girmiti software™

Introduction to C++

- Design Pattern

  - Singleton Design Pattern
  - Factory Design Pattern
  - Abstract Design Pattern
  - Adapter Design Pattern

Design Pattern

Design Pattern:

Design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Advantage of design pattern:

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.

Singleton Design Pattern

Singleton Design Pattern:

The Singleton pattern ensure a class has only one instance, and provide a global point of access to it.
Singleton pattern limits the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system

Advantage of Singleton design pattern:

Saves memory because object is not created at each request. Only single instance is reused again and again.

Singleton Design Pattern

How to create Singleton design pattern:

To create the singleton class, we need to declare the instance as a private static data member and provide a public static member function that encapsulates all initialization code and provides access to the instance.

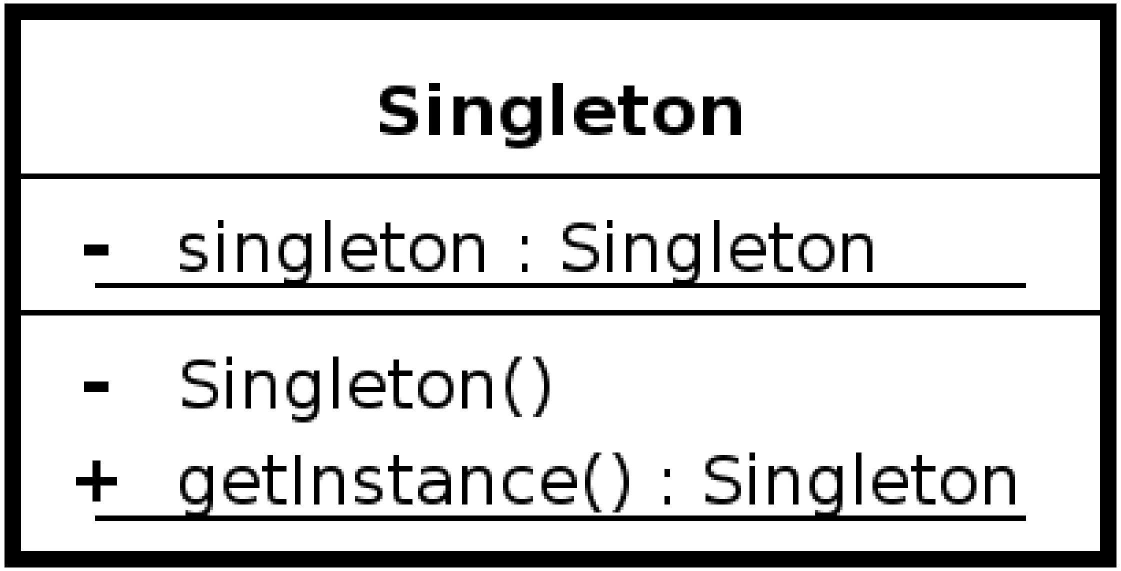
- Define a private static attribute in the "single instance" class.
- Define a public static accessor function in the class.
- Do creation on first use in the accessor function.
- Define all constructors to be private.
- Clients may only use the accessor function to manipulate the Singleton.

There are two forms of singleton design pattern:

Early Instantiation: creation of instance at load time.

Lazy Instantiation**:** creation of instance on first use.

Singleton Design Pattern



girmiti software™

Singleton Design Pattern

<u>Example of Singleton Design pattern(Lazy instantiation):</u>

```
#include <iostream>
using namespace std;
class Singleton
{
public:
        static Singleton *getInstance();
        void display();
private:
        Singleton(){}
        static Singleton* instance;
};
Singleton* Singleton::instance = 0;
Singleton* Singleton::getInstance()
{
        if(!instance) {
                instance = new Singleton();
                cout << "getInstance(): First instance\n";
                return instance;
        }
```

Singleton Design Pattern

```
else {                    cout << "getInstance(): previous instance\n";
                          return instance;
          }
}
void Singleton::display()
{
          cout<<"Display function call"<<endl;

}
int main()
{
          Singleton *s1 = Singleton::getInstance();
          Singleton *s2 = Singleton::getInstance();
          s1->display();
          s2->display();
}
```

Output:

```
getInstance(): First instance
getInstance(): previous instance
Display function call
Display function call
```

Singleton Design Pattern

<u>Example of Singleton Design Pattern(Early instantiation):</u>

```
#include<iostream>
using namespace std;
class Singleton
{
        private:
         static Singleton instance;
         Singleton()
        {

        }
        public:
         static Singleton getInstance()
        {
                return instance;
        }
         void doSomething()
        {
                cout<<"doSomething(): Singleton does something!";
        }
};
```

Singleton Design Pattern

```
int main()
{
        Singleton s=Singleton::getInstance();
        s.doSomething();
}
```

Factory Design Pattern

Factory Design Pattern:

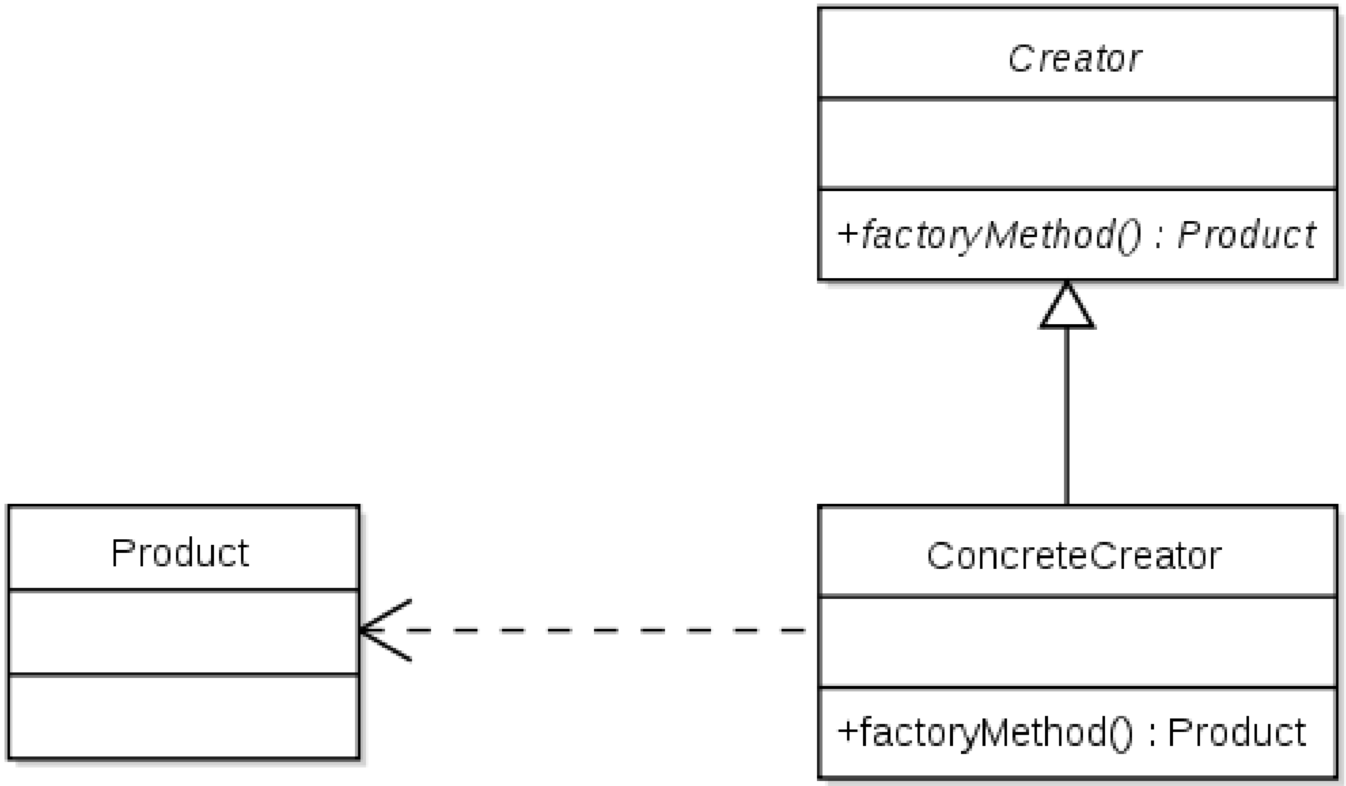
A factory method pattern is a design pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

This is done by creating objects by calling a factory method.

A factory method is a static method of a class that returns an object of that class' type.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.

Factory Design Pattern



girmiti software™

Factory Design Pattern

```
#include <iostream>
using namespace std;
class Bank
{
public:
        virtual void getInt() = 0;
};
class Canara: public Bank
{
public:
        void getInt() {
                std::cout << "Interest rate is 3% \n";
        }
};
 class Axis: public Bank
{
public:
        void getInt() {
                std::cout << "Interest rate is 4%\n";
        }
};
```

girmiti software™

Factory Design Pattern

```
class Hdfc: public Bank
{public:
		void getInt() {
				std::cout << "Interest rate is 5%\n";
		}
};
class Factory{
public:
		Bank* factoryMethod(string type)
		{		if((type.compare("Canara"))==0)
				{
						return new Canara;
				}
				else if((type.compare("Axis"))==0)
				{
						return new Axis;
				}
				else if((type.compare("Hdfc"))==0)
				{
						return new Hdfc;
				}
} };
```

Factory Design Pattern

```
int main()
{
Bank *b;
Factory f;
string bname;
cout<<"Enter bank name";
cin>>bname;
b=f.factoryMethod(bname);
b->getInt();
}
```

Abstract Design Pattern

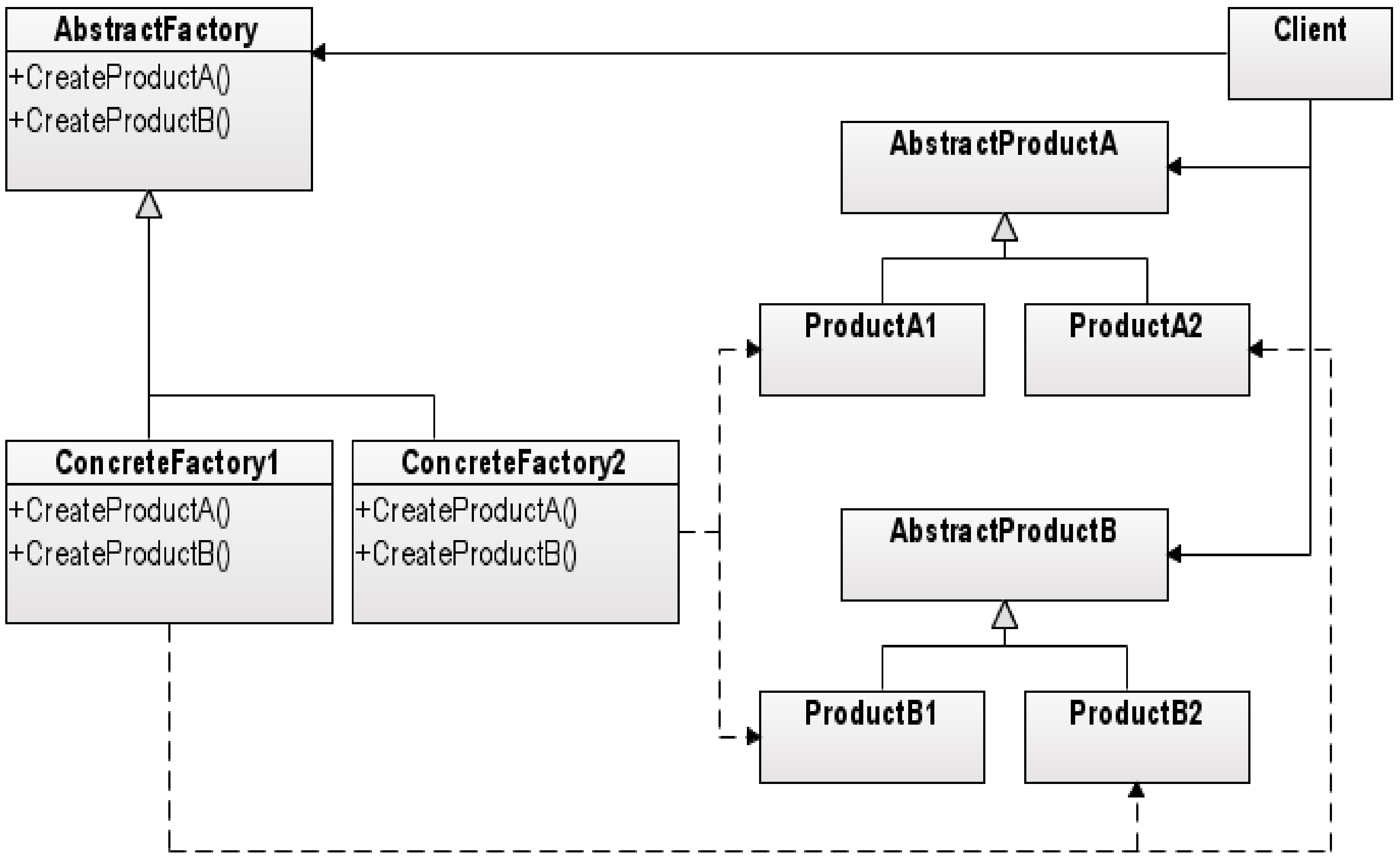
Abstract Design Pattern:

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

Abstract factory pattern in useful when the client needs to create objects which are somehow related.

There are times when the client code does not know the exact type of object (out of many) it need to create and this pattern solves the problem by providing the client with a uniform interface to create multiple type of objects that are related by common theme.

Abstract Design Pattern



Abstract Design Pattern

Let's see what each class does here:

AbstractFactory:
declares an interface for operations that create abstract products.

ConcreteFactory:
implements the operations to create concrete product objects.

AbstractProduct:
declares an interface for a type of product object.

Product:
defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface.

Client:
uses interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Design Pattern

Example:

```
#include <iostream>
using namespace std;
class Button
{public:   virtual void paint() = 0;
};
class WinButton : public Button
{public:   void paint ()
           {   std::cout << " Window Button \n";  }
};
class MacButton : public Button
{public:   void paint ()
           {   std::cout << " Mac Button \n";    }
};
class iPhoneButton : public Button
{public:   void paint ()
           {    std::cout << " iPhone Button \n";    }
};
class ScrollBar
{public:   virtual void paint() = 0;
};
```

Abstract Design Pattern

```
class WinScrollBar : public ScrollBar
{public:
        void paint ()
        {   std::cout << " Window ScrollBar \n";    }
};
class MacScrollBar : public ScrollBar
{public:
        void paint ()
        {   std::cout << " Mac ScrollBar \n";    }
};
class iPhoneScrollBar : public ScrollBar
{
public:
        void paint ()
        {   std::cout << " iPhone ScrollBar \n";  }
};
class GUIFactory
{public:
        virtual Button* createButton () = 0;
        virtual ScrollBar* createScrollBar () = 0;
};
```

Abstract Design Pattern

```
class WinFactory : public GUIFactory
{
public:
            Button* createButton ()
            {   return new WinButton;    }
            ScrollBar* createScrollBar ()
            {   return new WinScrollBar;   }
};
 class MacFactory : public GUIFactory
{public:
            Button* createButton ()
            {  return new MacButton;  }
            ScrollBar* createScrollBar ()
            {  return new MacScrollBar;  }
};
class iPhoneFactory : public GUIFactory
{public:
            Button* createButton ()
            {   return new iPhoneButton;  }
            ScrollBar* createScrollBar ()
             {  return new iPhoneScrollBar;  }
};
```

girmiti software™

Abstract Design Pattern

```
int main()
{
        GUIFactory* guiFactory;
        Button *btn;
        ScrollBar *sb;
        guiFactory = new MacFactory;
        btn = guiFactory->createButton();
        btn -> paint();
        sb = guiFactory->createScrollBar();
        sb -> paint();
        guiFactory = new WinFactory;
        btn = guiFactory->createButton();
        btn -> paint();
        sb = guiFactory->createScrollBar();
        sb -> paint();
        guiFactory = new iPhoneFactory;
        btn = guiFactory->createButton();
        btn -> paint();
        sb = guiFactory->createScrollBar();
        sb -> paint();
}
```

Adapter Design Pattern

<u>Adapter Design Pattern:</u>

As we all are familiar with adapters, we use adapters to connect two incompatible devices.

In design pattern term, the same is it's functionality.Using adapters we will convert interface of one class into another that the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter design pattern is also known as wrappers.

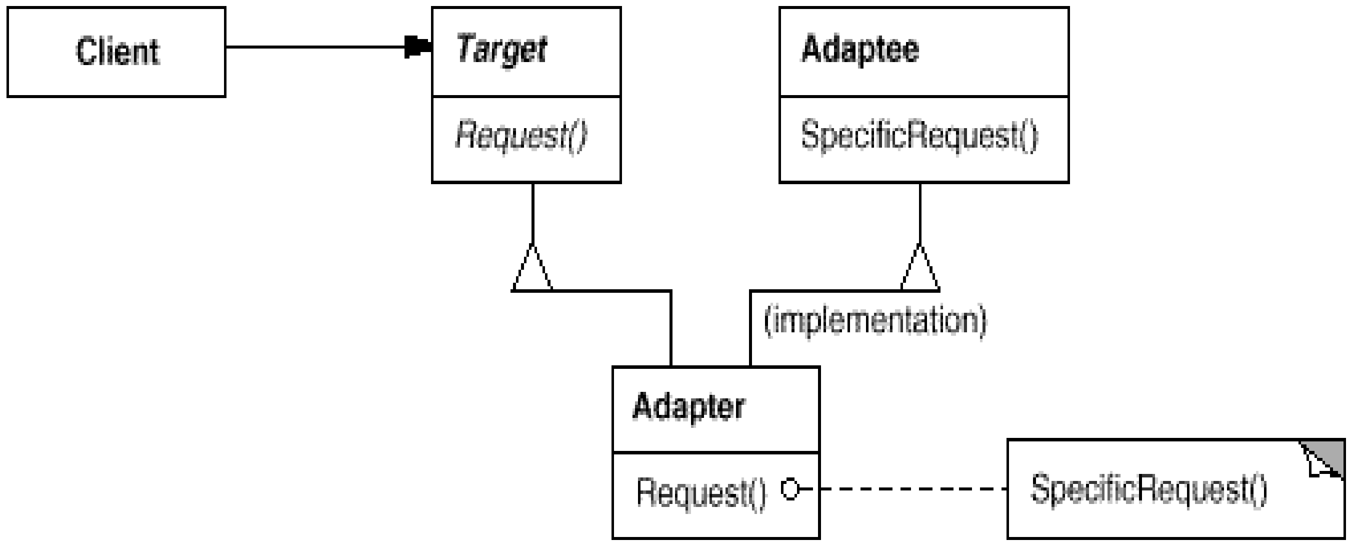
<u>It comprises three components:</u>

<u>Target</u>:
This is the interface with which the client interacts.

<u>Adaptee</u>:
This is the interface the client wants to interact with, but can't interact without the help of the Adapter.

<u>Adapter</u>:
This is derived from Target and contains the object of Adaptee.

Adapter Design Pattern



girmiti software™

Adapter Design Pattern

Example:

```
#include <iostream>
// Desired interface (Target)
class Rectangle
{
  public:
    virtual void draw() = 0;
};
// Legacy component (Adaptee)
class LegacyRectangle
{
 private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
  public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
```

girmiti software™

Adapter Design Pattern

```
 std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
};
// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(int x, int y, int w, int h):LegacyRectangle(x, y, x + w, y + h) {
         std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

Adapter Design Pattern

```
int main()
{
  int x = 20, y = 50, w = 300, h = 200;
  Rectangle *r = new RectangleAdapter(x,y,w,h);
  r->draw();
}
```

Adapter Design Pattern

Example:

```
// Abstract Target
#include<iostream>
using namespace std;
class AbstractPlug {
public:
  void virtual RoundPin(){}
  void virtual PinCount(){}
};
// Concrete Target
class Plug : public AbstractPlug {
public:
  void RoundPin() {
    cout << " I am Round Pin" << endl;
  }
  void PinCount() {
    cout << " I have two pins" << endl;
  }
};
```

girmiti software™

Adapter Design Pattern

```
// Abstract Adaptee
class AbstractSwitchBoard {
public:
  void virtual FlatPin() {}
  void virtual PinCount() {}
};
// Concrete Adaptee
class SwitchBoard : public AbstractSwitchBoard {
public:
  void FlatPin() {
        cout << " Flat Pin" << endl;
  }
  void PinCount() {
        cout << " I have three pins" << endl;
  }
};
// Adapter
class Adapter : public AbstractPlug {
public:
```

Adapter Design Pattern

```
AbstractSwitchBoard *T;
 Adapter(AbstractSwitchBoard *TT) {
       T = TT;
 }
 void RoundPin() {
       T->FlatPin();
 }
 void PinCount() {
       T->PinCount();
 }
};
// Client code
int main()
{
 SwitchBoard *mySwitchBoard = new SwitchBoard; // Adaptee
 // Target = Adapter(Adaptee)
 AbstractPlug *adapter = new Adapter(mySwitchBoard);
 adapter->RoundPin();
 adapter->PinCount();
}
```

Discussions