RZ 2787 (# 89265) 02/05/96
Computer Science/Mathematics 5 pages

Research Report

Abstract Factory Revisited

Bela Ban, Luca Deri

IBM Research Division
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Abstract Factory Revisited

Object Creational

Bela Ban, Luca Deri

IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Motivation. The AbstractFactory (87) pattern [2] presents to clients a collection of related, abstract interfaces to manipulate. The implementation is provided by concrete subclasses created by a factory which decides which instances to create. The client doesn't know the subclasses, but the factory has to know all the concrete subclasses. Rather than linking with the factory library which includes all concrete subclasses, the factory is moved to DLLs.[1] This means that factories can be added and replaced on the fly, without the client having to be recompiled or relinked with the factory libraries. Changes to the functionality of a system can therefore be easily done and an incremental development style is supported.

[1] DLL stands for dynamic link library.

Intent

Offer an abstract interface to clients and provide the implementation through subclasses dynamically loaded from DLLs (Components).

Also Known As

Components [1].

Applicability

Use this pattern to structure your code into several, small, independently replaceable and modifiable components. to write applications that may need to accommodate future extensions.

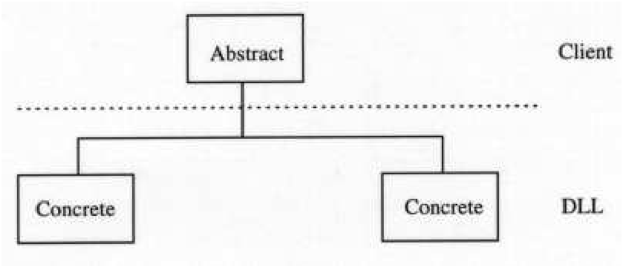

Figure 1: Structure.

Participants

- Client sees only the abstract class(es) and its defined interfaces.
- Implementation of each abstract class by a concrete subclass in a DLL.

Collaborations

The client loads the DLLs to create concrete subclasses of the abstract classes.

Consequences

1. Faster development cycle. A component can be modified without having to relink with the entire application (i.e. object modules) again.

2. The application can be extended/modified/repaired without having to kill it but just adding/replacing DLLs on fly.

3. Bug fixes can be sent to customers without having to send the entire executable. Rather, a new version of the defect component is sent to the customer, who only has to replace the broken part with the new component.

4. Locality of computation. Rather than being distributed in several places in an application, computation (algorithms) that belongs together are in the same component.

5. Few interdependencies. Components have a well-defined interface, their implementation is hidden and clients manipulate them only through their abstract interface. When we think of a concrete subclass as a *contractor* to the client and of interfaces that the concrete subclass itself uses as *subcontractors*, then the client clearly doesn't need to know the subcontractors. Having the subcontractors hidden in DLLs removes the dependency between them and clients, i.e. clients don't have to be recompiled when subcontractors change.

Implementation

One or more concrete subclasses are implemented in a DLL. For each subclass there has to be a creation-function that returns an instance. The structure of the DLL is prescribed and all DLLs have to obey it. In the case where one DLL is provided for each class,[2] the entry point into the DLL is a function that returns an array of 3 function pointers. The first function pointer points to an initialization function that is called by the client when the DLL is loaded into memory. The second is called when the DLL is unloaded and the third is the creation-function.[3] The client is responsible for loading the DLL and finding and calling the correct creation-function. For this purpose, a generic creation-function could be provided to the client that e.g. loads all DLLs at program startup time and, given the name of a class to be instantiated, would return the correct concrete subclass. There are several schemes that are possible, also depending whether one or many subclasses are in a DLL.

Sample Code

The following C++ code shows how the pattern concepts can be applied. The client knows only the abstract base class A. The concrete subclass B is implemented in the DLL. Two different implementations of the B class are shown. This is in order to show how the concrete class behaviour can be changed at run-time. Despite the fact that this is a general example, this code uses two OS depended functions, namely load and unload. In order to run the code on a platform different than IBM AIX™, the functions `LoadDll` and `UnloadDll` have probably to be modified.

Abstract Base Class Definition (included by client) [a.h]

```
class A {
  int   a;
  char* name;
public:
  virtual void foo(); // No instances of A should be created
  virtual void bar();
};
```

[2] Of course other schemes are possible. A DLL may e.g. have as entry point a dictionary with pairs of class names and addresses of creation-functions so that many classes can be accommodated in DLL.

[3] The creation-function is an abstract function.

Concrete Subclass Definition [b.h]

```
#include "a.h"
#include <iostream.h>

class B : public virtual A {
  int b;
public:
  void bar();
};
```

Concrete Subclass Implementation (in DLL) [b1.C]

```
#include "b.h"

void B::bar() { cout << "Method B::bar() called from b1.dll." << endl; }
A* Create()   { return((A*)new B); } // Create Function and DLL Entry Point
```

Concrete Subclass Implementation (in DLL) [b2.C]

```
#include "b.h"

void B::bar() { cout << "Method B::bar() called from b2.dll." << endl; }
A* Create()   { return((A*)new B); } // Create Function and DLL Entry Point
```

Abstract Base Class Implementation [a.C]

```
#include "a.h"
#include <iostream.h>

void A::foo() { cout << "A::foo()" << endl; }
void A::bar() { cout << "A::bar()" << endl; }
```

Client code [main.C]

```
#include <stdio.h>
#include <iostream.h>
#include <sys/ldr.h>
#include "a.h"  // client 'knows' only class A !

void* LoadDll(char* dllName) {
  void* dllHandle;

  dllHandle = load(dllName, 1, "./");
  if(!dllHandle) cerr << "DLL entry point not found !" << endl;
  return dllHandle;
}

void UnloadDll(void* dllHandle) { (void)unload(dllHandle);}
```

```
int main(int argc, char* argv[]) {
  A *a, *(*CreateFunction)();

  cout << "Loading b1.dll...\n\n";
  if((CreateFunction = (A* (*)())LoadDll("b1.dll")) == NULL)
    return -1;

  a = CreateFunction(); // Actually returns an instance of class B
  a->foo();             // Prints: 'A::foo()'
  a->bar();             // Prints: 'Method B::bar() called from b1.dll'
  UnloadDll((void*)CreateFunction);

  cout << "\nLoading b2.dll...\n"; // Let's now change the behaviour of the B class
  if((CreateFunction = (A* (*)())LoadDll("b2.dll")) == NULL)
    return -1;
  a->bar();  // Prints: 'Method B::bar() called from b1.dll'
  delete a;
  UnloadDll((void*)CreateFunction);
  return 0;
}
```

Makefile (for IBM AIX™)

```
CC              = xlC
MAKE_SH_LIB     = makeC++SharedLib

.SUFFIXES: .C .o .dll

.C.o:;    $(CC) -c -g $<
.o.dll:;  $(MAKE_SH_LIB) -o $@ -n Create__Fv -p 1024 $< a.o


all: main b1.dll b2.dll

main: main.o a.o
        xlC -g a.o main.o -o main

b1.dll: b1.o a.o
b2.dll: b2.o a.o
```

Known Uses

Some applications and research prototypes developed at the IBM Zurich Development Laboratory use this pattern. A few commercial applications use *tools* (DLLs), located in a specific directory, to enhance its core functionality at startup-time; as far as we know there are no applications able to modify/extend its behaviour at runtime by using DLLs.

Related Patterns

This pattern is an enhancement to Abstract Factory (87) in that the AbstractFactory (creation-function) is in a DLL and the subclasses are created from it. The fact that the AbstractFactory can be replaced on the fly by providing a new DLL make this pattern similar to a virtual constructor (Factory Method (107)).

Authors

Bela Ban (bba@zurich.ibm.com) and Luca Deri (lde@zurich.ibm.com) IBM Zurich Research Laboratory, Säumerstrasse 4, CH–8803 Rüschlikon, Switzerland.

References

[1] Deri, Luca. Breaking Monolithic Applications Apart. Internal Report. IBM Zurich Research Laboratory, August 1995.

[2] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.