Factory Method in C++

Back to **Factory Method** description

Another Factory Method C++ source code example

Discussion. Frameworks are applications (or subsystems) with "holes" in them. Each framework specifies the infrastructure, superstructure, and flow of control for its "domain", and the client of the framework may: exercise the framework's default behavior "as is", extend selected pieces of the framework, or replace selected pieces.

The Factory Method pattern addresses the notion of "creation" in the context of frameworks. In this example, the framework knows WHEN a new document should be created, not WHAT kind of Document to create. The "placeholder" `Application::CreateDocument()` has been declared by the framework, and the client is expected to "fill in the blank" for his/her specific document(s). Then, when the client asks for `Application::NewDocument()`, the framework will subsequently call the client's `MyApplication::CreateDocument()`.

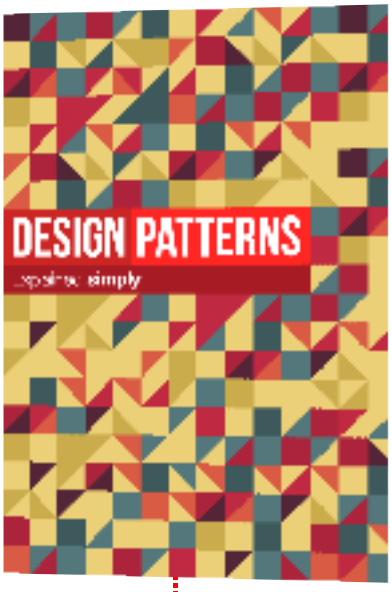
```
#include <iostream.h>

/* Abstract base class declared by framework */
class Document
{
  public:
    Document(char *fn)
    {
        strcpy(name, fn);
    }
    virtual void Open() = 0;
    virtual void Close() = 0;
    char *GetName()
    {
        return name;
    }
  private:
    char name[20];
};

/* Concrete derived class defined by client */
class MyDocument: public Document
{
  public:
    MyDocument(char *fn): Document(fn){}
    void Open()
    {
        cout << "   MyDocument: Open()" << endl;
    }
    void Close()
    {
        cout << "   MyDocument: Close()" << endl;
    }
};

/* Framework declaration */
class Application
{
  public:
    Application(): _index(0)
    {
        cout << "Application: ctor" << endl;
    }
    /* The client will call this "entry point" of the framework */
    NewDocument(char *name)
    {
        cout << "Application: NewDocument()" << endl;
        /* Framework calls the "hole" reserved for client customization */
```

```
        _docs[_index] = CreateDocument(name);
        _docs[_index++]->Open();
    }
    void OpenDocument(){}
    void ReportDocs();
    /* Framework declares a "hole" for the client to customize */
    virtual Document *CreateDocument(char*) = 0;
  private:
    int _index;
    /* Framework uses Document's base class */
    Document *_docs[10];
};

void Application::ReportDocs()
{
  cout << "Application: ReportDocs()" << endl;
  for (int i = 0; i < _index; i++)
    cout << "   " << _docs[i]->GetName() << endl;
}

/* Customization of framework defined by client */
class MyApplication: public Application
{
  public:
    MyApplication()
    {
        cout << "MyApplication: ctor" << endl;
    }
    /* Client defines Framework's "hole" */
    Document *CreateDocument(char *fn)
    {
        cout << "   MyApplication: CreateDocument()" << endl;
        return new MyDocument(fn);
    }
};

int main()
{
  /* Client's customization of the Framework */
  MyApplication myApp;

  myApp.NewDocument("foo");
  myApp.NewDocument("bar");
  myApp.ReportDocs();
}
```

Output

```
Application: ctor
MyApplication: ctor
Application: NewDocument()
   MyApplication: CreateDocument()
   MyDocument: Open()
Application: NewDocument()
   MyApplication: CreateDocument()
   MyDocument: Open()
Application: ReportDocs()
   foo
   bar
```

Read next



This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.

♥ Learn more

Code examples

Java	**Factory Method in Java**	
C++	**Factory Method in C++: Before and after**	**Factory Method in C++**

PHP	**Factory Method in PHP**
Delphi	**Factory Method in Delphi**

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us