
8.1. The Need for Object Factories

There are two basic cases in which object factories are needed. The first occurs when a library needs not only to manipulate user-defined objects, but also to create them. For example, imagine you develop a framework for multiwindow document editors. Because you want the framework to be easily extensible, you provide an abstract class `Document` from which users can derive classes such as `TextDocument` and `HTMLDocument`. Another framework component may be a `DocumentManager` class that keeps the list of all open documents.

A good rule to introduce is that each document that exists in the application should be known by the `DocumentManager`. Therefore, creating a new document is tightly coupled with adding it to `DocumentManager`'s list of documents. When two operations are so coupled, it is best to put them in the same function and never perform them separately:

Click here to view code image

```
class DocumentManager
{
   ...
public:
   Document* NewDocument();
   virtual Document* CreateDocument() = 0;
private:
   std::list<Document*> listOfDocs_;
};

Document* DocumentManager::NewDocument()
{
   Document* pDoc = CreateDocument();
   listOfDocs_.push_back(pDoc);
   ...
   return pDoc;
}
```

The `CreateDocument` member function replaces a call to `new`. `NewDocument` cannot use the `new` operator because the concrete document to be created is not known by the time `DocumentManager` is written. In order to use the framework, programmers will derive from `DocumentManager` and override the virtual member function `CreateDocument` (which is likely to be pure). The GoF book (Gamma et al. 1995) calls `CreateDocument` a factory method.

Because the derived class knows exactly the type of document to be created, it can invoke the `new` operator directly. This way, you can remove the type knowledge from the framework and have the framework operate on the base class `Document` only. The override is very simple and consists essentially of a call to `new`; for example:

Click here to view code image

```
Document* GraphicDocumentManager::CreateDocument()
{
   return new GraphicDocument;
}
```

Alternatively, an application built with this framework might support creation of multiple document types (for instance, bitmapped graphics and vectorized graphics). In that case, the overridden `CreateDocument` function might display a dialog to the user asking for the specific type of document to be created.

Thinking of opening a document previously saved on disk in the framework just outlined brings us to the second—and more complicated—case in which an object factory may be needed. When you save an object to a file, you must save its actual type in the form of a string, an integral value, an identifier of some sort. Thus, although the type information exists, its form does not allow you to create C++ objects.

The general concept underlying this situation is the creation of objects whose type information is genuinely postponed to runtime: entered by the end user, read from a persistent storage or network connection, or the like. Here the binding of types to values is pushed even further than in the case of polymorphism: When using polymorphism, the entity manipulating an object does not know its exact type; however, the object itself is of a well-determined type. When reading objects from some storage, the type comes “alone” at runtime. You must transform type information into an object. Finally, you must read the object from the storage, which is easy once an empty object is created, by invoking a virtual function.

Creating objects from “pure” type information, and consequently adapting dynamic information to static C++ types, is an important issue in building object factories. Let’s focus on it in the next section.