Oleg Shteynbuk

Up '97, Mohonk, NY
an International Workshop On Using Patterns
March 7-9, 1997

Presented at session

Before and After

Patterns that Make Libraries Reusable

The presentation consists of two parts. The first part describes the client data infrastructure library before patterns were used; and the second part describes the infrastructure library after the patterns were applied. All examples are in C++.

1. BEFORE

The infrastructure library has a collection of pointers to the abstract base class (ABC) Item as a part of its API. As a result, to be part of the API the object should be derived from the class Item. In order to use the infrastructure library a client must use the same collection class library. A collection of pointers to Item is used for several reasons: the objects could be rather big, the same object could belong to several different collections, and the duplication of objects is undesirable or unacceptable.
There are two reasons why redesign was needed:

- We have several applications that use different collection class libraries, and these applications have to use the infrastructure library. However, developers want to use their favorite collection class library
- There are cases when it is not feasible or desirable to make all classes inherit from the single class for different reasons: legacy code, third party products, and name clashes.

1.1. USAGE

In the first implementation of the infrastructure library the arguments to the function select(...) were an object of type Criteria and an object of a concrete collection class; we will demonstrate this using the STL vector class. This concrete class will

contain pointers to Item, because the infrastructure returns Item*.

Class Der: public Item {... };

```
Der d;

typedef vector< Item* > ItemVectorSTL ;
ItemVectorSTL  ItemCollSTL ;

select( &ItemCollSTL, Criteria ) ;
```

Then you can iterate through the collection; if you need to use a derived class you should use an explicit cast. The type to cast to is the name that has been passed as part of the criteria; and if your compiler supports dynamic cast it will be safe to cast.

```
Der* pDer ;
Item* pItem ;
int count = 1 ;
ItemVectorSTL::iterator itSTL ;
for( itSTL =ItemCollSTL.begin();  itSTL != ItemCollSTL.end();  itSTL++ )
 {
   pItem = *itSTL ;
  pDer = ( Der* )pItem ;
//    pDer = dynamic_cast< Der* >(pItem) ;  // safe cast
   cout << " STL vector " << "Item No. " << count++ << " " << *pDer << endl;
.....................................................................
  }
```

As you could see you collection is a vector<Item*>; all objects should inherit from Item and user should use cast.

2. AFTER

As was mention before, there were two reasons for this redesign:

- There are several applications that use different collection class libraries, and these applications need to be combined with the infrastructure library;
- There are cases when it is not feasible or desirable for all classes to inherit from the single class for a several reasons - legacy code, third party products, and name clashes.

2.1. The first reason

If you need to incorporate a library into an existing application you could either change the application or adapt the library. Of course the latter is preferable

Common mechanism to operate on any collection type looks very attractive. Then users do not need to know about an existence of the collection class library that is used internally by the infrastructure, and continue to use their favorite collections class library.

In the first implementation a set of pointers to Item was used. But it is not an easy decision to choose one collection class library over the other, not to mention choosing a specific collection type from this library such as a set or a vector, for example. First a set from the Rogue Wave collections was used; then it was changed to a set from the IBM collections that come with C Set++. There is always legacy code with different types of collections and developers prefer their favorite ones. It is not our purpose to get involved in religious wars over the collection class libraries but rather to make the infrastructure available to every application. It is probably worth mentioning that STL containers turn out to be more flexible and easier to wrap around than the other two collection class libraries.

An abstract base class ItemCollection replaced the set as a specific collection type

```
class Item ;
class  ItemCollection{
public:
  virtual ~ ItemCollection() { }

  virtual Boolean isValid() const = 0 ;
  virtual Boolean setToFirst() const = 0 ;
  virtual Boolean setToNext() const = 0 ;
  virtual Item* currentElement() const  = 0 ;
// return value is void because of diff in collections
// bool looks better but STL?.
  virtual void add(  Item* pItem ) = 0 ;
  virtual size_t numberOfElements() const  = 0 ;
  virtual Boolean isEmpty() const = 0 ;
};
```

One could think of ItemCollection class as a common denominator of the three collection libraries.

For this presentation we will keep the interface rather simple and assume that we are not going to use more then one iterator at a time. Then we could make this iterator part of the collection; and all access functions could use this embedded iterator. What we

really need is to populate the collection and iterate over it. Removal of elements from the collection is more complicated because some collections might contain multiple copies of elements, such as bag, for example. And the elements removal will be left for another presentation

Three classes are derived from ItemCollection, each of them serves as a reference-adapter class for the corresponding collection library. The name reference-adapter was chosen because this design has been greatly influenced by the STL adapter classes and IBM reference classes. This class acts as an adapter class: it adapts the interface of the corresponding collection class library to the ItemCollection interface and also acts as the reference class from the IBM collections by delegating all operations to the concrete class and keeping only the reference to this class. On one hand, the STL adapter classes have the concrete class as a member; and on the other hand, the IBM collections keep a reference to the concrete class but do not change the interface and redirect all interface functions to the concrete class.

One might think of this 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: an Adapter is meant to change the interface of an existing object; 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 its author was not familiar with the GoF book and it could serve as a good example of discovering patterns in the existing application.

In case you need more then one traversal pending at the same time, the Iterator pattern could be used and the Factory Method pattern could be used to instantiate the iterator. In our case we will use nearly the same technique for iterators as for collections. The abstract base class ItemIterator can define interface for all iterators and the three derived classes will adapt the interface to the corresponding collections iterator. One difference is that there is no need for these classes to behave like reference classes, adapter behavior is sufficient; and concrete iterators could be encapsulated in the adapter classes for each collection library. Four member functions of ItemCollectable: isValid(), setToFirst(), setToNext() and currentElement(), are really an iterator interface and will be transferred to the abstract base class of the iterator hierarchy. A factory method function will be added to instantiate the corresponding iterator adapter subclass.

Example for STL:

```
template< class Container >
class  ItemCollectionSTL : public  ItemCollection {
public:
  ItemCollectionSTL( Container& cnt )
    : _c( cnt ), _valid( false ) {}
```

```
 Boolean isValid() const { return _valid ; }

 Boolean setToFirst() const
 {

  ( ( ItemCollectionSTL<Container>* ) this )->_it = _c.begin() ;
  return ( ( ItemCollectionSTL<Container>* ) this )->validate() ;
 }

 Boolean setToNext() const
 {
  ( ( ItemCollectionSTL<Container>* ) this )->_it++ ;
  return ( ( ItemCollectionSTL<Container>* ) this )->validate() ;
 }

  Item* currentElement() const { return *_it ; }
 void add(  Item* pItem ) { _c.insert( _c.end(), (Container::value_type)pItem ) ; }
 size_t numberOfElements() const { return _c.size() ; }
 Boolean isEmpty() const { return _c.empty() ; }

protected:
 Boolean validate() { return _valid = ( _it != _c.end() ? true : false ) ; }

 Container& _c ;
 Container::iterator   _it ;
 Boolean _valid ;
};
// with new ANSI standard you do not need the ugly cast; use mutable or const cast
```

If there is ever a need for a different collection class library it could be added in the same way.

2.1.1. USAGE

If instead of a pointer to a concrete class as an argument to function select(...) we pass a pointer to an ItemCollection class then the infrastructure will be able to process any collection that has a concrete adapter-reference class. There are other arguments to select(...) but they are not relevant to the discussion.

```
class Der : public Item { ..... };

   Der d;
```

```
typedef vector< Der* > DerVectorSTL ;
typedef DerCollectionSTLT< DerVectorSTL > RDerVectorSTL ;

DerVectorSTL  DerCollSTL ;
RDerVectorSTL RDerCollSTL( DerCollSTL ) ;

select( &RDerColl, Criteria ) ;
```

and that is how we can use the collection.

```
Der* pDer ;
int count = 1 ;
DerVectorSTL::iterator itSTL ;
for( itSTL =DerCollSTL.begin();  itSTL != DerCollSTL.end();  itSTL++ )
 {
   pDer = *itSTL ;
  cout << " STL vector " << "Der No. " << count++ << " " << *pDer << endl;
.....................................................................
 }
```

Note that the vector now contains Der* instead of Item* and we do not need to use cast anymore; we can pass vector< Item* > or a vector that contains any type that is derived from Item. If Der class is not derived from Item there will be a compile-time error because the function current() contains a return-type conversion to Item*; and in the function add() a dynamic_cast could be used as well. Usage of mutable eliminates the ugly casts of pointer this. As described above, making a separate iterator hierarchy eliminates not only the casts but makes the code more readable.

But we need an additional typedef for the concrete reference-adapter class. And the user should manually construct and initialize an object of a reference-adapter class with the concrete collection class and pass it to the infrastructure. If this object was not constructed on the stack, it should be deleted since its only purpose is to be passed to the infrastructure as a wrapper around the concrete class.

To pass a collection of objects to the infrastructure, add a collection of objects to transaction; it might be much easier to iterate through the collection and add items one by one then to create a reference-adapter class. This is similar to forcing the developers to make a manual bridge to the infrastructure every time they use it. No wonder that a lot of developers have complained that it is difficult to use and makes their life more complicated and error-prone; but they wanted the functionality.

To address this problem helper (or adapter) functions have been provided. There is some analogy with the STL adapter functions. In the STL adapter functions create new function objects from existing ones. In our case helper functions create new reference-adapter classes from concrete collection classes. There are three helper functions: fromSTL(), fromIBM(), and fromRW(); fromIBM(). The function fromIBM() is

overloaded because we could not deduct the type of the object in the IBM collections but we could deduct in the STL; in the RW everything is an Object. There is one function for every collection class library. The STL adapter functions return function objects by value. Actually it is a reference counted smart pointer to an abstract base class, managed pointer class that comes with the IBM collections is used as a reference counted smart pointer. This makes the construction and destruction of reference-adapter classes invisible to users, and if they use automatic template instantiation, then there is no need 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. If we need more than one iterator pending at a time we can implement it using the factory method function that was discussed above. That factory method function should return an auto_ptr because iterator usually is made for local or sequential use and that is what an auto_ptr is for, and there is no overhead of reference counting. And the users should not worry about memory management for collections and iterators.

The following typedef is from the ItemCollection header:

```
typedef IMngPointer<  ItemCollection > PtrItemCollection ;
```

and the helper function for STL looks like:

```
// helper function
template < class Container >
inline PtrItemCollection
fromSTL( Container& anSTL ) {
  return PtrItemCollection( new ItemCollectionSTL< Container > ( anSTL )  ) ;
}
```

the additional argument to IMngPointer() IINIT is not shown here.

```
// helper function

template < class Container >
inline PtrItemCollection
FromSTL( Container& anSTL ) {
  return PtrItemCollection( new ItemCollectionSTL< Container > ( anSTL ), IINIT) ;
}
```

.... usage for STL.....

```
   Der d;
   Criteria derCriteria( d.name() ) ;
```

```
typedef vector< Der* > DerVectorSTL ;

DerVectorSTL  derCollSTL ;
select(  fromSTL( derCollSTLVec ), derCriteria ) ;
count = 1 ;
Der* pDer = 0 ;
 ItemVectorSTL::iterator itSTL ;
for( itSTL = derCollSTLVec.begin(); itSTL != derCollSTLVec.end(); itSTL++ )
 {
  pDer = *itSTL ;
  cout << " STL vector " << "Item No. " << count++ << " " << *pDer << endl;
 }
```

As you can see the usage is nearly as simple as before with the following benefits:

- You could use collection of your choice from your favorite collection class library and if it is an associative collection it will be sorted according to the collection;
- You do not need to use cast because in the concrete collection of your choice there are Der* not Item* as before.
- And it seems that typing fromSTL(derCollSTLVec) instead of derCollSTLVec is a reasonable price to pay for an additional functionality.

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

2.2. THE SECOND REASON

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

```
template< class T >
class ItemTie : public Item
{
public:
  ItemTie()
     : _item( new T ), _myItem( true ) {}
  ItemTie( T* anItem )
     : _item( anItem ), _myItem( false ) {}

  ItemTie( const Oid& oid )
     : MSDNItem( oid ), _item( new T ), _myItem( true ) {}
  ItemTie( const Oid& oid, T* anItem )
     : Item( oid ), _item( anItem ), _myItem( false ) {}

  ~ItemTie() { if( _myItem ) delete _item ; }
…….
```

```
  T* item() { return _item ; }

private:
……..


  T* _item ;


bool _myItem ;
} ;
```

Some functions have a default implementation, default is a delegation to class T; and other functions do not have a default implementation, and the user should supply it as a specialization if your compiler could handle it.

The user should create a TIE class and pass a real object of class T as an argument to the constructor. The infrastructure will operate on objects of class ItemTIE.

2.2.1. USAGE

First scenario:

```
 typedef  ItemTie<Portfolio> PortfolioTie ;

Portfolio* myPortfolio  ;
……

PortfolioTie* myPortfolioTie = new PortfolioTie( myPortfolio ) ;

aTransaction.add( myPortfolioTie ) ;

…….

delete myPortfolioTie ;
delete myPortfolio ;
```

Second scenario:

The infrastracture create collection of myPortfolioTie and client ask myPortfolioTie for myPortfolio.

Portfolio* myPortfolio = myPortfolioTie->item() ;

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

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

But there is a price to pay:

- 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. Class PortfolioTie is for internal usage by the infrastructure library where this overhead is not significant, and user will operate on a class Portfolio as before.

This approach was implemented in August 1996.

3. CONCUSION

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. Template reference-adapter classes are small and compile time is not an issue here as it is often the case with template libraries. The overhead of a virtual function call and an additional indirection for accessing the collections should not be an issue, as 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.

There are alternatives, for example, the infrastructure could create a new collection and return it to the client; a vector or a set could be good candidates for the default collection. Then the client should copy the collection that gets populated by the infrastructure into the collection of his/her choice. There are two drawbacks of this approach. First, there is the overhead of copying the elements; time of the element insertion depends on the size and type of the collection, and the memory management scheme. However, it might be not an issue at all, taking into account that overhead of using reference-adapter classes is eliminated and it is a collection of pointers, but the users have to copy the collection and cast the elements. Second, the users have to deal with two different collection libraries with interfaces as different as, say, the STL collections and the RW collections; and that could be a source for additional bugs. Also it is more difficult to couple the concrete collection with callbacks to update the collection when a new object is created or an old one is deleted.

The approach, described here, makes the integration of the infrastructure library with the applications easier and less error-prone. Also the code is clearer and less error-prone because it frees the developers from building the bridges and adapters between the library and the applications.

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.