Oleg Shteynbuk

Presented at COOTS '97
Advanced Topics Workshop
Software Components: Integration and Collaboration
June 20, 1997

Non-Intrusive Component Design and *Integration*

Abstract

Component design affects component integration into an application in several ways. One of them, the component API, is the subject of this paper. How the component API affects component integration can be seen from the following example. A component can require that to be part of its API the object should be derived from a certain class or should implement a certain interface. A more specific way of communication between a component and its clients involving a collection of objects is our primary focus. In this paper we also discuss how to develop non-intrusive components, i.e., components imposing the least possible restrictions on its clients. Adapters and bridges used to simplify the integration of components are discussed as well. This design is cited in the External Polymorphism pattern as an independent discovery of a variant of the External Polymorphism Design Pattern [PloP96].

Introduction

In the C++ world components are often wrapped around libraries. And very often an application needs several libraries. Library designers usually do not have any prior knowledge of the way their creation is going to be used.

Here we discuss the approach to component design that has been applied to the client data infrastructure library developed in our company.

The client infrastructure library has a collection of pointers to the abstract base class (ABC) Item as part of its API. That creates two problems:

- Different internal applications, which need to be integrated with the infrastructure, use different collection class libraries: some applications use the Rogue Wave collections; some use the IBM collections that come with C Set+, while the others use the STL.
- There are cases when it is not feasible or desirable to make all classes inherit from a single class (we will call it Item) for several reasons - legacy code, third party products, name clashes.

Designing components for use with different collections and providing adapters and bridges

If you need to incorporate a library into an existing application you could either change the application or adapt the library. This problem has a personal touch for the author as one of my assignments was to replace all usages of the Rogue Wave collections with the IBM collections. The reason given for this was that one very important application was using the IBM collections and did not want to deal with any other collection library. From this experience I have realized that common mechanism to operate on any collection type looks very attractive. Ideally, users do not have to know what collection class library is used internally by the infrastructure library; what users care about is the interface with the library and ability to use their favorite collections class library.

In our company we have three different types of collections in use, and people could develop personal attachments to their favorite collections. It is not our purpose to get involved in a religious war over the collection class libraries, but rather to make the infrastructure available to every application.

In the original infrastructure implementation a set of pointers to Item was a part of the API. But it is not an easy decision to choose one collection class library over the other, and after this decision is made there is the next one – to choose a specific collection type from this library; for example, a set or a vector,. In the first implementation the Rogue Wave collections were used; then they were changed to the IBM collections that come with C Set++.

To adapt the infrastructure to different collection libraries an abstract base class ItemCollection was introduced. This class replaced the specific collection type. One could think of the ItemCollection class as the class having the functionality that is present in all three collection libraries.

Three classes derived from ItemCollection serve as the reference-adapter classes for the corresponding collection library. The name reference-adapter was chosen because this design has been greatly influenced by the STL adapter classes and the IBM reference classes. The reference-adapter class acts as an adapter class because it adapts the interface of the corresponding collection class library to the ItemCollection interface; and it also acts as a reference class by delegating all operations to a concrete class and keeping only the reference to the class. For comparison, the STL adapter classes have a concrete class as a member, and the IBM reference classes keep a reference to the concrete class but do not change the interface.

One might think of this design as a variant of the adapter pattern from the GoF book (object adapter) [GHJV95] or the bridge pattern. This might well be a murky area

between the Adapter pattern and the Bridge pattern. The GoF book says that an Adapter is meant to change the interface of an existing object; and a Bridge is meant to separate an interface from its implementation. Because ItemCollection class was created at the same time as the reference-adapter classes, it might well be a separation of interface from its implementation. But at the time of the design the author was not familiar with GoF book and it could serve as a good example of discovering patterns in the existing application.

To make life easier for the developers helper (or adapter) functions have been provided. A helper function creates a new reference-adapter class from a concrete collection class. There are three helper functions, one for each collection class library: fromSTL(), fromIBM(), and fromRW(). The helper function fromIBM() is overloaded, because we could not deduct the type of an object in the IBM collections but we could deduct it in the STL; and in the RW everything is an Object.

The managed pointer class that comes with the IBM collections is used as a reference counted smart pointer making the construction and the destruction of the reference-adapter classes invisible to the users; if the automatic template instantiation is used then there is no need even to know about their existence. An auto_ptr, that is part of the ANSII standard, is not used because a reference to this class could be used and kept in more than one place.

Another benefit of this design is that a collection can now contain not only Item* but pointers to any type that is derived from Item, and if this class is not derived from Item you will get a compile time error.

Example of usage for STL:

```
class Portfolio : public Item{….} ;
Portfolio d;
Criteria PortfolioCriteria( d.name() ) ;

typedef vector< Portfolio* > PortfolioVectorSTL ;

PortfolioVectorSTL  PortfolioCollSTL ;
select(  fromSTL( PortfolioCollSTLVec ), PortfolioCriteria ) ;
count = 1 ;
Portfolio* pPortfolio = 0 ;
 ItemVectorSTL::iterator itSTL ;
for( itSTL = PortfolioCollSTLVec.begin(); itSTL != PortfolioCollSTLVec.end();
itSTL++ )
  {
   pPortfolio = *itSTL ;
   cout << " STL vector " << "Item No. " << count++ << " " << *pPortfolio << endl;
  }
```

Code comments:

- You can use a collection of your choice from your favorite collection class library and if an associative collection is used, this collection will be sorted accordingly;
- There is no need to use cast because in the concrete collection of your choice you can have Portfolio* and not Item*.
- And typing fromSTL(PortfolioCollSTLVec) instead of PortfolioCollSTLVec is a reasonable price to pay for the additional functionality.

This design was implemented for STL, RW, and IBM collections in April 1996.

Removing forced inheritance from the single class Item

Derivation has been replaced with composition. (see GoF book[GHJV95], Orbix TIE approach and external polymorphism pattern[PloP96].)

Usage:

First scenario:

```
typedef  ItemTie<Portfolio> PortfolioTie ;

Portfolio* myPortfolio  ;
……

PortfolioTie* myPortfolioTie = new PortfolioTie( myPortfolio ) ;

aTransaction.add( myPortfolioTie ) ;

…….

delete myPortfolioTie ;
delete myPortfolio ;
```

Second scenario:

The infrastructure creates collection of myPortfolioTie and the client asks myPortfolioTie for myPortfolio.

```
Portfolio* myPortfolio  = myPortfolioTie->item() ;
```

In that case when myPortfolioTie gets deleted it will delete myPortfolio.

- As you could see the class myPortfolio is not derived from the class Item.

This approach was implemented in August 1996.

Performance considerations

There are following performance penalties to consider:

- user should operate on two objects instead of one;
- there is an additional function call in class PortfolioTie that could be inlined and run-time overhead could be reduced to one indirection. The class PortfolioTie is for internal use by the infrastructure library where this overhead is not an issue, and the user will operate on the class Portfolio as before.

The user should create a TIE class and pass a real object of the class T as an argument to the constructor. The infrastructure will operate on objects of class ItemTIE because they are derived from Item and by default will delegate all virtual calls to an object of type T. The user could override the default behavior or provide a specialization. The TIE class could act as an adapter so clients of the component are not forced to implement a certain interface. The TIE class could also act as a pluggable adapter [GHJV95]

Because the infrastructure library does not have compile time dependency on the concrete reference-adapter classes there is a binary compatibility with the infrastructure library; and regardless of the collection library chosen by the users, they do not need to recompile the library. The template reference-adapter classes are small and compile time is not an issue as it is often the case with template libraries. The only overhead is a virtual function call and an additional indirection for accessing the collections. The reference-adapter classes are created for internal use by the infrastructure, and inside the infrastructure this overhead is insignificant. The users are supposed to use concrete classes of their favorite collections - no overhead here.

Concluding Remarks

This approach made the infrastructure library easier and less error-prone to use, and also enabled it to be used with third-party libraries and legacy code. Developers were freed from building bridges and adapters between components and applications. And that brings up the question: should a component look like an octopus with a lot of pluggable adapters?

References

[GHJV95] *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*. Design Patterns: Elements of Reusable Object-Oriented software, Addison-Wesley,

1994.

[PloP96] *Chris Cleeland, Douglas C. Schmidt, and Tim Harrison*. Proceedings of the 3rd Annual Conference on the Pattern Languages of Programs, Urbana-Champaign, Il 3-6 September 1996.