Presenter First: TDD for Large, Complex Applications with Graphical User Interfaces

Michael Marsiglia
Atomic Object
mike@atomicobject.com

Brian Harleton
X-Rite, Inc.
bharleton@xrite.com

Carl Erickson
Atomic Object
carl@atomicobject.com

Abstract

Presenter First extends the benefits of functionality organized, customer prioritized, test driven development to complex applications with graphical user interfaces. A variation of the Model View Presenter pattern is used to isolate and test customer specified functionality. Unit testing the presenter allows for test driven development to be applied to customer described functionality. This creates an executable design specification for the behavior of the application, independent of the implementation of the user interface and business logic. Given the difficulty and cost of automating system tests for applications with GUIs, unit testing the presenter is a more economical means to achieve test coverage of system functionality. In addition, by concentrating on the functionality specified by the customer and developing the presenter first, minimal and complete requirements for the model and view are automatically uncovered. A complex color measurement .NET/C# application suite developed by the authors is used to illustrate Presenter First.

1. Motivation

Graphical user interfaces are painful! We've all been there. The graphical user interface was so simple at first glance. How could we ever have created such a massive mess so quickly? Why are small changes to our design so painful? How did we ever get into the trap of creating such a large and time intensive piece of code at the time of maximum ignorance about the feature? How did we stray so far from the goal of feature-centric development? Why is it so difficult to write the needed unit tests that will surely relieve our stress and pain? How are we going to tell the customer that it is going to take one week to make a simple change inside the custom dialog?

The first step is to separate business logic and interface. To use the terminology of the Model View Presenter pattern [6], the model is isolated from the presenter and view. Using this technique we are able to unit test our business logic (the model) and follow TDD principles. This approach is not sufficient, however, since it leaves the application flow or functionality encoded in the GUI (the presenter and view). It is possible to test GUI functionality using an automated system level test framework [2]. In our experience it is expensive to build and maintain these test suites.

Michael Feathers' Humble Dialog Box improves the testability of applications with graphical user interfaces by removing all functionality and logic from the view [8]. This approach shifts the presenter from the view to the model, creating a smart object. Managing this smart object is not ideal as the flow and logic of the user interface are now coupled to the business logic of the application. All changes to the application flow directly effect the business logic and vice versa. Because of this coupling, requirement changes are expensive to handle.

Ideally we want an approach that will allow us to follow the Extreme Programming practices of user prioritized, story-based, test-driven development for complex GUI applications in a cost effective manner [1]. In order to achieve this goal we need to use TDD throughout the development process. We need to strive for clean, robust, decoupled, and completely tested code. We want code and process that make requirement changes easy and inexpensive.

Presenter First addresses the problem of building complex applications and application suites with graphical user interfaces. It helps us decouple development from the ever changing interface, and concentrate on the problem of functionality. Because user stories are generally functionality-centric, Presenter First encourages us to think in terms of functionality and not buttons, sliders, and checkboxes.

Presenter First provides us with tested, maintainable code and a process for the implementation and testing of the application's features.

2. Presenter First

Presenter First is a combination of process and pattern. The pattern is a variation on the Model View Presenter (MVP) design pattern. The process aspect of Presenter First determines how the application is built and tested using MVP. The result is to push the boundary of the effectiveness of test driven development out to include system functionality described and prioritized by the customer.

Acceptance tests validate the functionality of the system from the customer's perspective. Since the customer's perspective includes the application's interface, the obvious approach to automating acceptance tests requires automation of the interface. This approach is complicated and expensive, particularly when the interface is graphical. An alternative approach that seems to have become conventional wisdom in the Extreme Programming community is to make the GUI as thin as possible and test to the layer just beneath it. Presenter First provides a concrete realization of this conventional wisdom.

The Model View Presenter pattern derives from the classic Model View Controller pattern of Smalltalk [7], was first described by Taligent [5], was widely used by Dolphin [4], and has been more recently described by Fowler [6]. The intent of MVP is to separate business rules from presentation, as with MVC, but to further isolate the behavior from the mechanics of the presentation. The three components of MVP are:

Model The business logic and data of the application. Invisible and irrelevant to the customer.

View The interface of the application. To the customer, the view is the application. Customer stories describe functionality in terms of doing something to the view and seeing results in the view. The view will therefore have a high rate of change throughout the project. Decoupling from this change is a major motivation for Presenter First.

Presenter The presenter is the custom logic that allows the model to interact with the view, and vice versa. It represents the functionality or flow within the MVP triad. Customer stories or requirements correspond to functionality in the presenter.

The variation of MVP that is used in Presenter First concerns the patterns of communication within the triad. The original MVP pattern allowed for the model to directly communicate with the view. Presenter First requires all communication to flow through the presenter. This restriction insures isolation between the model and view, and greatly improves the testability of each element of the MVP triad. Figure 1 contrasts the patterns of communication in MVP and Presenter First.

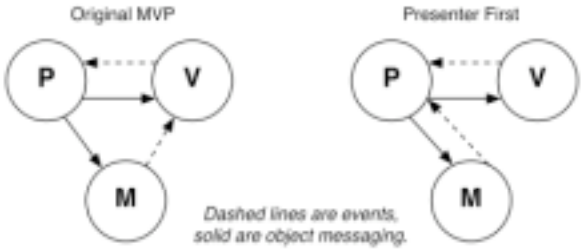


Figure 1. Communication in original MVP versus Presenter First approach.

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" approach 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. The model should be easy to test, and easy to develop in a test driven fashion. The key 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 using MVP in 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 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 Presenter First. 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. As a consequence of developing the presenter first, a minimal and complete specification for the model and view are created, while the developer's focus remains on customer functionality.

While it is an implementation option in MVP, Presenter First requires interfaces for both the model and the view. Communication with the presenter is handled via the event subsystem, decoupling both the model and view from the presenter. Figure 2 shows the relationships between classes in Presenter First.

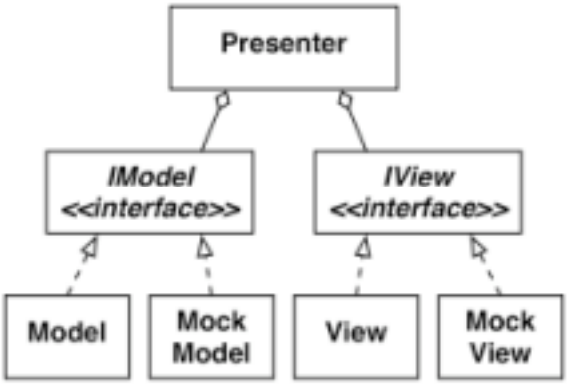


Figure 2. UML class diagram of Presenter First MVP classes, tests classes, and interfaces.

2.1 How it works

A strength of Presenter First is the simple, concrete nature of the approach. Development proceeds as follows:

1. Create a stub presenter class that takes a model interface and a view interface in its constructor

2. Create mock test objects that satisfy the model and view interfaces

3. For all user stories
   3.1. Select a prioritized user story
   3.2. Analyze story for impact on view
   3.3. Add support to view interface for the story
   3.4. Analyze story for impact on model
   3.5. Add support to model interface for the story
   3.6. Implement methods in the mock objects for the new interface methods which confirm they were called, or which return typical data
   3.7. For all things that can break (TDD loop)
      3.7.1 Do for each test
         3.7.1.1. Write a test for the presenter that exercises the app via an event or action on the view or model (an external system event)
         3.7.1.2 Make assertions on the state of the model and the state of the view
         3.7.1.3 Implement private methods in presenter
      Until the test passes
   3.8. Create a minimal user interface implementation to satisfy the view interface for this story

The outer loop of the Presenter First process requires frequent refactorings of the model and view interfaces. These refactorings are not expensive, however, since at this point there are only mock implementations of these interfaces.

One very nice consequence of Presenter First is that when the presenter has encoded all current requirements from the customer, the model and view interfaces are minimal as well as complete specifications for their respective implementations. Completing the application at this point is a matter of implementing the model following standard TDD practice, and implementing the view to satisfy the view interface and the customer's desires.

In order to get feedback from the customer, we typically create a very simple user interface with a minimum amount of time and effort. The customer may decide that the interface needs to be redesigned by a usability expert, or designed by marketing, or is actually good enough for its purpose. In the meantime, the application's functionality may be exercised, only a small effort has been expended on interface code, and new interfaces may be readily developed in the future which satisfy the view interface.

Testing the actual view may either be automated, if the tools and budget allow, or left to a final manual system test phase. The view is very thin, with methods which consist of little more than firing properly typed events to be handled by the presenter. Assuming that the underlying GUI widgets work (buttons click, etc), the main use of unit tests on the view would be to assert that the widgets are properly placed on the screen, and that they are programmed to fire the correct event type.

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 the effort required to support what from the customer's perspective seem like simple modifications and additions to the application, simple to implement for the developer.

Communication with the presenter is made possible by the use of the 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 for the view to communicate with the presenter keeps the logic of the view very shallow. The view delegates all event processing to the presenter. This design results in a "thin GUI" that has almost no behavior of its own to test. 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.

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, fattening the view and requiring unit testing.

2.2 Example use of Presenter First

X-Rite is a global leader in and provider of color measurement solutions. X-Rite systems comprise hardware, software and services for the verification and communication of color data [9]. Atomic Object worked with X-Rite to design a large, complex software system that manages printing quality of a printing press. This system consists of approximately 45 distinct MVP triads and 8 individual applications. Appendix 1 shows a representative screenshot from this application.

The use of Presenter First helped manage the complexity of the suite of applications developed to control press quality. Responding to changes and adding and removing functionality was not a problem and did not impact unrelated portions of the code base. One example of an MVP triad in this project was a means to allow the user to register their product. We had to support multiple means for the user to register as well as supporting multiple screen resolutions. Using Presenter First, this task was quite easy to test and implement.

The class diagram describing the relationships for the registration example with tests is shown in Figure 3. Sample code that demonstrates creating this registration functionality using Presenter First is shown in Appendix 2. Note that the implementation of the presenter `RegPresenter` is not shown. The two tests (`test_Registration_Fail` and `test_Registration_Pass`) show how the same event which `RegPresenter` fires when a user registers the application are used to test the implementation of `RegPresenter`. Assertions are made on the mock view and model to confirm that the registration logic is properly wired in the presenter.

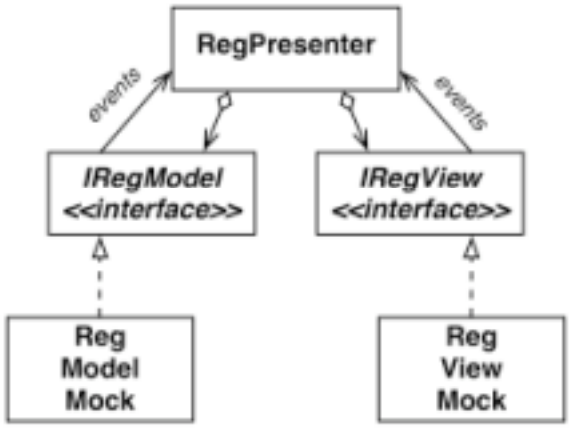


Figure 3. Class diagram for tested Presenter First triad for registration feature.

By using Presenter First, additional customer requests or changes did not present headaches to the developers. For example, if the customer decided to have the user submit the registration code by clicking on a graphical push button instead of just hitting the enter key, the change could be made to the view without affecting either the presenter or the model. If the customer decided to switch from validating the registration code via a database lookup to validating via a web service, the resulting change to the model will affect neither the presenter nor the view.

3. Benefits of Presenter First

Presenter First helps keep the focus on functionality and delivering user prioritized features. Developers will spend less time focusing on the details of features (model and view) and more time on the functionality (presenter). The developer can work at a higher level of abstraction early in development versus getting distracted by details too early.

Functionality is easily testable using Presenter First. The interfaces of the model and view are easily mocked, thus putting the focus of the tests at the functionality level within the presenter class. With the view and model components mocked, this gives the developer complete control over all aspects of the view and model and gives the developer the freedom to easily test all functionality cases within the presenter class.

In addition to improving the ability to create unit tests for all the functionality of the user interface, creating mock classes to represent the view and model give additional benefits as well. By mocking the model interface, the developer avoids problems such as connecting to a live database, having to use communication devices and manipulating files. The responsibility of testing those types of features is put on other classes within the actual model. Presenter First is only concerned with testing the functionality accessible through the user interface, not the details of how this functionality is actually implemented.

Mocking the view has its benefits as well. Since real user interface controls and forms do not need to be instantiated, unit test suites will not be slowed down by these expensive operations. When unit tests are slow and painful to run, they tend to be run less often and therefore less valuable to the development team.

Managing the user interface is very much improved by using Presenter First. The unit tests for the presenter class become the living specification for the functionality of the user interface. By examining the presenter class's unit tests, developers can easily determine the behavior and rules of the user interface.

Functionality is easily managed by the unit tests of the presenter class. When functionality is either added or removed, the presenter class's unit tests provide the confidence that things are still working correctly. Test driven development has been extended to include higher level system functionality.

Although it is well understood that decoupling the user interface from the business logic is an important practice to follow, nearly every developer finds themselves tempted to couple code inappropriately at one time or another. Presenter First forces the decoupling of the user interface and the business logic. Decoupling drastically increases the flexibility of the developer's code making changes much less painful.

When large customers ask for a specific look and feel, this is done easily by simply changing the graphical components. Or perhaps marketing decides the product must be supported on a Palm Pilot. From the user interface perspective, this change is now easily done. Simply replacing the graphical components with Palm Pilot friendly graphical components that implement the same view interface and the user interface changes are completed. Imagine the ease and power of completely changing the supported platform or the look and feel by simply installing a different package. These changes can be made with confidence as changing the graphical component does not change any of the functionality and therefore none of the other code.

By enforcing the decoupling of the components of Presenter First, replacing a component of the MVP triad becomes trivial and does not affect the other components. So long as the view component implements the required view interface, changing what graphical user interface to use is painless. Different graphical user interfaces can easily be swapped in or out. In fact, graphical user interfaces can now be an installation or runtime decision. In the same respect, the model component can be modified or completely replaced without effecting the presenter or view. A user interface that accesses information from a file today could easily be made to access via a database tomorrow with no impact to the presenter or view. We have even used the same view and replaced the presenter in some cases. The graphical user view is the same, but the behavior (presenter) is different.

Presenter First is scalable. Through composition of the MVP pattern, complex applications are effectively suites of many small applications. Each mini-application is itself an MVP triad. These mini-applications are manageable and completely decoupled from one another. Structurally, this increases the manageability of the project as pieces can be made independently and put together later without fear of dependency issues. Patterns for building and composing the MVP triads help maintain testability of nested triads.

Presenter First is cost effective and practical for teams. Development is streamlined and manageable. The presenter portion can be completely tested and finished without having to have the view and model started, let alone completed. So long as they hold up

their end of the contract (model interface and view interface), separate teams can be creating code independently of each other at separate speeds and timelines.

Using Presenter First, the graphical view becomes so simple and thin that with the proper tool, graphic designers or usability experts can be trained to create the graphical view. Developers no longer waste large amounts of time implementing graphical user interfaces. Developers generally are not qualified or trained to create and implement graphical user interfaces and therefore end up spending a lot of time creating a poor graphical user interface. Presenter First allows developers to do what they do best, which is program, and avoid spending time on things they are not trained to do, which is creating usable interfaces.

10. References

[1] Kent Beck, *Extreme Programming Explained*, Addison Wesley, 2000.

[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*, http://xpwestmichigan.org/meeting040928.page, 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.

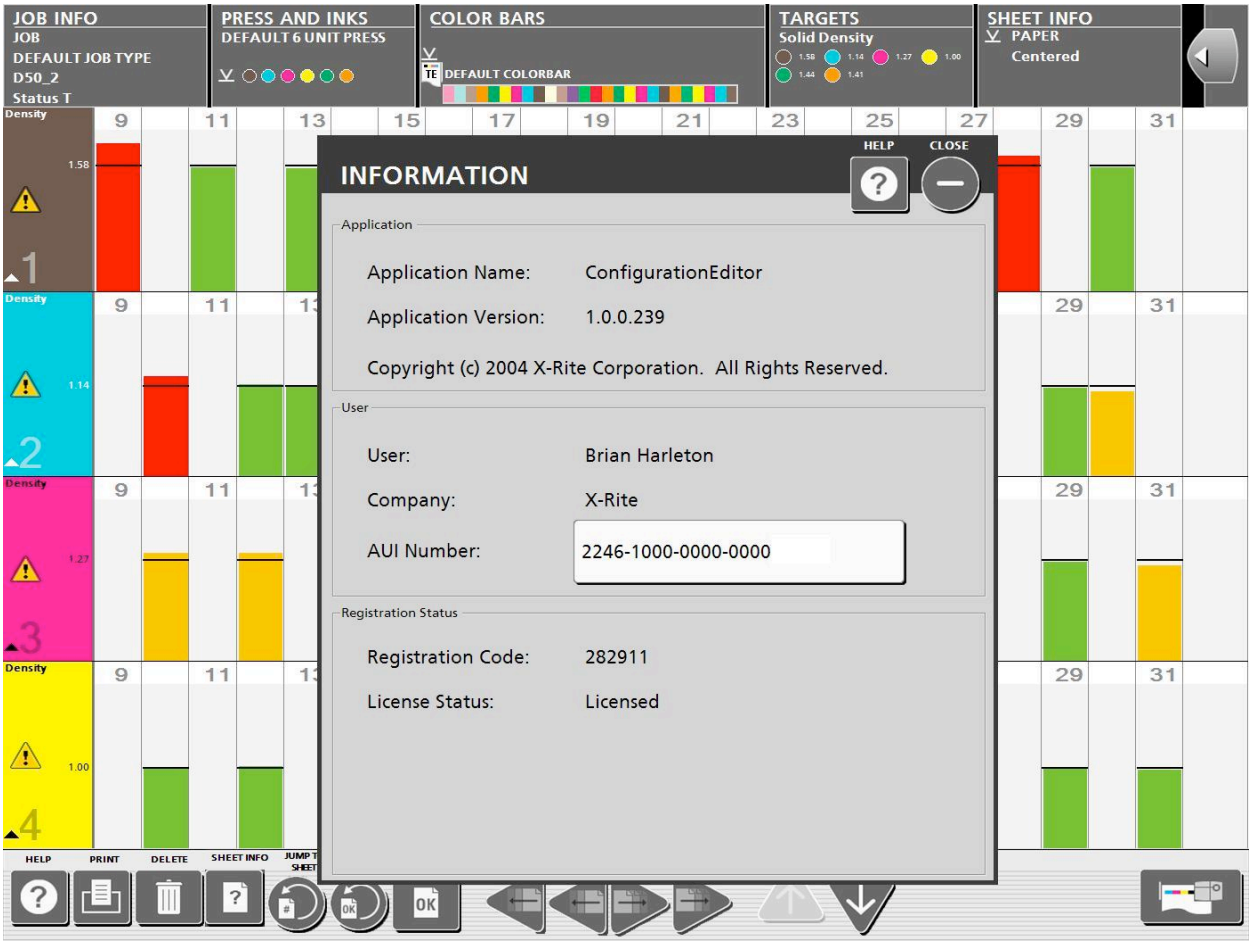
[6] Martin Fowler, "Model View Presenter", www.martinfowler.com/eaaDev/ModelViewPresenter.html, July 2004.

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

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

[9] X-Rite, Inc., http://xrite.com/

Appendix 1. Screenshot of X-Rite press quality control application.



Appendix 2. Sample code showing model and view interfaces, mock test classes, and test methods.

```
public delegate void RegSubmitted();

public interface IregView {
  void AddRegSubmittedEvent(RegSubmitted ev);
  string GetSubmittedData();
  void SetRegistrationStatus(bool bSuccess);
}

public interface IregModel {
  bool Register(string strRegistration);
}

public class RegViewMock : IregView {
  private RegSubmitted regSubmittedEvent;

  public void FireRegSubmittedEvent() {
    RegSubmitted evtCopy = regSubmittedEvent;
    if (evtCopy != null) {
      evtCopy();
    }
  }

  public void AddRegSubmittedEvent(RegSubmitted ev) {
    regSubmittedEvent += ev;
  }

  public string GetSubmittedData() {
    return "ThisIsMyRegistration";
  }

  public bool bRegistrationStatus;
  public void SetRegistrationStatus(bool bSuccess) {
    bRegistrationStatus = bSuccess;
  }
}

public class RegModelMock : IregModel {
  public bool regSuccess;
  public string regInput;

  public bool Register(string strRegistration) {
    regInput = strRegistration;
    return regSuccess;
  }
}
```

```
public void test_Registration_Pass() {
  /* force the registration to succeed */
  modelMock.regSuccess = true;

  /* Simulates the user submits event by firing an event
   * that the presenter should handle.
   */
  viewMock.FireRegSubmittedEvent();

  /*  Now just verify that what the user entered as the
   *  registration code is what the model actually uses
   *  as input.
   */
  Assert.AreEqual("ThisIsMyRegistration", modelMock.regInput);

  /*  Verify that the view has been updated to reflect
   *  a successful registration.
   */
  Assert.IsTrue(viewMock.bRegistrationStatus);
}

public void test_Registration_Fail() {
  /* force registration process to fail */
  modelMock.regSuccess = false;
  viewMock.FireRegSubmittedEvent();

  Assert.AreEqual("ThisIsMyRegistration", modelMock.regInput);

  /* verifies that the view was updated to reflect
   * an unsuccessful registration.
   */
  Assert.IsFalse(viewMock.bRegistrationStatus);
}
```