http://www.drdobbs.com/cpp/c-view-objects/196513737?pgno=1

C++ View Objects

By Gabhan Berry, November 28, 2006

Creating a view of data is a concept used extensively by database developers but is rarely used in other software domains. However, the view concept is equally valid in all programming domains where data is central to the nature of the problem being solved. When data needs to be presented in different ways, the natural solution is to create multiple copies of the data with each copy arranged in a specific manner. This is not always the most efficient solution and can lead to overcomplicated or duplicated code further down the line.

This is the problem that view objects solve; they provide an efficient and concise means of presenting data in multiple ways without inefficiency. Let's consider a simple example. Say that we have a vector of sales records where each sales record is an instance of a record class that we have written.
`vector<record>` records;

Let's say that in our program we need to be able to present the records in ascending order of sales value. The natural response is to sort the vector object. Something similar to the following:

```
sort(records.begin(), records.end(), someFunctor);
```

However, this is changing the data. That is, the records in the vector object are being rearranged to be in a different order. In this simple case, this is fine. But what happens if the program now needs to present the data in ascending order of sales date as well? Now the program needs two presentations of the data; one sorted by sales value and one sorted by sales date.

Without views, there are two solutions to this problem. You could make a copy of the records vector (so that there are two records vectors) and apply a different sorting to

each one. Alternatively you could sort the vector just-in-time, that is, apply the sort-by-date algorithm just before you need to access the data by date and then apply the sort-by-value algorithm as and when you need to access it by value.

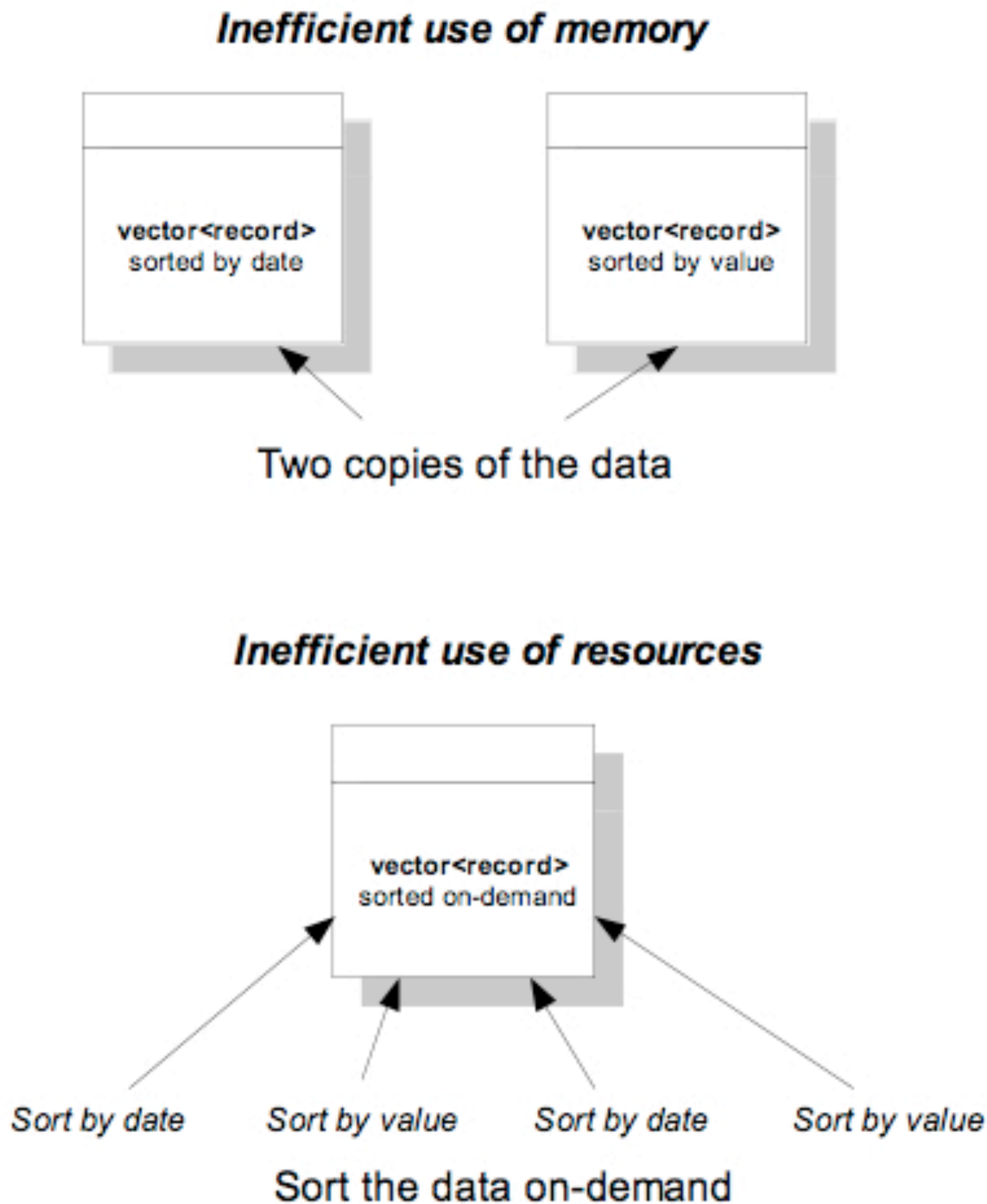

Figure 1: Efficiency and Scalability Issues With Record Copying and on-Demand Sorting.

Both of these solutions have efficiency and scalability issues (see Figure 1). As time goes by and more and more representations of the data are required, so the efficiency further decreases.

The view-based solution to this problem is this: create one view of the record vector which is sorted by date and another view that is sorted by value. (See Figure 2.)

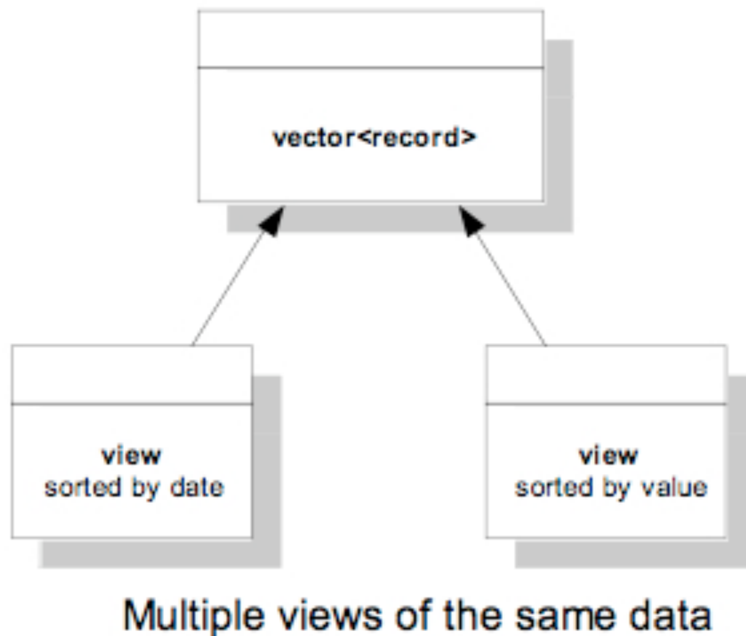

Figure 2: View-Based Solution to the Problem of Figure 1.

There is only one copy of the original record vector in memory and it is never changed by the views. As time goes by and new representations of the vector are required, new views are created. This scales better than the previous two solutions because a view needs to be built just once, avoiding repeatedly rearranging the data, and does not create extra copies of data thus reducing memory usage.

In this article I will show how to develop a view class that can be used to implement the view concept in any C++ program.

Designing the View Class

In order to apply the view concept we need a view class. Before we think about the code it is worth stating the fundamental behaviour that a view class should exhibit.

- The view contains links to the underlying data, not copies of it

- Only the data that qualifies for the view can be accessed via the view
- The view can reference multiple underlying data containers

The first behaviour is crucial. It is this feature that provides view objects with their efficiency. The efficiency comes from the fact that a link to a piece of data is nearly always smaller and never larger than the data itself. So storing links uses no more and normally considerably less memory than storing copies of the data. The second behaviour means that the view does not grant any access to the data in the underlying container that is not contained within the view. This is very important and means that, for instance, a view showing the top five sales records by sale value will grant access only to those five records and not to any other part of the data. The third behaviour allows views to virtually merge data from multiple sources into one view. For instance, we should be able to create a view that represents data from two vector objects or from two map objects. This is a very powerful behaviour and can be used to construct sophisticated views.

Now we are ready to start coding. We'll write the view class by coding each one of the three behaviours one-by-one.

Note that the techniques applied and the code written in the article are both focused on using the STL.

Coding the View Class

#1: The view contains links to the underlying data, not copies of it

We want our view class to work with any type of data; we want to be able to use the same class to build a view of a vector of integers as we do to build a view of a vector of custom objects. Furthermore, we need the view to be container agnostic. What I mean by this is that there should be no restriction on the underlying data source. So we want the same view class to work for maps and vectors seamlessly and without change.

This means that we need to think a little more carefully about how we represent the underlying data because the underlying data can be of any type and can be stored in any type of container.

So let's think in general terms. We have a container of type C holding objects of type T and we want to create a view of type V in such a way that there are no extra copies of type C or T.

The first step is to consider what class of object the view will actually contain. If it is not allowed to contain copies of the underlying data then it cannot contain elements of class T. An obvious solution is instead of storing objects of type T in the view, we store objects of type P(T) where P(T) represents some kind of proxy of T.

If we stay within the conceptual realms of the Standard Library, an ideal P(T) is the *iterator*. An iterator is an indirection object or a proxy object used with all the standard containers. So if we make our view class store iterators then we will be able to work with all the standard containers seamlessly.

Therefore we have, our view V contains n elements of type P(T) where n is number of elements in the view and P(T) is an iterator to an object of type T.

Before we can express this in code, there is one more issue we need to deal with. A view is container agnostic but an iterator is not. For example, *vector<int>::iterator* is not of the same type as *set<int>::iterator*. So, although the view itself is not conceptually tied to the type of underlying container it does need to know the type so that it can determine what type the iterators P(T) are. We can let the view know all this by making the view a parameterised type and specifying the type of underlying container in a type parameter.

These intricate details turn out to be very easy to describe in code.

```
template<typename _CtnType>
class view{
protected:
	typedef _CtnType::const_iterator element_type;
	vector<element_type>_elements;
public:
	element_type operator[]( size_type x ){
		return _elements[x];
```

```
    }
}
typedef typename vector< element_type >::iterator iterator;
```

Notice that I am using a *vector* object to store the *iterator*s. This is just because vectors give the most flexibility and speed. This implementation detail will be hidden from the user of the view.

This definition gives us the first required behaviour, that the view contains links to the underlying data not copies of it. However, if we left the view class like this we would not be making the usage of the class as nice as it could be.

We want to encapsulate the intricacies of the view from the code using the view but, at present, the fact that we are storing *iterator* objects is not hidden and means that code that accesses the view will constantly have to dereference the *iterator* object to get at the real data. This would give rise to the following style of code.

```
vector<wstring>ctn;
view<vector<wstring>>theView;
wstring v = *(theView[0]); //need to explicitly dereference the iterator
```

It is easy for us to dereference the iterator internally within the view meaning that the client code does not have to. The client code only needs to deal with the data which makes the view much nicer to use.

However there are other considerations, which I haven't mentioned yet, which affect what type P(T) really should be. These considerations have to do with making the view as easy as possible for client code to use, both in the construction of the view and in accessing the view. Therefore, instead of P(T) being an *iterator* object let's define a separate proxy class called *view_element* which we will use instead.

The *view_element* class is just a lightweight wrapper of an *iterator* but it gives us the added capabilities of defining custom constructors and operators that allow us to finely control how our view class is used by client code and the encapsulations we make.

The *view_element* class needs to know the type of underlying container so that it knows the type of iterator that it stores. We'll pass this information in as a type parameter like before.

This gives the following definitions for the *view_element* and *view* classes.

```
template<typename _CtnType>
class view_element{
protected:
	typename _CtnType::const_iterator	_itr;
public:
	typedef typename _CtnType::value_type value_type;
	virtual const value_type& value() const{
		return *_itr;
	}
};

template<typename _CtnType>
class view{
protected:
	typedef view_element<_CtnType> element_type;
	typedef typename _CtnType::value_type value_type;
	vector<element_type> _elements;
public:
	value_type operator[]( size_type x ){
		return _elements[x].value();
	}
}
```

This is all the code we need to implement the first behaviour, so let's continue onto the second.

#2: Only data that qualifies for the view can be accessed via the view

The purpose of this behaviour is to make sure that the *view* object encapsulates only the subset of the data included in the view. Client code should not be able to use the

view to access data outside of the view. It turns out that the code we wrote to implement the first behaviour also provides us with the second behaviour.

This is because the *view* class simply stores a *vector* of proxy objects (called *view_elements*) where each proxy represents only one element in the underlying data. Client code, however, receives the underlying data via the *view_element* proxy. The raw iterator to the underlying data is hidden within the *view_element* proxy and is not returned to the client. So the client has no way back to the underlying container and can only receive data via the view.

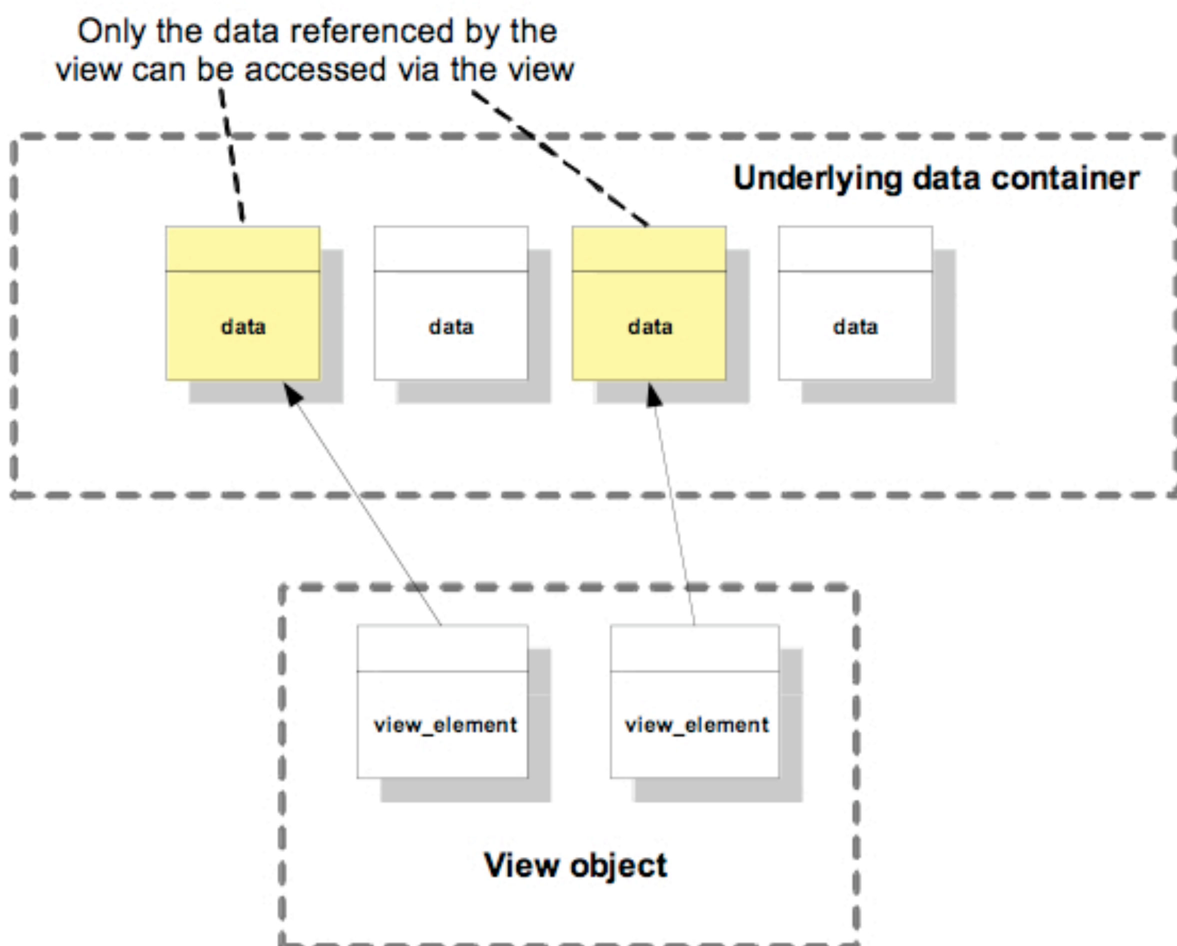

Figure 3: Limiting the View's Data Access.

This would not have been the case if we had decided to store *iterator* objects directly in the view and not our *view_element* proxy. If the *view* object returned raw iterators to client code then client code could use these iterators to traverse to data outside of the view thus contradicting the purpose of the view in the first place.

An additional benefit is that any code can iterate over the contents of any view object without having to understand how the view was built or have a direct reference to or understanding of the underlying containers. This decoupling allows you to hide the internal details of the data storage and logic but still grant finely controlled access to the data.

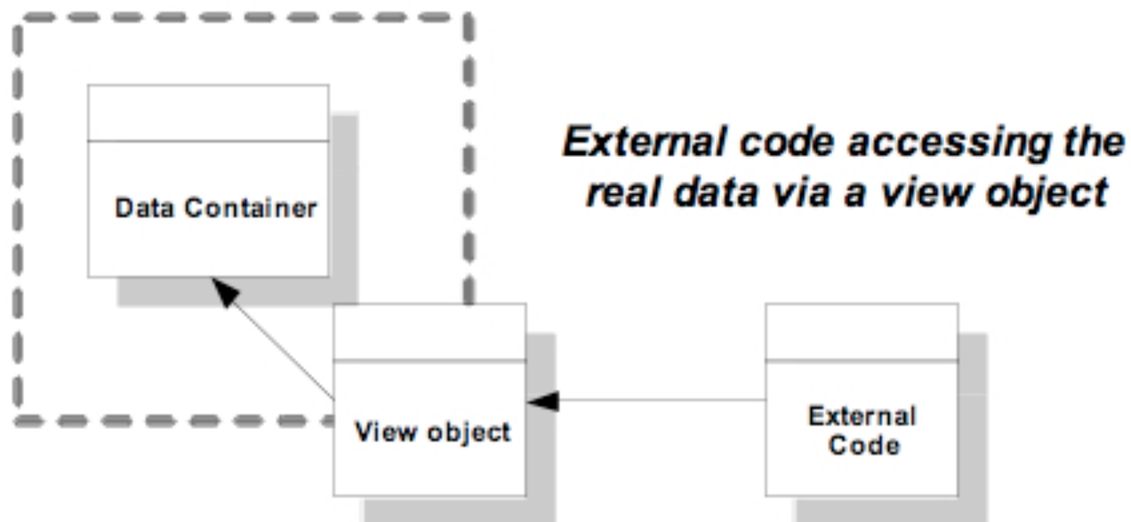

Figure 4: Ease of External Code Access.

#3 The view can reference multiple underlying data containers

Views not need be limited to referring to only one container. Data from multiple containers can be combined to create a coherent and navigable view object that looks and acts no differently than any other view object. The consumer of the view does not know, or need to know, that the view object refers to multiple underlying containers.

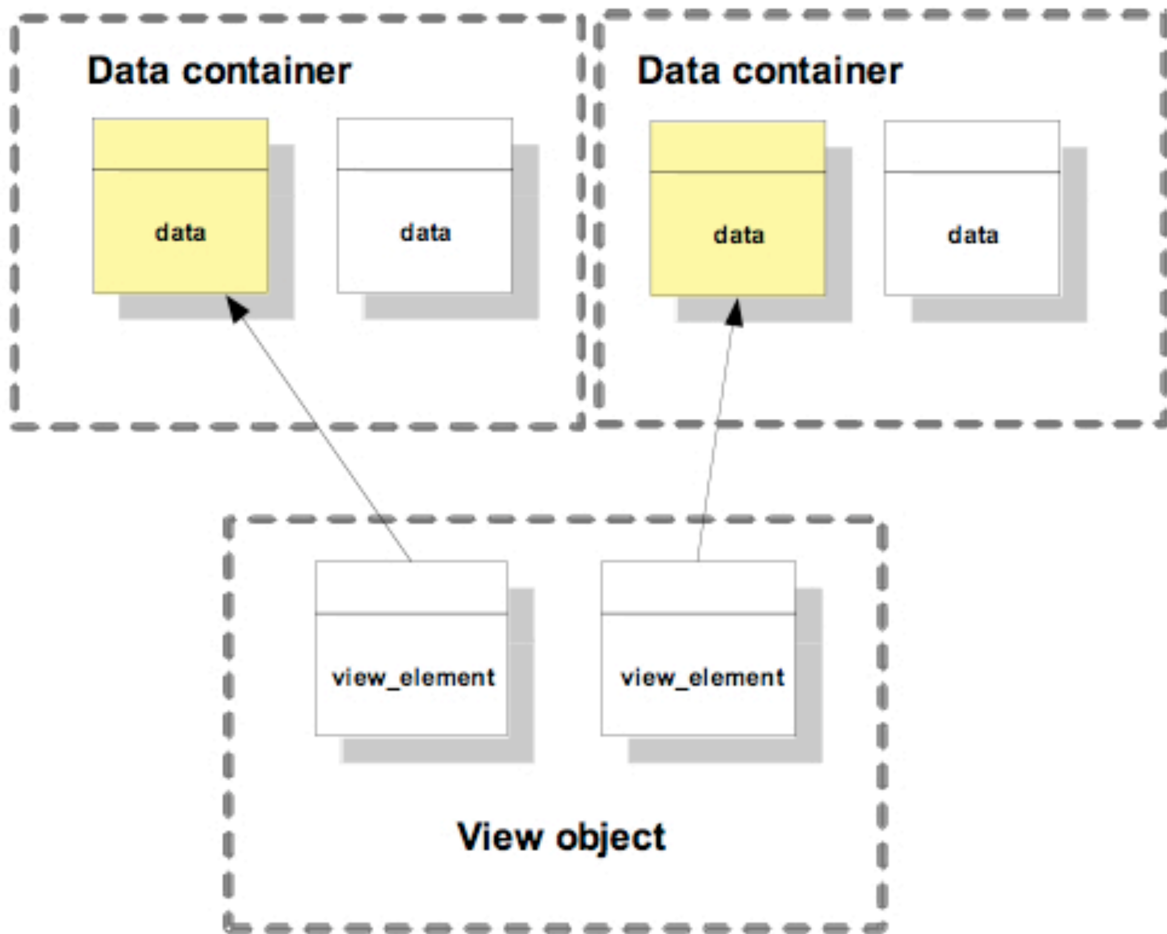

Figure 5: a View Referencing Multiple Containers.

The view stores proxy objects each of which represents a particular piece of data in an underlying container. We have implemented this proxy using iterator objects. In order to access the real data via the iterator we simply have to dereference the iterator.

So our current *view* and *view_element* classes can already create multi-container views. All that is required is to add iterators that point to different containers into the same view object. It's as simple as that.

```
typedef vector<wstring> wstringVec;
wstringVec ctn1;
wstringVec ctn2;
view<wstringVec> wstringVecView;
```

```
wstringVecView.add(ctn1.begin()+1);  // Add the second element from ctn1
wstringVecView.add(ctn2.begin()+1);  // Add the second element from ctn2
```

However, there is one caveat. Within a view object, all proxy objects have to be of the same type, P(T) where T is the type of the underlying containers. Therefore, all underlying containers have to be of type T. So it is not possible, at least in this implementation, for a view<T> to point to containers that are not of type T. This is because a view<T> is composed of view_element objects of type view_element<T>. So if we have two containers of type T0 and T1 and a view_element pointing to each then we have view_element<T0> and and view_element<T1>. If these view_elements are to be placed into a view object of type view<T> then they must be of type view_element<T>. Therefore, it follows that view_element<T0> = view_element<T1> = view_element<T> and this is only true if T0 = T1 = T.

Summary

It is a simple idea requiring simple code but offering huge benefits. View objects enable a programmer to build as many presentations of data as required without having to create and maintain multiple copies.

When writing a view class, remember the underlying behaviours of view objects:

- The view contains links to the underlying data, not copies of it
- Only the data that qualifies for the view can be accessed via the view
- The view can reference multiple underlying data containers

This article has described just one possible implementation of view objects, but there are others. A complete view class, using the techniques described in this article, is available for download and can be used in your own code. Alternatively, you can create your own. But whichever you use, views are a powerful programming tool that can be applied to solving a variety of problems from the large scale to the small scale. In my experience I've found that wherever data is being manipulated or presented, view objects have something to offer.