships with other components. This means when the main component changes, all the dependent components also updates. Imagine a physics system. Now we want enemy1 and enemy2 to update as soon as the physics system updates. Then we may use this pattern.

## Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to implement the observer pattern.
2. Open Visual Studio.
3. Create a new C++ project
4. Select a win32 Windows application
5. Add a source file called Source.cpp
6. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <vector>

#include <conio.h>

using namespace std;

class PhysicsSystem {

vector < class Observer \* > views;

int value;

public:

void attach(Observer \*obs) {

views.push\_back(obs);

}

void setVal(int val) {

value = val;

notify();

}

int getVal() {

return value;

}

void notify();

};

class Observer {

PhysicsSystem \*\_attribute;

int iScalarMultiplier;

public:

Observer(PhysicsSystem \*attribute, int value)

{

\_attribute = attribute;

iScalarMultiplier = value;

\_attribute->attach(this);

}

virtual void update() = 0;

protected:

PhysicsSystem \*getPhysicsSystem() {

return \_attribute;

}

int getvalue()

{

return iScalarMultiplier;

}

};

void PhysicsSystem::notify() {

for (int i = 0; i < views.size(); i++)

views[i]->update();

}

class PlayerObserver : public Observer {

public:

PlayerObserver(PhysicsSystem \*attribute, int value) : Observer(attribute, value){}

void update() {

int v = getPhysicsSystem()->getVal(), d = getvalue();

cout << "Player is dependant on the Physics system" << endl;

cout << "Player new impulse value is " << v / d << endl << endl;

}

};

class AIObserver : public Observer {

public:

AIObserver(PhysicsSystem \*attribute, int value) : Observer(attribute, value){}

void update() {

int v = getPhysicsSystem()->getVal(), d = getvalue();

cout << "AI is dependant on the Physics system" << endl;

cout << "AI new impulse value is " << v % d << endl << endl;

}

};

int main() {

PhysicsSystem subj;

PlayerObserver valueObs1(&subj, 4);

AIObserver attributeObs3(&subj, 3);

subj.setVal(100);

\_getch();

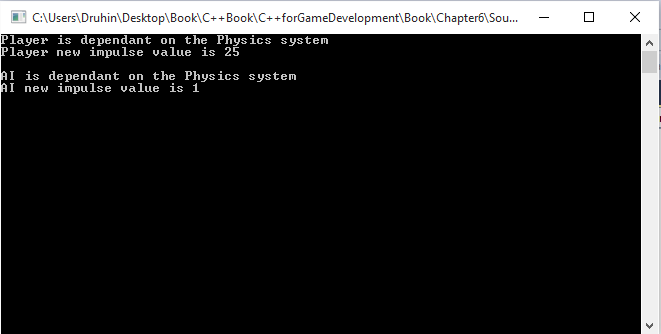
}

## How it works...

In this example we have created a physics system which continuously updates its value. Other components which are dependent on the physics system must attach themselves to the physics system, so that they are notified as soon as the physics system is updated.

The physics system that we have created holds a vector list of all the components that are observing from it. In addition to it, it contains methods to get the current value or set a value to it. It also contains a method to notify all the dependant components once a value has been changed in the Physics system. The Observer class contains reference to the Physics system as well as a pure virtual function for update which the derived class must override. The Player and the AI class can derive from this class and have their own implementation of impulse that they want to have based on the changes of the physics system. Both the Ai and player system will continuously receive updates from the physics system unless they detach themselves from it.

This is a very useful pattern and has loads of implementation in games. The diagram below shows how a typical output would look like.



Insert Image B04929\_06\_03.png

# Using the flyweight pattern

The fly-weight design pattern is mostly used when we want to reduce the amount of memory that is used to create the objects. This pattern is often used when we want to create something hundreds or thousands of times. In games mostly a forest structure, would often use this design pattern. This design pattern falls under the structural design pattern. In this pattern, the object, let’s say the tree object is divided into parts, one that is dependent on the state of the object and one that is independent. The independent part is stored in the fly-weight object whereas the dependent part is handled by the client and sent to the fly-weight object as and when invoked.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to implement the flyweight pattern.
2. Open Visual Studio.
3. Create a new C++ project
4. Select a win32 console application
5. Add a source file called Source.cpp
6. Add the following lines of code

Source.cpp

#include <iostream>

#include <string>

#include <map>

#include <conio.h>

using namespace std;

class TreeType

{

public:

virtual void Display(int size) = 0;

protected:

//Some Model we need to assign.For relevance we are substituting this with a character symbol

char symbol\_;

int width\_;

int height\_;

float color\_;

int Size\_;

};

class TreeTypeA : public TreeType

{

public:

TreeTypeA()

{

symbol\_ = 'A';

width\_ = 94;

height\_ = 135;

color\_ = 0;

Size\_ = 0;

}

void Display(int size)

{

Size\_ = size;

cout << "Size of " << symbol\_ << " is :" << Size\_ << endl;

}

};

class TreeTypeB : public TreeType

{

public:

TreeTypeB()

{

symbol\_ = 'B';

width\_ = 70;

height\_ = 25;

color\_ = 0;

Size\_ = 0;

}

void Display(int size)

{

Size\_ = size;

cout << "Size of " << symbol\_ << " is :" << Size\_ << endl;

}

};

class TreeTypeZ : public TreeType

{

public:

TreeTypeZ()

{

symbol\_ = 'Z';

width\_ = 20;

height\_ = 40;

color\_ = 1;

Size\_ = 0;

}

void Display(int size)

{

Size\_ = size;

cout <<"Size of " << symbol\_ << " is :" << Size\_ << endl;

}

};

// The 'FlyweightFactory' class

class TreeTypeFactory

{

public:

virtual ~TreeTypeFactory()

{

while (!TreeTypes\_.empty())

{

map<char, TreeType\*>::iterator it = TreeTypes\_.begin();

delete it->second;

TreeTypes\_.erase(it);

}

}

TreeType\* GetTreeType(char key)

{

TreeType\* TreeType = NULL;

if (TreeTypes\_.find(key) != TreeTypes\_.end())

{

TreeType = TreeTypes\_[key];

}

else

{

switch (key)

{

case 'A':

TreeType = new TreeTypeA();

break;

case 'B':

TreeType = new TreeTypeB();

break;

//...

case 'Z':

TreeType = new TreeTypeZ();

break;

default:

cout << "Not Implemented" << endl;

throw("Not Implemented");

}

TreeTypes\_[key] = TreeType;

}

return TreeType;

}

private:

map<char, TreeType\*> TreeTypes\_;

};

//The Main method

int main()

{

string forestType = "ZAZZBAZZBZZAZZ";

const char\* chars = forestType.c\_str();

TreeTypeFactory\* factory = new TreeTypeFactory;

// extrinsic state

int size = 10;

// For each TreeType use a flyweight object

for (size\_t i = 0; i < forestType.length(); i++)

{

size++;

TreeType\* TreeType = factory->GetTreeType(chars[i]);

TreeType->Display(size);

}

//Clean memory

delete factory;

\_getch();

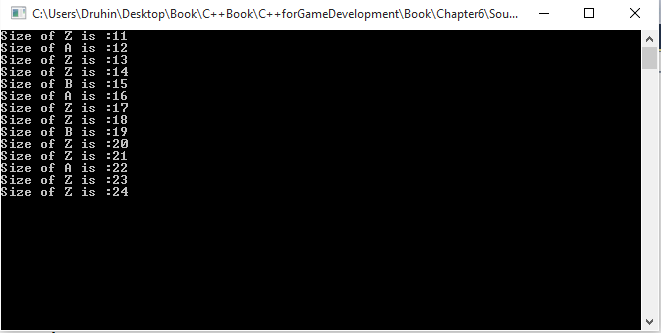
return 0;

}

## How it works...

In this example, we have created a forest. The basic principle of fly weight pattern is applied whereby part of the structure is shared across all trees and part is dictated by the client. In this example, except the size (this could be anything, size is just chosen to be an example) every other attribute is chosen to be shared. We create a tree type interface which contains all the attributes. We then have derived classes with have the attributes overridden and a method to set the “size” attribute. We can have multiple such trees. Generally, more the variety of trees, the more detailed the forest will look like. Let us say that we have 10 different types of trees, so we need to have 10 different classes which derives from the interface and has a method to assign the “size” attribute from the client size.

Finally, we have the Tree factory which assigns each tree at run time. We create a reference to the interface as we do with any factory pattern. However, instead of directly instantiating a new object, we first check in the map if that tree attributes are already present. If it is not, we assign a new object and push the attributes to a map. So the next time, a request comes for a similar tree structure which has already been assigned, we can share the attributes from the map. Finally, from the client, we create a forest type document which we feed to the factory and it generates the forest for us using the trees listed in the document. As the majority of the attributes are shares, the memory footprint is very low. The following image shows us how the forest is created.



Insert Image B04929\_06\_04.png

# Using the strategy pattern

The strategy design pattern is a very smart of designing the code. In games, this is mostly used for the AI component. In this pattern, we define a large number of algorithms and have all of them from a common interface signature. Then at runtime, we can change the clients of the algorithms. So in effect, the algorithms are independent of the clients.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. In this recipe we will find out how easy it is to implement the strategy pattern.
2. Open Visual Studio.
   1. Create a new C++ project
   2. Select a win32 console application
   3. Add the following files: Source.cpp
   4. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

using namespace std;

class SpecialPower

{

public:

virtual void power() = 0;

};

class Fire : public SpecialPower

{

public:

void power()

{

cout << "My power is fire" << endl;

}

};

class Invisibility : public SpecialPower

{

public:

void power()

{

cout << "My power is invisibility" << endl;

}

};

class FlyBehaviour

{

public:

virtual void fly() = 0;

};

class FlyWithWings : public FlyBehaviour

{

public:

void fly()

{

cout << "I can fly" << endl;

}

};

class FlyNoWay : public FlyBehaviour

{

public:

void fly()

{

cout << "I can't fly!" << endl;

}

};

class FlyWithRocket : public FlyBehaviour

{

public:

void fly()

{

cout << "I have a jetpack" << endl;

}

};

class Enemy

{

public:

SpecialPower \*specialPower;

FlyBehaviour \*flyBehaviour;

void performPower()

{

specialPower->power();

}

void setSpecialPower(SpecialPower \*qb)

{

cout << "Changing special power..." << endl;

specialPower = qb;

}

void performFly()

{

flyBehaviour->fly();

}

void setFlyBehaviour(FlyBehaviour \*fb)

{

cout << "Changing fly behaviour..." << endl;

flyBehaviour = fb;

}

void floatAround()

{

cout << "I can float." << endl;

}

virtual void display() = 0; // Make this an abstract class by having a pure virtual function

};

class Dragon : public Enemy

{

public:

Dragon()

{

specialPower = new Fire();

flyBehaviour = new FlyWithWings();

}

void display()

{

cout << "I'm a dragon" << endl;

}

};

class Soldier : public Enemy

{

public:

Soldier()

{

specialPower = new Invisibility();

flyBehaviour = new FlyNoWay();

}

void display()

{

cout << "I'm a soldier" << endl;

}

};

int main()

{

Enemy \*dragon = new Dragon();

dragon->display();

dragon->floatAround();

dragon->performFly();

dragon->performPower();

cout << endl << endl;

Enemy \*soldier = new Soldier();

soldier->display();

soldier->floatAround();

soldier->performFly();

soldier->setFlyBehaviour(new FlyWithRocket);

soldier->performFly();

soldier->performPower();

soldier->setSpecialPower(new Fire);

soldier->performPower();

\_getch();

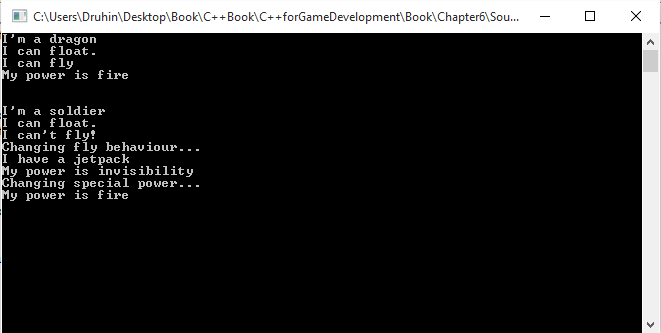
return 0;

}

## How it works...

In this example, we have created different interfaces for different properties which the enemy may have. So since we know, special power is a property every enemy type will have, we have created an interface called specialpower and then derived two classes from it called fire and invisibility. We can add as many special powers as we want, we just need to create a new class and derive it from the special power interface. Similarly, all the enemy types should have a property of flying. Either they fly or don’t fly or fly with help of jetpacks.

So we have created a Flybehavior interface and have the different fly type classes derive from it. After that we have created an abstract class for enemy type which contains both the interfaces as references. Hence any derived class can decide what fly type and what special power it needs. This also gives us the flexibility to change the special powers and the fly ability at run time. The diagram below shows a brief example of this design pattern.



Insert Image B04929\_06\_05.png

# Using the command design pattern

The command design pattern generally involves around encapsulating a command as an object. This is highly used in networking for games, in which the player movements are sent across as objects which are run as commands. The 4 main points to remember in a command design pattern are the client, invoker, receiver, command. The command object has knowledge about the receiver object. The receiver does the work after it receives a command. The invoker performs the command, without having any knowledge of who has sent the command. The client controls the invoker and decides which commands are to be performed at which stage.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to implement the command pattern.
2. Open Visual Studio.
3. Create a new C++ project console application
4. Add the following lines of code.

#include <iostream>

#include <conio.h>

using namespace std;

class NetworkProtocolCommand

{

public:

virtual void PerformAction() = 0;

};

class ServerReceiver

{

public:

void Action()

{

cout << "Network Protocol Command received" <<endl;

}

};

class ClientInvoker

{

NetworkProtocolCommand \*m\_NetworkProtocolCommand;

public:

ClientInvoker(NetworkProtocolCommand \*cmd = 0) : m\_NetworkProtocolCommand(cmd)

{

}

void SetCommad(NetworkProtocolCommand \*cmd)

{

m\_NetworkProtocolCommand = cmd;

}

void Invoke()

{

if (0 != m\_NetworkProtocolCommand)

{

m\_NetworkProtocolCommand->PerformAction();

}

}

};

class MyNetworkProtocolCommand : public NetworkProtocolCommand

{

ServerReceiver \*m\_ServerReceiver;

public:

MyNetworkProtocolCommand(ServerReceiver \*rcv = 0) : m\_ServerReceiver(rcv)

{

}

void SetServerReceiver(ServerReceiver \*rcv)

{

m\_ServerReceiver = rcv;

}

virtual void PerformAction()

{

if (0 != m\_ServerReceiver)

{

m\_ServerReceiver->Action();

}

}

};

int main()

{

ServerReceiver r;

MyNetworkProtocolCommand cmd(&r);

ClientInvoker caller(&cmd);

caller.Invoke();

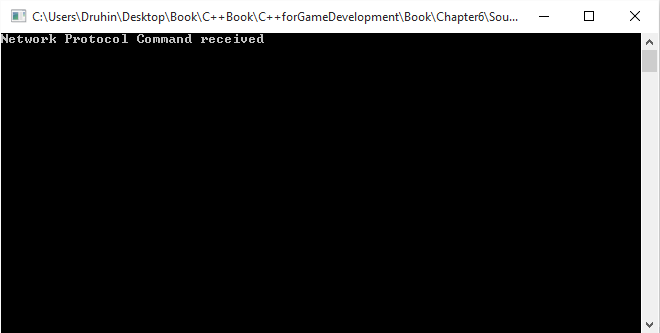
\_getch();

return 0;

}

## How it works...

As we can see in this example, we have a setup an interface to send information via the network protocol command. From that interface we can derive multiple child instances to be used from the client side. We then need to create a server receiver which will receive the commands sent from the client. We also need to create a client invoker which will invoke the command. A reference of the network protocol command should also be present in this class. Finally, from the client side, we need to create an instance of the server and attach the instance to the object of the network protocol command’s child which we created. We then take the help of the client invoker to invoke the command and send it via the network protocol command to the receiver. This ensures that an abstraction is maintained and also the entire message is send via packets. The following image explains a part of the process.



Insert Image B04929\_06\_06.png