Private Interface **Class Structural**

James Newkirk
newkirk@oma.com

Intent

Provide a mechanism that allows specific classes to use a non-public subset of a class interface without inadvertently increasing the visibility of any hidden member variables or member functions.

Also Known As

I finally found a use for private inheritance.

Motivation

Some applications benefit from the use of multiple threads in order to accomplish various tasks. For example, a user interface is better implemented using threads because it allows the application to respond to the user even when some processing-intensive-task is executing. Nothing can be more frustrating than initiating a processing-intensive-task (i.e. incremental backup of a document) and having the whole application freeze while that task is going on. Figure 1 is a UML[1] class diagram that describes a potential solution to this problem.

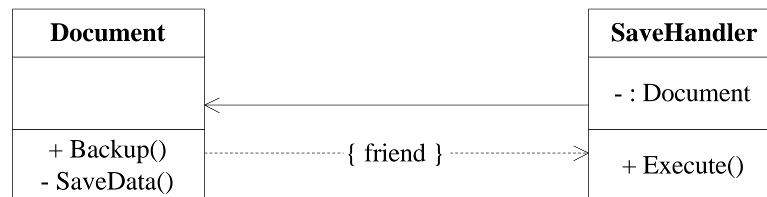


Figure 1: Initial Solution

In this solution there is a class called `Document`, which is responsible for holding the data that the program is working on. This class also has a public member function called `Backup()` which clients use to save the Document's data. In order to insure that the saving of the data is performed in a different thread than the calling thread the Backup() member function creates a SaveHandler object. The `SaveHandler` object is responsible for creating the new thread and then calling `SaveData()` in the `Document` class to save the data. The `SaveData()` member function needs to be inaccessible to the clients of `Document` otherwise they could short circuit this

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. Universal Modeling Language V1.0: The Notation. http://www.rational.com

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

mechanism and call SaveData() directly. This would be unacceptable because then the saving of the data would be done in the context of the Client's thread.

The mechanism employed to control visibility in this manner is specific to the programming language that is used. For example, in C++ the SaveData() member function can be declared private. Since the SaveHandler class must be able to call the SaveData() member function, SaveHandler must be declared a friend of the Document class. This allows the SaveHandler class to call the private SaveData() member function.

The positive aspects of this approach are that for each Backup() request that a Document object receives it creates a new object that spawns a separate thread and then runs the call to the SaveData() member function in this newly created thread. This moves an almost certain dependency on a thread library or a particular OS out of the Document class and into the SaveHandler class. This allows the Document class to potentially be reused in different environments without change related to this activity.

The downside of this approach is related to the break in encapsulation that is required to allow the SaveHandler class access to the private member function SaveData(). This break in encapsulation could be justified since these two classes (Document and SaveHandler) are intimately associated with one another. However, when friendship is declared access to all member variables and member functions is allowed. This can lead to maintenance problems in the future. In this example, not only can a SaveHandler object call SaveData() it can also access any member variable or any other member function inside of Document. These maintenance problems are often not foreseen by the original designer. Maintenance programmers, who are often chartered with task of fixing difficult problems without the benefit of understanding the architecture of the application, exploit such loopholes. Once the door has been open to allow classes to be defined as friends it becomes easy to start adding more and more friends. What is left over are classes whose encapsulation boundaries are severely compromised.

The Private Interface pattern describes how to retain the positive aspects of the above solution while eliminating the negative aspects. Figure 2 describes the class structure for the above mentioned problem recast utilizing the Private Interface pattern.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

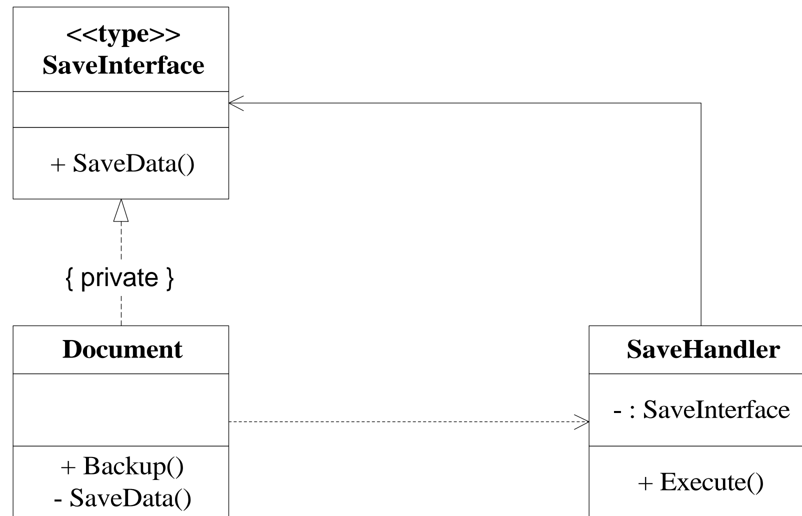


Figure 2: Private Interface Solution

In Figure 2 a new class has been introduced called `SaveInterface`. The `SaveInterface` class is a <<type>> (i.e. a class with no implementation, all its methods are abstract). It has one public member function called `SaveData()`. The `Document` class from the initial solution now implements the `SaveInterface` type in a manner which keeps the `SaveData()` method from being accessible to clients of `Document`. For example, in C++ the `Document` class would *privately* inherit from `SaveInterface`.

Notice that, in this solution, `SaveHandler` does not have a navigable association with the `Document` class. This has been replaced with a navigable association to the `SaveInterface` class. Since `SaveData()` is declared publicly in the `SaveInterface` class `SaveHandler` no longer needs to be declared as a `friend` of the `Document` class.

Since the `SaveHandler` class is no longer a `friend` of the `Document` class it does not have the ability to change any of its private member variables or call any of its private member functions. So the encapsulation of the `Document` class has been preserved. The `SaveHandler` class no longer even needs to know about the `Document` class since it now depends on an interface called `SaveInterface`. This reduces the overall dependencies in the design and creates a potential for reuse.

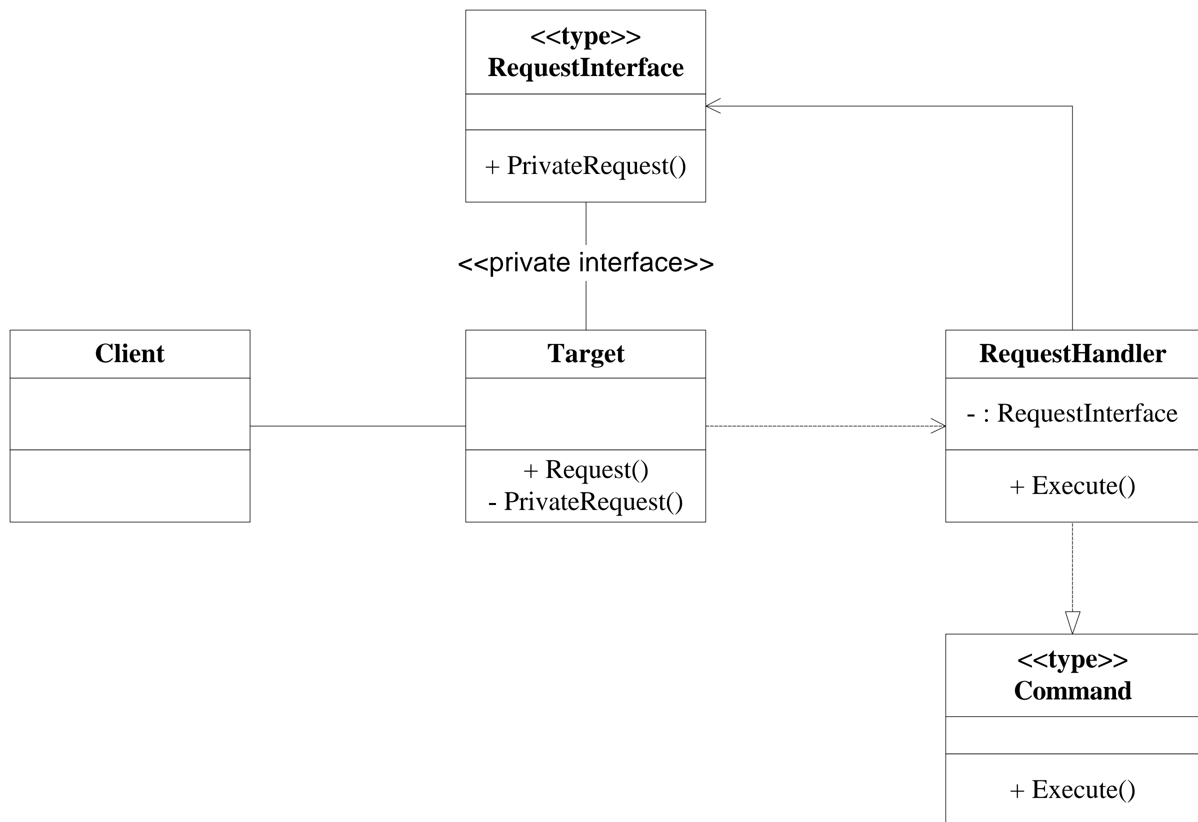
Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Applicability

Apply the Private Interface pattern when any of the following are true:

- In C++, a class A has declared that another class F as a friend so that class F can access a private or protected member function of A.
- In Java, when a member function has been declared protected in order to allow other classes in the package to call it.

Structure



Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Participants

- **Target** (`Document`)
  - defines the public interface for the client.
  - constructs the `RequestHandler` (`SaveHandler`) object and passes to the `RequestHandler` an instance of the `RequestInterface` (`SaveInterface`) class.
  - in the motivating example, the `Document` class exports an interface that allows clients to start asynchronous saves.
- **Client**
  - clients use the exported `Target` interface to perform specific functions.
- **Command**[2]
  - the `Command` interface is employed because it is likely that the `Target` interface will want to create many different kinds of `RequestHandler` objects. If all of these objects implement the `Command` interface then a Factory[2] could be used to create the individual `RequestHandler` objects.
- **RequestHandler** (`SaveHandler`)
  - created with a reference to the `RequestInterface`.
  - implements the `Command` interface so the `Target` class can call the `Execute()` member function to perform the request.
  - in the motivating example, the `SaveHandler` class is used to create a separate thread of execution and then call a member function defined in the `SaveInterface` to actually save the data in the `Document` in the newly created thread.
- **RequestInterface** (`SaveInterface`)
  - specifies an abstract interface for a particular request.

---

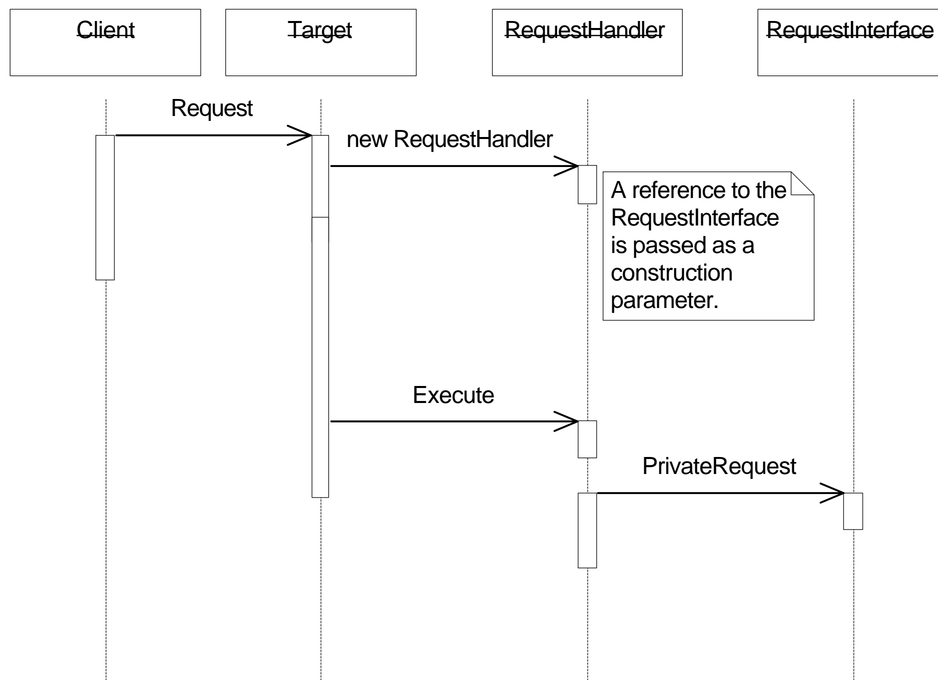
[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Collaborations

- The client directs the `Target` class to perform the request.
- The `Target` class creates the `RequestHandler` object and passes an instance of `Target`'s private interface, `RequestInterface`.
- `Target` calls `Execute()` to perform the request.
- `RequestHandler` performs any local operations (i.e. creation of a thread) and then calls the `PrivateRequest` member function defined in the `RequestInterface`.

The following sequence diagram illustrates how `Target` and `RequestHandler` cooperate in order to perform the request.



Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Consequences

1. *Preserves the encapsulation of the Target class.* When a class is granted friendship or a member function is declared protected it potentially allows other classes to access hidden members of the class. This pattern hides the Target internals, thereby preserving the encapsulation boundary.
2. *It provides a mechanism to reduce overall dependencies in the implementation.* In the `Document` example the `Target` class depends on the `SaveHandler` class. And the `SaveHandler` class depends on the `Document` class. Also, in the motivating example the `SaveHandler` class runs the call to `SaveData` in a newly created thread. Had this been implemented inside the `Document` class there would have been a dependency on a threads package. In order to manage the dependencies in this design the introduction of the `SaveInterface` class provides the opportunity to move the thread dependency out of the `Document` class and place it in the `SaveHandler` class. Also, the `SaveHandler` class does not depend on the `Document` class, it depends on the `SaveInterface` class. This isolates the `SaveHandler` from the `Document` class and allows the `Document` class to change without adverse impact on the `SaveHandler`.
3. *Increases the use of multiple inheritance.* In programming languages that implement the pattern using private inheritance (Like C++) there is an increase in the use of multiple inheritance. Each request needs a corresponding interface for the specific member function. In C++ this leads the `Target` class to multiply and privately inherit each of the interfaces. Since this is inheritance of interface and not implementation there is no need for virtual inheritance. One alternative is to create a single interface class to represent all of the private interface member functions. However, this allows all of the clients of the interface to access any of the other member functions, which decreases the overall benefit of the pattern.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Implementation and Sample Code

The following are two implementations of the Private Interface pattern. Each is an implementation of the `Document/SaveHandler` example described in the Motivation section. These examples each demonstrate a distinct approach to the implementation of the private interface pattern. The C++ example implements the interface using private inheritance. The Java example implements the private interface using anonymous classes, which are part of Java 1.1.

C++ Example

The following example provides the C++ code for the `Document/SaveHandler` example described in the Motivation section. This example has removed the thread creation code in the `SaveHandler` to show only the collaboration between the classes. The place where the thread creation code would be is commented in the example.

Interfaces

First we define the `SaveInterface` class which provides the interface that the `SaveHandler` will use to call the `SaveData()` member function.

```
class SaveInterface
{
  public:
      SaveInterface();
      virtual ~SaveInterface();

      virtual void SaveData() const = 0;
};
```

This class has one pure virtual member function `SaveData()` which must be implemented by derived classes. Notice that the `SaveData()` member function is declared in the public area of the class definition.

The Command class is defined in a similar manner:

```
class Command
{
  public:
      Command();
      virtual ~Command();

      virtual void Execute() = 0;

};
```

Implementers of this interface must provide a definition of the `Execute()` member function.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

SaveHandler

```
#include "Command.h"
class SaveInterface;

class SaveHandler : public Command
{
  public:
      SaveHandler(SaveInterface*);
      virtual ~SaveHandler();

      void Execute();
  private:
      SaveInterface* itsInterface;
};
```

There are a number of interesting items in this header declaration. The first is that the constructor argument is a pointer to `SaveInterface` rather than a pointer to `Document`. This pointer is saved in a private member variable called `itsInterface`, which is shown in the constructor code shown below.

```
SaveHandler::SaveHandler(SaveInterface* anInterface)
: itsInterface(anInterface)
{}
```

The second item of interest is the code in the `Execute()` member function. This is the declaration that satisfies the `Command` interface. The Command pattern is useful in this situation because it is likely that we would use the Factory pattern to create these commands if there were more than one. This code, shown below, indicates where the thread creation would occur and then the call to the `SaveData()` member function which would take place in the context of the newly created thread.

```
void SaveHandler::Execute()
{
  // Here is where the thread would be created in
  // in order to perform the call to SaveData().

  itsInterface->SaveData();
}
```


Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Target Class
The last item of interest is the Document class definition and implementation. The class definition is as follows:

```
#include "SaveInterface.h"
class SaveHandler;

class Document : private SaveInterface
{
  public:
       Document();
       virtual ~Document();

       void Backup() const;

  private:
       void SaveData() const;
       SaveHandler* itsSaveHandler;

};
```

The Document class privately inherits from the SaveInterface class. It also implements the SaveData() member function declared in SaveInterface. Even though the SaveData() member function is declared in the public section of SaveInterface it is declared privately here so clients of Document cannot call SaveData() directly. The implementation of the constructor and the Backup() member function are shown below:

```
Document::Document()
{
  itsSaveHandler = new SaveHandler(this);
}

void Document::Backup() const
{
  itsSaveHandler->Execute();
}
```

The constructor creates the SaveHandler object. It passes itself as the parameter. Since this expression is inside the Document class it is possible to up-cast the this pointer to the private base class, SaveInterface. In the Backup() member function we simply call Execute() to have the backup sequence started in a different thread.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

Java Example

The main point of this example is to demonstrate that, in Java, the private interface can be implemented without forcing the `Target` class to inherit from many different interfaces. Also Java does not permit the overloading of visibility that is required in order to implement the solution in a similar way to the C++ implementation. For a detailed understanding of Inner Classes and Anonymous Classes see the Java 1.1 Documentation[3]. An additional advantage of Java is that the `Target` class does not have to retain the created instances of the `SaveHandler` object since it will be garbage collected when it is no longer required.

Interfaces

The `SaveInterface` and `Command` interface are implemented as follows:

```
interface SaveInterface
{
    public void SaveData();
}

interface Command
{
    public void Execute();
}
```

SaveHandler

The `SaveHandler` class implements the `Command` interface. It is created with a reference to the `SaveInterface` object. Upon construction the object saves this reference in the variable named `itsInterface`.

```
import SaveInterface;
import Command;

class SaveHandler implements Command
{
    private SaveInterface itsInterface;

    SaveHandler(SaveInterface anInterface)
    {
        itsInterface = anInterface;
    }
```

[3] Sun Microsystems, "Inner Classes in Java 1.1," http://www.javasoft.com/products/jdk/1.1/docs/index.html, 1996, for an excellent discussion of Inner class see: Shur J. "Exploring Inner Classes in Java 1.1," *C++ Report* 9(5): May 1997

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

The `Execute()` member is defined as follows:

```
        public void Execute()
        {
            Runnable aRunnable = new Runnable()
            {
                public void run()
                {
                    itsInterface.SaveData();
                }
            };

            Thread t = new Thread(aRunnable, "SaveData");
            t.start();
        }
    }
```

The first thing that is performed in here is that an anonymous class of type `Runnable` is created and held in a variable named `aRunnable`. When `run()` is called on `aRunnable` it will call the `SaveData()` member function. The reason for doing this is that the `Thread` class needs an object of type `Runnable` as a construction parameter. Once `aRunnable` is created we then create a new thread and pass in `aRunnable` which will eventually execute the call to `SaveData()` in the newly created thread.

Target Class

The Target class is defined as follows:

```
import SaveHandler;
import SaveInterface;

class Document
{
    private void Save()
    {
        // this is where the actual data would be
        // saved
    }
```

The member function `Save()` is declared as private and does not have to conform to any specific interfaces.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference

The exported member function `Backup()` is implemented in this manner:

```
    public void Backup()
    {
        SaveInterface si = new SaveInterface()
        {
            public void SaveData()
            {
                Document.this.Save();
            }
        };

       (new SaveHandler(si)).Execute();
    }
}
```

The first object that is created is an instance of the anonymous class that implements `SaveInterface`. The implementation of the `SaveData` member function specifies the call to the private member function `Save()`.

The last task to perform is the creation of the `SaveHandler` object passing in the newly created `SaveInterface` object and calling `Execute()` in order to perform the actual request.

Known Uses

This pattern is used extensively in an embedded real-time environment at Xerox. It is employed in order to break the thread of execution in classes that perform various services for their clients.

Copyright© 1997, James Newkirk. Permission is granted to copy for the PloP-97 conference