Presenter First: Organizing Complex GUI Applications for Test-Driven Development

Micah Alles
David Crosby
Carl Erickson
Atomic Object

Brian Harleton
Michael Marsiglia
X-Rite

Greg Pattison
Curt Stienstra
Burke Porter Machinery

Abstract

Presenter First (PF) is a technique for organizing source code and development activities to produce fully tested GUI applications from customer stories using test-driven development. The three elements of Presenter First are a strategy for how applications are developed and tested, a variant on the Model View Presenter (MVP) design pattern, and a particular means of composing MVP triads. Presenter tests provide an economical alternative to automated GUI system tests. We have used Presenter First on projects ranging in size from several to a hundred MVP triads. This paper describes MVP creation, composition, scaling, and the tools and process we use. An example C# application illustrates the application of the Presenter First technique.

1. The Problem

Test-driven development from user stories lets us build high quality code and deliver working software early and often. In our experience, following this approach with complex GUI applications is hard, as testing the view through an application's interface is difficult and expensive. The cost of GUI testing and the pain we felt in view changes drove us to find a better way. Presenter First is what we call the better way.

Presenter First is a combination of process and pattern. The process aspect of Presenter First is the most important. Learning how to build applications Presenter First lets you do test-driven development for large GUI applications following user-prioritized stories.

We use a variant of the Model View Presenter (MVP) design pattern. The three components of MVP are:

Model The data and logic that serves the needs of a presenter for some aspect of the business.

View A portion of the interface of the application–what the user sees and interacts with.

Presenter Behavior that corresponds directly to customer stories. Application wiring.

Figure 1 shows the variant of MVP we use to build applications following Presenter First. The model and view are isolated from each other; the presenter is the center of the universe.

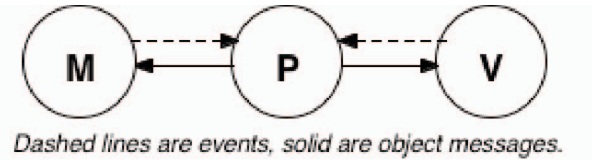

Figure 1. MVP variant used in PF.

We have used Presenter First in at least six applications, three languages, and diverse business domains. It gives us test coverage corresponding directly to user stories. It isolates our code and tests from high rates of change in the view. It's fairly easy to learn. Once learned, it's simple to apply. It scales up. And it eliminates the pain of GUI testing.

2. Our Story

The evolution of our Presenter First approach began with a software application written in C# that interfaced with a device for measuring and managing the color of sheets that are printed by large high speed presses. The application consists of over 30 separate user interfaces and a SQL Server database.

Our team was already applying agile design practices. Though we had no formal standards in this regard, we strove to separate the view from the business logic. Our success at this was put to the test when one year into a two year project, our company hired a graphic design firm to redesign the user interfaces for the entire application. We soon discovered that our decoupling efforts were insufficient for such a major change. The view portion of the code base understandably had to be redone. However, we also discovered that the control logic, such as 'when

Proceedings of AGILE 2006 Conference (AGILE'06)
0-7695-2562-8/06 $20.00 © 2006 IEEE

list is empty, disable button', also had to be nearly completely redone. In fact, even portions of our model and business logic had to be modified to account for redesigned user interfaces.

This experience convinced us that we needed a more rigorous method of separating the view from the business logic. We introduced a presenter/controller to handle the interactions between view and model. There are many versions and implementations of the MVC or MVP pattern. Our prior experience with user interface overhauls convinced us that we wanted the strictest separation possible between these concerns.

Another requirement of this application was that it had to support large customers who wanted to brand the application with their own custom graphics, style, and logos. Our goal was to isolate the changes to the view portion of the code to support different customers. The presenter/controller logic should remain exactly the same. From this, it became clear that if the view supported an interface for the presenter to work with, then we could easily swap in and out the appropriate view depending on the version of the application. Not only did this improve the portability of the view code, it greatly improved the testability of the presenter.

Seeing this improvement of testability and portability, we made interfaces for the models such that the presenter no longer worked with any concrete classes. Though we improved the decoupling of our application and the testability of our presenters, we still needed an effective way to test the most common part of a user interface: the user's interaction.

As testing views is difficult and tedious, we decided the view should be as thin and simple as possible. Ideally, the view should simply act as a pass-through of user events and as getters and setters of display information. To handle this, all user events such as button clicks and item selections were represented by events on the view interface. With the view and model as members of the presenter, the presenter can then attach itself to these view events. View events could easily be mocked and fired in the presenter unit tests, simulating customer interaction. Likewise, model events such as a measurement from the scanning device or a project loaded from the data source could be captured and tested in a similar fashion.

This led us to the realization that the presenter was only the interpreter of events and no longer had a public API. An interesting side effect of this was that we no longer even had to hold a reference to the presenter. We simply constructed the presenter with the appropriate view and model, which glued them together for the lifetime of the application. Further interaction with the presenter was no longer needed. Figure 2 demonstrates the relationships between classes in Presenter First.

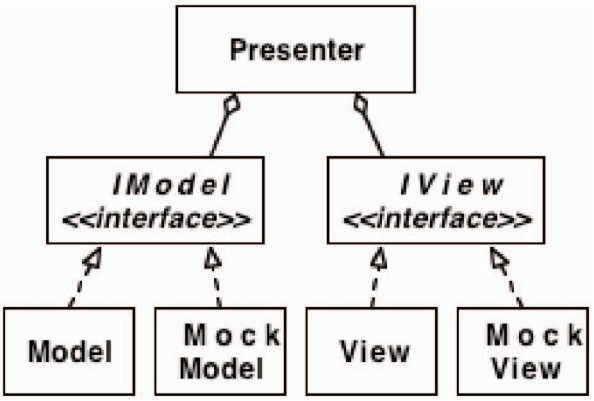

Figure 2. Class diagram of Presenter First MVP classes, tests classes, and interfaces.

After developing several tests for the presenter, it became clear that they closely resembled the original user stories. For example, a test named something like `test_UserClicksSave_Measurement` sounds much like the title of a story. The strong relationship between stories and tests led us into the technique we now call Presenter First.

3. Building an Application Presenter First

Presenter First is dry walling, not quantum physics. Once you learn it, you can do it repeatably, regularly, and (with good tools) quickly. Development proceeds according to the pseudo-code in Figure 3.

The outer loop of the Presenter First process requires frequent refactoring of the model and view interfaces. These refactorings are not expensive, however, since at this point there are only mock implementations of these interfaces. After the presenter is complete, specifications for the view and model are both complete and minimal and ready to be implemented.

The view portion of the code could be handed off to someone with usability expertise who does not need much programming knowledge. When the user sees the application interface, they often request changes such as 'move this button over here', or 'put these controls in a group box'. With Presenter First these changes are isolated to the view portion of the code and do not affect the rest of the application. Changes to the view become easy to make and present very low risk to the application.

Testing the view may either be automated, if the tools, budget, and risk dictate, or left to a final manual system test phase. The view is very thin, with methods that consist of little more than firing properly typed events to be handled by the presenter. Assuming that the underlying GUI widgets work, the primary use of unit tests on the view are to assert that the widgets

```
Create a stub presenter class that takes a model
interface and a view interface via its constructor

Create mock test objects that satisfy the model and view
interfaces

For all user stories in customer priority order
  Analyze story for impact on view
  Add support to view interface for the story
  Analyze story for impact on model
  Add support to model interface for the story
  Implement methods in the mock objects for new
      methods confirming they were called
      or returning typical data
  For everything that can possibly break for this story
    Do
      Write a test for the presenter that exercises the
          app via an event  or action on the
          view or model (an external system event)
      Make assertions on the state of the model
          and the state of the view
      Implement private methods in presenter
    Until the test passes
  Create a minimal user interface implementation
      to satisfy the view interface for this story
```

Figure 3. Algorithm for doing presenter first development.

are properly placed on the screen, and that they are programmed to fire the correct event type. A consequence of the shallow view is that the view interface takes only primitive types as parameters. Sending complex types to the view would require processing in the view, widening the view and requiring unit testing.

The presenter is intentionally stateless and has no public methods. The unit tests of the presenter implicitly test the proper implementation of the wiring of the application as expressed by the presenter's private methods. The application's functionality is coordinated between the view and the model by the presenter. This makes the presenter, and its suite of unit tests, a cohesive and centralized source of documentation for the application's behavior. In effect the presenter and its tests are a living, executable specification for the application. Our experience is that changes and additions to application requirements (as distinct from user interface tweaks) can often be handled solely in the presenter. Because the presenter has thorough unit test coverage, this work can be done quickly and confidently, making modifications to the application simple from both the customer and developer perspective.

Communication with the presenter is made possible by the use of an event subsystem to loosely couple the model and view to the presenter. The most common case is for the view to fire events that the presenter consumes, though model triggered events are also possible. Using events to communicate with the presenter allows for separate packaging of components, reduces compilation dependencies, and allows for the same view to be connected to presenters with different behaviors.

Presenter First allows our application development to satisfy one of the most fundamental pillars of the agile approach: developing exactly what your customer wants, and nothing more. The portions of the code most prone to ambiguity and over-development (model and view) have specific and complete requirements based on customer stories. This substantially reduces over-engineering of code and wasted development time.

3.1 Scaling Presenter First

Our naive first approach to applying the MVP pattern ran into scalability problems. One problem is where the model, view, and presenter objects are created. Our approach is to pass the model and view to the presenter via its constructor. This arrangement of dependency injection allows straightforward testing of the presenter, and answers the question of where model and view objects are created. The remaining, crucial question of where the presenters should be created is deceivingly simple. Answering it lead us to a breakthrough in scalability of the MVP pattern.

We first tried instantiating and wiring together all of the application's MVP triads from the main function. This soon became brittle, tedious, and un-testable. For larger, more dynamic applications we experimented with classes we called constructors and composers. Similar to factories, these classes were given the sole job of creating MVP objects. The repetitive, predictable nature of these classes and a dislike for silly typing inspired us to build or find tools to generate them.

3.2 Tools

Good tools help make Presenter First practical for large-scale applications. The importance of good tools cannot be underestimated. Presenter First requires the creation of many, many interfaces and test classes. Automating the tedious aspect of this work makes development more fun and improves the likelihood of sticking with the process.

We use passive custom code generators written in Ruby for creating new MVP triplets. All new classes are automatically added to the Visual Studio IDE's project file by the tool.

Using Spring.NET streamlines the composition of MVPs into the application [1]. Leveraging Spring provides two significant benefits: painless MVP

construction and the ability to test that construction. By relying on constructor based dependency injection for all application composition, both the pain of adding many small classes to the application was reduced and cyclic dependencies were easily avoided. Spring.NET also allows us to test that we are properly creating all necessary model, view, and presenter objects.

The Resharper plugin for the Visual Studio IDE aids in the liberal use of interfaces throughout the application [9]. It provides a micro level of just-in-time passive code generation that aids in stubbing methods in interfaces and implementations, as well as quick templates for subscription patterns and conventions we developed.

Since almost every class created in the application has a corresponding interface, we use NMock to create dynamic mocks for the models and views when testing presenters [10]. Resharper templates are also used for test case method generation, NMock dynamic mock creation and usage, and other convenient shortcuts. NMock removes much of the tedium of creating mock objects for the interaction testing we favor when using Presenter First [11].

4. Example Uses

Atomic Object works closely with manufacturers of low-volume, high-value products for which software represents a competitive advantage. Presenter First was developed and refined during projects with two of our customers. X-Rite is a global provider of color measurement solutions. X-Rite systems comprise hardware, software and services for the verification and communication of color data. Burke Porter Machinery is a world-leader in advanced electrical, mechanical, and software solutions for automotive test systems.

4.1 Adapters for Complex Views

A Presenter can become overburdened when you're trying to present complex, abstract data structures to the user for viewing and editing; there is no natural, it-just-works user component that can handle your structure. It's fine to send up a few words or numbers—every GUI toolkit has a text field or label construct, so view-level implementation is trivial—but what about when you need to show a data-table view of your Contact database? (Some GUI toolkits provide nice data-table controls that can be used without much effort—let's say, though, that there isn't one to suit our needs. For example, .NET's DataTable wasn't good enough for us in a recent project.)

The language level of our Presenter starts out at the domain level; we speak in terms of Contacts, e.g., "the user edits a Contact's name," "the model notifies us of a change in Contact data," and so on. But we've decided to use a 3rd-party data grid whose API provides a lot of flexibility in terms of cells, cell behavior, and cell rendering, but *not* in terms of Contacts or column headers. Wrapping the API's cell-and-string-oriented functions with domain-oriented functions will require some logic, string formatting and parsing, cell-click event handling, etc. We're not comfortable putting that responsibility inside our view, since we're not going to be able unit test that code.

Our initial response was to shift this code into our model, and make the presenter speak between the model and view in terms of cell coordinates and multidimensional arrays of strings. Suddenly our domain language was mixed with (or replaced entirely by) "widget" language. In addition to (or instead of) editing Contacts, we're pushing typeless string arrays around, subtracting column indices and row counts to account for headers, reinterpreting mouse clicks, etc. And worse, our model has become overloaded with the responsibility of data conversion, data cell location offsets, and internal management of both Contacts *and* a translated version of presentation data. This is very hard to test, very hard to read, and hard to understand if you just wanted to know what the model was supposed to *do*. And reading the presenter doesn't help much anymore.

Wouldn't it be nice if our view just handled Contacts directly? We could extract edited Contact objects from the view and store them in the model, and vice versa.

Since we're reserving the "view" term for actual concrete views, let's insert an adapter in between the view and the presenter. The presenter now references the adapter, which in turn references the view. We clean up our presenter and model to speak only about Contact-centric events and to deal only in Contact objects. Now, it's the adapter's responsibility to handle the translation. By adding this extra layer of abstraction between our presenter and view, we have a logical, isolated zone for translating GUI-centric events and data structures to and from Contact-centric events and structures. Figure 4 shows the new classes and their event messaging relationships.

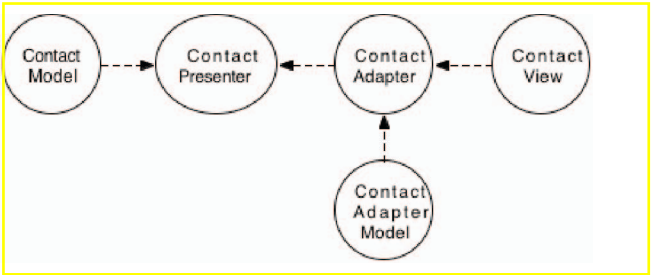

Figure 4. Adapter classes extend MVP for complex views.

If we think of the adapter as a sort of presenter for our custom user control, we might want to give it an adapter model. Incoming data from the presenter, pushed into the adapter via a public setter, could be passed blindly into the AdapterModel, which could carve up Contacts and create string arrays, emit an event saying the grid data has changed (which the Adapter is listening for), and the adapter could pull that string data out of the AdapterModel and push it into the view. The lower-level events from the view could be forwarded down to that same AdapterModel, which could use its correlative knowledge of Contacts and the string-array representation of that data, to convert `"Cell 0,2 Clicked"` into `"UserSelectedContact2"`, which could then be relayed through the adapter, which the presenter is listening to, and so on.

We've effectively created an MVP triplet whose language is geared toward the problem of a custom user control for Contact data without polluting the end-user domain language of contacts. We chose the name adapter because it seemed to make semantic sense. However, these classes also fit the classical Adapter design pattern, in that it bridges the gap between our desired interface and the reality of available user controls.

4.2 Automotive Test Cell Controller

The Presenter First technique was used consistently throughout the development of a visually configured automotive test and measurement application for Burke Porter Machinery. The application consists of approximately 1200 classes, 100 simple MVP triads and 20 adapters. A toolbox allows users to drag graphical elements onto a canvas. Depending on the type of canvas that is currently active, either display elements (dial gauges, strip charts, etc.) or control elements (machinery mode control commands, logging controls, etc.) are found in the toolbox.

Code for the toolbox MVP triad is shown in Appendix 1. Looking at the `ToolboxPresenter` constructor we can quickly see the view must notify the presenter when an item is dragged. The model must notify the presenter when the available toolbox items change (when the canvas switches between display and control elements.) The final line in the constructor insures the toolbox is initially empty.

The two private methods define what is done when the view and model fire events to indicate something of interest has occurred. These methods partially define the required public interfaces of the model and view. In the case of `ToolboxItemsChanged`, the view is updated with a list of names and images to use. `ItemDrag` gets the selected item from the view, verifies the selected item can be dragged via the model and then passes the required data to the view to initiate the window's `DragDrop`.

The model is constructed with the two types of registries (`widgetRegistry` has the display elements, `seqRegistry` has the control elements) and a holder. The holder knows what type of canvas is active so the `ToolboxModel` subscribes for notification of changes to the `ActiveDocument` (i.e. canvas). If the active document changes, the `ActiveDocumentType-Changed` method notifies the presenter of the change. It's up to the presenter to decide what to do about the change. Other methods provide the presenter with the required data (extracted from the registries that the model was constructed with.)

The view was the last code written and has very little logic in it. `SetToolboxItemTypes` takes an array of names and images (simple data types) and converts them to the format needed by this particular view (an image list.) Data is then added into the listview. `GetSelectedToolboxItemIndex` is a more typical view method in that is has only a single line of code. In this case it returns the index of the selected item in the listbox. Events from the view are typically wired up to fire events to the presenter as demonstrated by the `m_listView_ItemDrag` method.

4.3 Modal Dialog

A modal dialog box is a hard stop in the flow of an application; users cannot do anything else in the application until they respond to it. Modal dialogs are most commonly used to display information or gather data before the application can continue. Although we always strive to keep the GUI elements separated from the business logic, it is especially important in the case of modal dialogs—we do not want something out of our control halting the progress of test suites.

Presenter First provides a solution that cleanly separates the dialog box action from the logical components of the application we wish to test. The following code sample demonstrates how this is done.

```
// Model code
if (unable to open file) {
  FireMessageBoxEvent("can't open file");
}

// Presenter code (in the constructor)
model.SubscribeForMessageBoxEvent();

// upon catching a messagebox event
view.ShowMessageBox(event.Text);

// View code
ShowMessageBoxEvent(text){
  MessageBox.Show(text);
}
```

Although this takes care of one problem with modal dialogs, it neglects another: quite often, dialogs are used to gather information back from the user. This raises the question of how the user's response is best handled. It is tempting to have the view handle this, but this is difficult in that many languages do not support returning data from events. And some languages that do support return values also support multiple listeners for those events. This makes predicting the return value of an event from multiple listeners ambiguous. In addition, this method takes a lot of code in several different class to accomplish what is essentially a logging function (or database access function in the case of message boxes that return information).

Our solution is to compose the model object with a reference to a "shower" object. The shower object's responsibility is to abstract away the details of displaying and retrieving data from the user.

Let's consider the modal dialog's purpose again. We pop one up to inform the user of something, or to obtain their input, and then we're on our way again. From the model's perspective it is just log information in the case of a message, or a database access in the case of getting information. The dialog doesn't care if the user sits there three second or for an hour before finally hitting the magic [OK] button – the model makes the call, and when the call returns we move on.

We've seen this before: making calls on objects we've been composed with. Need to access the database? Compose your model with the IDatabaseReader interface. Need to log messages? Compose your model with the ILogWriter interface. As we already know, by accessing a composed interface, the code remains easily testable and decoupled from the functionality represented by that interface.

Whenever the model needs data from the user, it can delegate this duty to the shower. The following is the code from above modified to use a shower:

```
if (!m_fileReader.Open(filename)) {
m_messageboxShower.ShowErrorMessage(
    "Cannot open the file");
}

// or, for fetching information:
string filename =
       m_openFileDialogShower.Show(
              "Select file to load");
if (filename != null) {
  // Do something
}
```

Tests for this code are just as simple:

```
// happy day case...
fileReaderMock.ExpectAndReturn(
  "Open", true, filename);

// failure handling test
fileReaderMock.ExpectAndReturn(
  "Open", false, filename);
messageboxShowerMock.Expect(
  "ShowErrorMessage", "Cannot open the
file");
```

So, there you have a simple, testable method of displaying those modal dialogs we all use every day. But what about our own custom modal dialogs? What if the modal dialog itself is represented by one (or more) MVP triads?

The solution, from the calling model's standpoint, is pretty much the same. You add a shower interface for your modal dialog, and users of the dialog access it through the shower. The tricky bits are found in the model and the view of your modal dialog. From the modal dialog user's perspective there is little difference between popping up a built in dialog via a shower and popping up a custom, MVP based one:

```
m_fancyModalDialogShower.Show();
```

To make this work on the inside, however, requires a bit of finesse. There are three basic steps to running a PF based modal dialog: initialize it, show it, close it.

The shower is composed with an interface to the dialog's model. All interaction between the dialog and shower are funneled through the model. Figure 5 represents the event flow between the various objects.

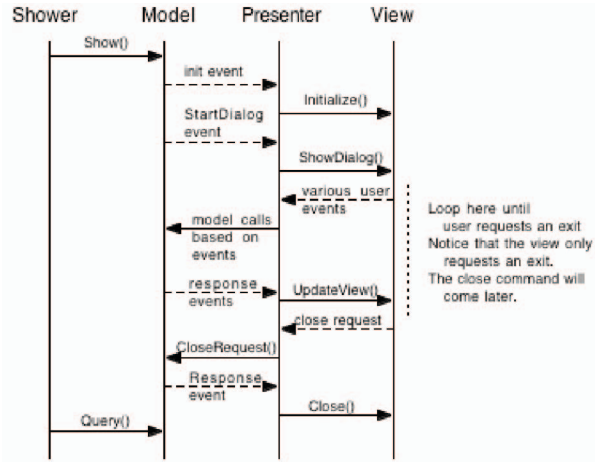

Figure 5. Sequence diagram for modal dialog MVP with shower.

Notice that as the dialog runs, the model stays up-to-date with the information content of the dialog. As in any other PF construction, we want our view to remain as stateless as possible. All information relative

to the outside world should be maintained in the model. The view part of the dialog may request a close in more than one way—an Accept event, a Cancel event, or just a Close event (usually treated the same as a cancel). It is the responsibility of the model to determine the proper result code to return to the shower.

One aspect you may notice in this construct is that the Shower is a thin wrapper for the model. Users of the shower class will call Show(), and make queries on the results. In most cases these calls will be directly mapped to similar (if not identical) methods on the model. We like to use the shower here anyway, as it leaves the consuming class completely ignorant of the construct of the dialog. It appears to the consumer as just another base modal dialog.

5. History

Presenter First builds on other people's good ideas and has evolved over the course of many of our projects and evolutionary dead-ends. This section describes some of this legacy.

The Model View Presenter pattern derives from the classic Model View Controller [7] pattern of Smalltalk, was first described by Taligent [5], was widely used by Dolphin [4], and is in the process of being documented by Fowler [6]. The intent of MVP is to separate business rules from the user interface, as with MVC, but to further isolate the behavior of the application from the mechanics of the presentation.

Michael Feathers' Humble Dialog Box improved the testability of our applications with graphical user interfaces by removing all functionality and logic from the view [8]. Humble widgets shift logic from the view to the model, creating a "smart object" (model and presenter) and a "dumb object" (view). The smart object is more testable, as it is decoupled from the GUI. The dumb view object is so thin that it does not require unit testing. Humble dialog took us a step forward, but was ultimately insufficient. We found evolving smart objects over time difficult as the behavior of the application is now bundled with the business logic of the application.

5.1 Why the Presenter First?

The process aspect of Presenter First addresses the question of how development is organized over time, and how the application is tested while it is being developed. The first question every application development effort faces is: where to start? For applications based on MVP, there are three possible answers. Starting with the model is a form of the "infrastructure first" mistake of traditional software development [3]. Drawbacks to this include building the model before knowing with certainty what it needs to support, and focusing early development on things invisible to the user. We feel that the key to success is to delay working on the model until the requirements for the model have been uncovered by feature requests from the customer.

Starting with the view seems quite logical when following a customer prioritized feature-driven development process. The logic of this approach is seductive: customer stories describe actions taken on the view and results shown in the view, feedback from the customer requires some interface for them to use the application, and the importance of the model (infrastructure) is minimized. Unfortunately view first is an easily made and expensive mistake.

View first development has several drawbacks. The view is special in that it tends to attract strong feelings, a hesitancy to commit to specifics, and a high rate of change requests from customers. In our experience, user stories rarely specify detailed interfaces, but do describe application functionality more generally. The remaining ambiguity can mean spending long hours creating an interface only to have it be dismissed by the customer. In addition, focusing on the view tends to increase the danger of fattening the view with business logic. Lastly, the difficulty of testing the interface undermines the desirable test-driven development cycle.

In our experience, the best alternative of the three possibilities is to start with the presenter. By starting with the presenter, and organizing development around it, the application may be built from user stories following test-driven development practices. Unit tests on the presenter are economical to write and maintain, and confirm the correct operation of the application's functionality without being coupled to specific interface elements.

6. Summary

Presenter First is an economical, effective technique for building large GUI applications and driving development from user stories in a test-driven fashion. Specifications for model and view classes develop naturally from user stories, thereby avoiding the overbuilding that is common for these types of classes.

Presenter First forces the separation between interface and business logic. The view of an application can be replaced without disturbing the application or business logic classes.

Once mastered, Presenter First can be used to build a large GUI application in a way that is repeatable, maintainable, thoroughly tested, and user specified.

Further examples, descriptions and lessons learned using Presenter First are available at http://atomicobject.com/presenterfirst

Acknowledgments

Thanks to Matt Fletcher and Scott Miller for help writing and editing this paper.

References

[1] Spring.NET Application Framework, www.springframework.net/

[2] Carl Erickson, Ralph Palmer, David Crosby, Michael Marsiglia, Micah Alles, "Make Haste, not Waste: Automated System Testing", *Extreme Programming and Agile Methods - XP Agile Universe 2003*, New Orleans, LA, USA

[3] Ron Jeffries, "Implications of delivering software early and often", *XP West Michigan User Group*, xpwestmichigan.org/site/node/29, September 2004.

[4] Andy Bower, Blair McGlashan, "Twisting the Triad: The evolution of the Dolphin Smalltalk MVP application framework", *European Smalltalk User Group (ESUG)*, 2000.

[5] Mike Potel, "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", Taligent Inc, 1996.

[6] Martin Fowler, "Model View Presenter", www.martinfowler.com/eaaDev/ModelViewPresenter.html, July 2004.

[7] Trygve Reenskaug, "MODELS - VIEWS - CONTROLLERS", Technical note, Xerox PARC, December 1979.

[8] Michael Feathers, "The Humble Dialog Box", Object Mentor, 2002.

[9] Resharper Visual Studio.NET Plugin, http://www.jetbrains.com/resharper/

[10] NMock: A Dynamic Mock Object Library for .NET, http://www.nmock.org/

[11] “Mocks Aren’t Stubs,” Martin Fowler, www.martinfowler.com/articles/mocksArentStubs.html, July 2004

Appendix 1. MVP for Toolbox in Automotive Test Cell Controller

```
//
// View interface
//
public interface IToolboxView
{
  void SubscribeItemDrag(UpdateDelegate listener);
  void SetToolboxItemTypes(string[] typeNames, Image[] imageList);
  int GetSelectedToolboxItemIndex();
  void StartDragDrop(BepHostDragDropData data);
}

//
// Model interface
//
public interface IToolboxModel
{
  void SubscribeToolboxItemsChanged(UpdateDelegate listener);
  string[] ToolboxItemTypes { get; }
  string[] FriendlyToolboxItemNames { get; }
  Image[] ToolboxIcons { get; }
  BepHostDragDropData CreateDragDropData(int idx);
}

//
// Presenter
//
public class ToolboxPresenter {
  private IToolboxModel m_model;
  private IToolboxView m_view;

  public ToolboxPresenter(IToolboxModel model, IToolboxView view) {
    m_model = model;
    m_view = view;
    m_view.SubscribeItemDrag(new UpdateDelegate(ItemDrag));
    m_model.SubscribeToolboxItemsChanged(new
              UpdateDelegate(ToolboxItemsChanged));
    m_view.SetToolboxItemTypes(new string[0], new Image[0]);
  }

  private void ToolboxItemsChanged() {
    m_view.SetToolboxItemTypes(m_model.FriendlyToolboxItemNames,
              m_model.ToolboxIcons);
  }

  private void ItemDrag() {
    int idx = m_view.GetSelectedToolboxItemIndex();
    BepHostDragDropData data = m_model.CreateDragDropData(idx);
    if (data != null) {
      m_view.StartDragDrop(data);
    }
  }
}
```

```
//
// Selected portions of ToolboxView.cs, pertinent to the subscription for
// ItemDrag events, and the relaying of the lower-level GUI event that causes
// us to emit ItemDrag.
//
public class ToolboxView : UserControl, IToolboxView
{
  private ListView m_listView;
  private event UpdateDelegate ItemDragEvent;
  ...
  private void InitializeComponent()
  { ...
    this.m_listView.ItemDrag +=
       new System.Windows.Forms.ItemDragEventHandler(this.m_listView_ItemDrag);
    ...
  }
  public void SubscribeItemDrag(UpdateDelegate listener)
  {
    ItemDragEvent += listener;
  }
  private void FireItemDrag()
  {
    UpdateDelegate evtCopy = ItemDragEvent; // Make a copy to avoid concurrent
                                                    // mods to dispatch list
    if (evtCopy != null) // avoid no-subscribers situation
      evtCopy(); // Fire the event
  }
  ...
}
//
// Model implementation
//
public class ToolboxModel : ItoolboxModel {
  // ivars not shown
  public ToolboxModel(IToolboxItemRegistry widgetRegistry,
                      IToolboxItemRegistry seqRegistry,
                      IActiveDocumentHolder holder) {
    // ivar initialization
    m_holder.SubscribeActiveDocumentChanged(
                       new UpdateDelegate(ActiveDocumentTypeChanged));
  }
  private void ActiveDocumentTypeChanged() {
    // code to check what type of document is now active and check I
    //   there is a new type document was removed
    if (docTypeHasActuallyChanged){
      FireToolboxItemsChanged();
     }
  }
   // List of human-friendly widget names.
   public string[] FriendlyToolboxItemNames { … }
   public string[] ToolboxItemTypes { … }
   public Image[] ToolboxIcons { … }

   public BepHostDragDropData CreateDragDropData(int idx) { … }
   private void FireToolboxItemsChanged(){ … }
   public void SubscribeToolboxItemsChanged(UpdateDelegate listener) { … }
}
```