The Humble Dialog Box

Michael Feathers
Object Mentor, Inc.
mfeathers@objectmentor.com

Let's take a look at our friend the dialog box. He's a little workhorse. Whenever we have something clever to do in our application, we create a class, bind it to a dialog box and send it off to talk to the user. Tool vendors have made it easy to create dialog boxes. Nearly every IDE on the market has a GUI builder. You can drag and drop all sorts of components on dialogs and drill down to wire them up. Life is supposed to be simple, and with all of that IDE support you should be able to hammer out ten or twelve dialog boxes a day!

No, no, I haven't been drinking market literature or anything else intoxicating. I was just joking, don't put down the article. The truth is I've been burned by dialog generation tools many times. Sure, I appreciate what their designers are trying to do for us. Nobody likes tedious coding, but most things that make dialog boxes easier to create make them harder to work with. Code generators give you all of the wiring. It is easy to just override an event from a component and drop your interaction logic right there in the dialog box class. While this works fine in simple cases, you end up with a series of dilemmas when the inevitable happens. Here is small list of "inevitables":

- You have to add more logic to the dialog interaction; you introduce a bug because you couldn't get your dialog class into a test harness.
- You start working on a similar dialog, and you sense that it is doing some of the same work, but it is hard to see just how similar it is because the code spends as much time working with low level component API as it spends making decisions.
- You start to add acceptance tests for your application, but wish that you didn't have to use a GUI scripting tool. If you had some separation between the UI and the interaction logic, you could just work with it directly in a test harness.

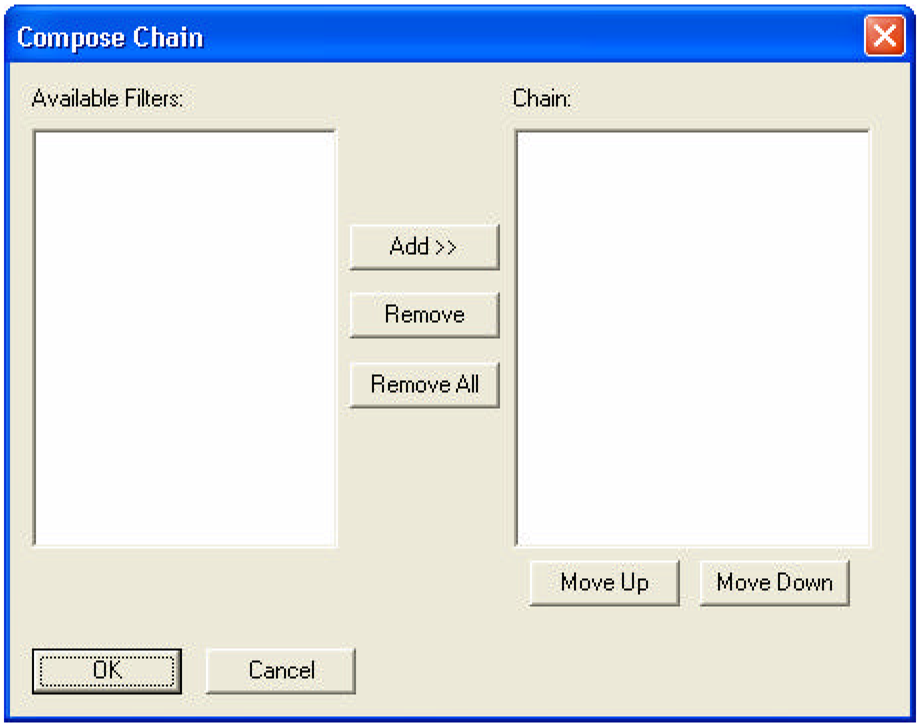
The best way to get around these issues is to resist the temptation to put code in the dialog box class. The easiest way to do that is to create another class first and create the dialog class last. When you do that, you end up with two classes: a smart tested class and a humble dialog class.

Chain Composer

For me, the hardest part of writing is dreaming up examples. The specific solutions I've developed with clients are too fresh in my mind and not shareable, but I can describe humble dialogs with a little example from a program that I've written in a couple different languages over the years.

Music is one of my hobbies and you can definitely do some interesting things at the intersection of music and programming. Often the most straightforward way of working with sound in a programming language is to use a MIDI library. MIDI stands for Musical Instrument Device Interface. It is a standard that allows musical instruments to talk to each other. Among other things, the standard defines events that you can use to turn a musical note on or off. To play a MIDI device you send it the right events at the right time. You can also receive events from MIDI devices and store them for later use. You can also transform them and send them out to a MIDI device. When you are able to treat music as data you can do some very interesting things. For instance, you can write a program which timestamps events as it receives them and then sends them out to the same MIDI device after a fixed interval. If the interval is small, you have a reverberation effect. If it's a little longer, it's an echo. Longer still and you have a delay.

In my program, I have a collection of effects like the ones I just described. The metaphor of the program is that an effect is a chain of filters which accepts events one at a time at one end, and produces zero or more events at the other end. To tie filters together in a chain, we need a dialog box that shows all of the available filters and lets users build up an ordered list that represents the chain.



First Steps

To start, we need a class. If I were making the dialog class first, I'd probably call it ComposeChainDialog. Eventually we will need that name when we get around to making our humble dialog, so what I'll do now is create a smarter class named ChainComposer. Let's approach it test-first.

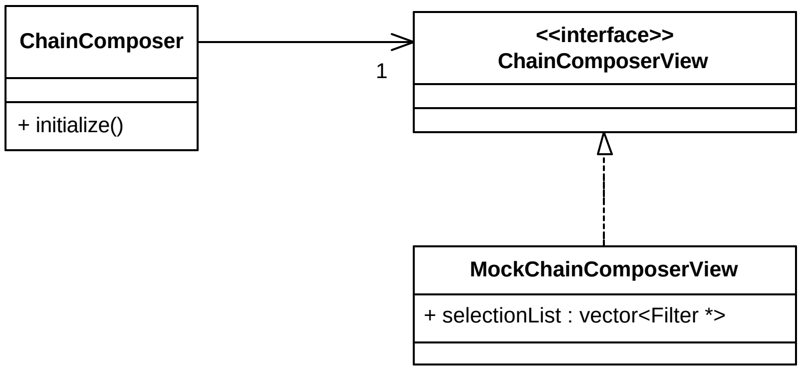
What we'd like to do is handle the startup for the dialog. In other words, what data should we see when it pops up? By default, we'd like to see a list of the available filters in the left hand list box. How do we get them there? Let's write a test for what we expect.

Here is the first test for ChainComposer:

```
TEST(initialize,ChainComposer)
{
      MockChainComposerView   view;
      ChainComposer           composer(view);

      CHECK_LONGS_EQUAL(0, view.selectionList.size());

      composer.initialize();

      CHECK_LONGS_EQUAL(1, view.selectionList.size());
}
```

In the example, I'm using a unit testing framework named CppUnitLite. The framework lets you create small independent tests without manually creating classes and test suites. The test above is a test for the initialize method of ChainComposer. The way it is set up, we are expecting chain composers to accept a reference to a view class. When composers are sent the message "initialize", they give the view a list of filters to select from. In this case, the composer will have only one filter type to provide for selection. The composer should take that filter, and send it along to a view class. We can then ask the view class whether it received it.

To make this test pass, we have to build up a little object structure:

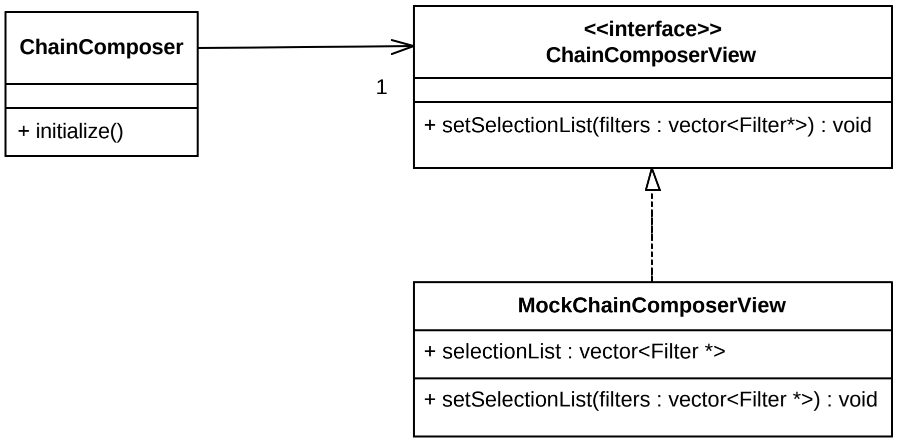


The ChainComposerView class will be very useful. It will contain only pure virtual functions and it will define the "language" that the composer will eventually use to talk to the dialog. On the other hand, the mock view class is just a stand-in to use during tests. It lets us build up the composer without having to work with the GUI classes right away.

Once we have that class structure in place, we can use it to make the test pass. More specifically, if you call initialize on the composer the number of filters that the view knows about should become 1. For that to happen, the composer has to pass a vector of filters over. To handle this, we can create a method named setSelectionList and declare it on ChainComposerView. To make the test pass, we add some code to the composer's initialize method, and then we'll override setSelectionList on the mock view and make it save the vector.

```
void ChainComposer::initialize()
{
      filters.push_back(new ReverbFilter);
      view.setSelectionList(filters);
}

void MockChainComposerView::setSelectionList(
            const std::vector<Filter *>& filters)
{
      selectionFilters = filters;
}
```

Flipping back to UML, this is what our structure looks like now:


ChainComposer
+ initialize()
1
<<interface>>
ChainComposerView
+ setSelectionList(filters : vector<Filter*>) : void
MockChainComposerView
+ selectionList : vector<Filter *>
+ setSelectionList(filters : vector<Filter *>) : void

At this point, we can round out the behavior with a few more tests. We can initialize the view with several filters, and verify that we can get their names back. We can also deal with the question of where the initial set of filters will come from. Will they be parameters to the ChainComposer class, or will the composer talk to some other object that will provide them? What happens when the user hits the "okay" button? To handle each case, we write another test against the smart object and let it lead us to a simple solution, but right now I want to deal with another test case which fleshes out more of the humble dialog idea.

When we make our dialog, it will have a button to add a selected filter to the chain. How do we write a test for that functionality? Here is one possibility.

```
TEST(selectFilter,ChainComposer)
{
      MockChainComposerView   view;
      ChainComposer           composer(view);

      composer.initialize();
      composer.add(0);

      CHECK_LONGS_EQUAL(1, view.composedFilter.size());
}
```

Here we're adding an "add" method to the chain composer. We choose the name "add" because we will have an "add" button on the dialog. We want a one to one mapping from commands, or "gestures" that require logic, to operations on the composer class. If we have that, then the code we have to write in the dialog will be dead simple. We'll just be delegating calls for events. When you do this a few times, you notice that you look at the method names in your class browser more often than the dialog resource. The method names are a more accurate vision of what the dialog actually does.

The "add" method takes the index of the currently selected filter as an argument. The composer can use it to pull the associated filter from its filters vector, add it to its chain vector, and set the chain of the view. And that is how we get the test to pass:

```
void ChainComposer::add(int filterIndex)
{
      chain.push_back(filters [filterIndex]);
      view.setChainList(chain);

}
```

C++ being what it is, I had to add setChainList as a pure virtual function on ChainComposerView. Then, I added it to MockChainComposerView. Its implementation there parallels the one for setSelectionList.

Tying It All Together

Now that we have a few methods implemented, let's look back at what we've done. The tests that we've written aren't typical. Typically, we'd write a test which sends messages to an object and then checks return values, or queries the object to find out what state it's in after our actions. Our tests have been a little different. We've sent messages to a smart object and then asked questions of some other object it's connected to; in this case it is a fake, a mock object. The reason we've done that is so that we can drive the creation of an *internal protocol* between the smart object and its view. The methods setSelectionList and setChainList are part of that protocol.

As we build up the functionality of the smart object, there are a couple of things to consider. Do all actions against the dialog have to have operations on the smart object? Actually, they don't. Let's look at an example. When we started to write a test for the "add" operation, we could have done a bit of analysis and discovered that the user had to select a listbox item before pressing "add." Instead of starting with "add", we could have started by creating a test for a selectItem method on the smart object. One we had that in place we would have been able to write a test for an "add" method that worked against the current selection. It wouldn't have required any arguments. Would that have been better? In my opinion: no. Querying for the selection index in the view is easy. Once you have it, you can pass it to the add method. It really doesn't require any computation. Typically I don't add an action to the smart class for things that do not require computation in the view. The goal is to get all of the logic in the smart object and keep it out of the view.

Once we have all of the functionality we want in the smart object, it is time to work with the dialog class. To do this, we implement the view interface on a dialog class. If we decided to stop short, this is what the dialog class would look like for what we've implemented so far. Notice that each action method is very small, just a simple delegation. Every method that receives data on the view is as close to a pure "set" method as possible.

```
void ChainComposerDialog::OnAdd()
{
      composer->add(getListBox(IDC_SELECTIONLIST)->GetCurSel());
}

void ChainComposerDialog::setSelectionList(
      const std::vector<Filter *>& filters)
{
      setList(IDC_SELECTIONLIST, filters);
}

void ChainComposerDialog::setComposedList(
      const std::vector<Filter *>& filters)
{
      setList(IDC_CHAINLIST, filters);
}

CListBox *ChainComposerDialog::getListBox(int id)
{
      return (CListBox *)(GetDlgItem(id));
}

void ChainComposerDialog::setList(
      int id,
      const std::vector<Filter *>& filters)
{
      CListBox *list = getListBox(id);

      list->ResetContent();

      for (std::vector<Filter *>::const_iterator
                  it = filters.begin();
                  it != filters.end();
                  ++it)
            list->AddString((*it)->getName().c_str());

}
```

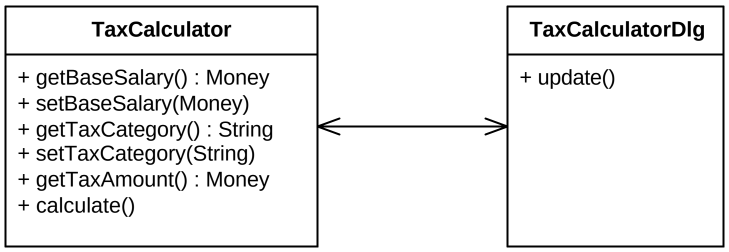
To make the composer available to the dialog class, we create an instance and bind it:

```
ChainComposerDialog::ChainComposerDialog(CWnd* pParent)
      : CDialog(ChainComposerDialog::IDD, pParent)
{
      composer = new ComposeFilter(*this);
}

ChainComposerDialog::~ChainComposerDialog()
{
      delete composer;
}
```

```
BOOL ChainComposerDialog::OnInitDialog()
{
      CDialog::OnInitDialog();

      composer->initialize();
      return TRUE;
}
```

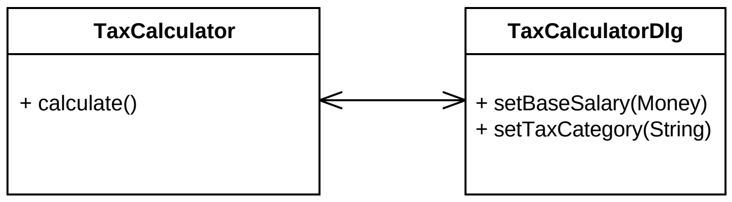
Basically, that is all there is to making humble dialogs. It seems like a bit of work, but to figure out whether it was worth it, let's take a look at where we are. We have all the functionality of the dialog separated from the GUI classes. The only untested code is a very thin wedge at the interface that consists of simple delegation and value setting. Once the code is in the smart object, you can refactor it any way that you need to. Often when refactoring, getting the code under test is the hard part. When you make humble dialogs, all of your logic is under test by default and it is not uncommon to discover duplication across several smart objects, particularly in the area of verification. Mike Hill has pointed out that when you have these smart objects, sometimes you can nest them to build up interactions in more complicated dialogs.

If you've seen many of the usual user interface design patterns: Model/View/Controller, Model/View/Presentation, or Presentation/Access/Control (Swing and MFC use collapsed variations of the first), you may be thrown by the fact that the smart object pushes data onto the view class. This is exactly the opposite of the tendency in most UI frameworks. Why are we doing that? To me, it feels more natural, but I can also provide some concrete justification. Let's take a look at another example.



Here we have a more conventional way of structuring a UI. We have a "model" class and a "view" class. Whenever a change is made on the model class, it calls "update" on the view. The view turns around and gets all of the data that it needs to display. While this works nicely in the beginning, it is often tempting to start adding computation to the view class. Why? Well, because it is so convenient. The view class already has a reference to the model along with a whole plank of getters and setters. With all of that capability, you can make the tax system arbitrarily complex without the rest of the system knowing about it. You'd have to hire a lawyer to find the logic. In other words, your costs and anxiety will grow without bounds.

Let's contrast this with the humble dialog.



The logic of the system remains encapsulated. It is pretty hard to add logic to the dialog class. You actually have to go out of your way to do it. We have more control because we just tell the dialog the things it needs to know when it needs to know them.

The Humble Dialog

1,2,3..

1. Create a class for the smart object, and an interface class for the view. Pass the view to the smart object
2. Develop commands against the smart object, test first. Write your tests against a mock view.
3. Create your dialog class and implement the view interface on it. Gestures on the dialog should delegate to commands on the smart object. Calls from the smart object to the dialog should resolve to simple setter methods.

When you follow these steps, you end up with tested code and a great interface for driving acceptance tests programmatically.