

February 2007 \$9.95 www.StickyMinds.com

BETTER SOFTWARE


The Print Companion to  **StickyMinds.com**

THE MAGIC NUMBER
Three techniques for
testing with databases

COMPLEX CODE?
Get back to basics

The Case of the Missing Fingerprint

**Solve the Mystery of
Successful End-of-Project
Retrospectives**



Atomic Object developers have hammered out a software development method that bridges the language gap between code and customers, opens wide the door to test-driven development, works with many domains and languages, scales to complex applications, and, once mastered, can be practiced with almost mechanical repetition. We call the approach Presenter First, and we think you can use it, too.

In Presenter First, we write a class called a presenter, and we write it first. This means we select a user story and write it into our code—preserving its spoken-word semantics—programming by assumption against emerging interfaces. We then fill in the blanks, adding supporting classes (often models and views) until the customer accepts the feature as

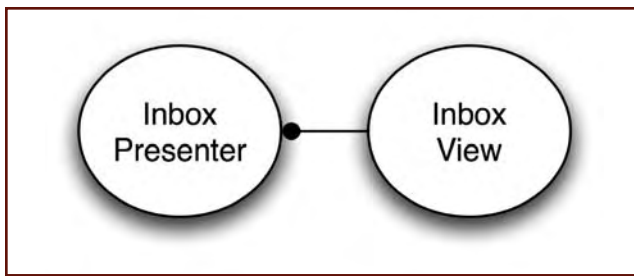
BIG, COMPLEX, AND TESTED? JUST SAY “WHEN”

Software Development Using Presenter First

*by David Crosby
and Carl Erickson*

complete. There usually are many presenters in an application—as many or more than there are user stories. We’ve built applications with hundreds of model/view/presenter triads, and we’ve refined Presenter First over the course of time, working with our customers in a variety of projects and domains (see the StickyNotes for the Presenter First resource page).

Here’s an example. Our customer might say, “When I click the mail icon, my inbox should appear.” From this we create the InboxPresenter class, expressing its high-level functionality in terms of the not-yet-written class InboxView. Figure 1 shows an object composition diagram and corresponding Ruby sample code for a presenter for this user story. Ruby makes for tight, readable examples in an article, but we’ve applied Presenter First in many languages, including C#, Java, C, ActionScript, and OpenLaszlo.



```
class InboxPresenter
  def initialize(inbox_view)
    inbox_view.when :mail_icon_clicked do
      inbox_view.display_message_list
    end
  end
end
```

Figure 1: At the top is a composition diagram of inbox objects. `InboxPresenter` has an `InboxView`. Below the diagram is Ruby sample code representing the presenter for the inbox story.

Notice the usage of the word “when” in the code in figure 1. “When” is a big word in Presenter First, because it is such a common word in user stories. “When I quit the program, I want to be warned about unsaved changes,” the customer says. Or she says, “The dock icons should bounce,” and we have to ask, “When should they bounce?” Just about everything in a desktop application happens as a response to user action. At first, the use of “when” in method names in our code felt awkward and unfamiliar, and we pondered this. We realized that while code libraries are generally code-oriented—linguistically biased toward writing code to accomplish feats of programming—our code exists to define the user’s intentions and expectations of system behavior. We stopped being embarrassed about having code that looked different from the rest of the world’s. For example, the line `inbox_view.when :mail_icon_clicked` means “subscribe to the `mail_icon_clicked` event on the `InboxView`” in traditional code-speak, but in Presenter First, we’re trying to represent customer concerns as clearly and directly as possible. The customer says, “When I click,” not, “You should write two objects that interact according to the Observer pattern.”

At this point we have neither the `InboxView` nor code to instantiate the `InboxPresenter`. But we now know that such code is needed and what is required of these objects: The `InboxView` has to notify us when the mail icon is clicked (whatever the “mail icon” ends up being), and it must be able to display the message list. We know where to go next because we’ve encoded the wishes of the customer and are now staring at a list of compiler errors telling us what we need to do.

Let’s take a quick peek at the view, whose interface has been defined by the needs of the `InboxPresenter`. It’s likely our GUI library uses slightly different terminology, so we’ll have to do some translation from our idealistic constructs to the reality of the windowing API we’re using:

```
class InboxView < FXMainWindow
  can_fire :mail_icon_clicked

  def initialize(application)
    super(application, "Email Inbox")
    panel = FXHorizontalFrame.new(self, 0, 0,0,0,0, 5,5,5,5)

    @mail_icon = FXButton.new(panel, "Inbox", load_icon("mailbox.jpg"))
    @mail_icon.connect(SEL_COMMAND) do
      fire :mail_icon_clicked
    end

    @message_list = FXList.new(panel, nil, 0,0,0,0, 100,0)
    @message_list.hide
  end

  def display_message_list
    @message_list.show
  end
  ...
end
```

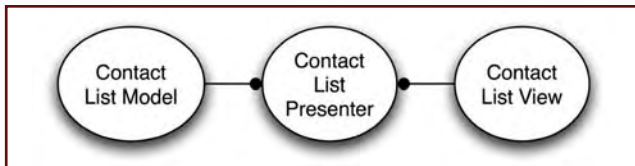
The particulars of the GUI toolkit (this sample is based on FXRuby; see the StickyNotes for more information) should not influence the external interface of our View class. We write code internally according to the interface of the toolkit and do the simplest thing we can to map the demands of the presenter to the toolkit’s capabilities.

Looking back at the `InboxPresenter`, we notice that it has no public methods aside from the constructor. All communication into a presenter is done with events. Those events cause the presenter to act outward on other objects. A presenter depends on other objects, but no other objects depend on it or ever hold a reference to it. In a sense, the only real client of a presenter is the customer herself!

This first example illustrates only some of the basic advantages of Presenter First—we knew where to begin and how to drive the creation of the view. Next we will see how a presenter can insulate a view from its underlying application model and vice versa.

Suppose the customer wants a contact management tool added to her application. She tells us, “I want to see a list of the names of my contacts, and I want to be able to click on one and edit it.” (She didn’t say “when,” but she could have.) We decompose her statement into two different features: the contact list and the contact editor. With the customer’s permission, we’re able to associate two user stories with the contact list: 1) “When a name is clicked, the associated contact is made current for editing”; and 2) “When the list of contact names changes, the names in the list are refreshed.” We can encode this language, first in a unit test, then into the presenter itself, as shown in figure 2.

We test our presenter in figure 2 first by constructing it with two mock objects (see the StickyNotes for more on mock objects), one each for the Contact List Model and Contact List View, and then by triggering events from the mocks and asserting that certain methods are called on the opposing object. Because it lacks a public interface, the only way to exercise the presenter’s behavior is by triggering the



```

class ContactListPresenter
  ...
  def setup
    @contact_list_model.when :contacts_changed do
      @contact_list_view.set_names(
        @contact_list_model.contact_names)
    end
    @contact_list_view.when :selection_changed do
      @contact_list_model.select_by_index(
        @contact_list_view.selected_index)
    end
  end
end
  
```

Figure 2: Contact list objects and presenter class—the presenter is constructed with model and view

events it has subscribed to: “when contacts changed” (from the model mock) and “when selection changed” (from the view mock). When it comes down to it, all our presenters really do is react to change events in one object by extracting data from it and passing it to the other. When the list changes, we pull the latest list of names out of our application model and push it into the view.

Unit testing and implementation of the model is straightforward, but as we develop its tests, we discover we’d like a method to inject contact data into the model. The question “Where is the data coming from?” has not been asked yet, but the simplest answer at this point has to be “Someone will hand it to us.” If we assume that much, we’ll avoid over-designing parts of our system that we’re not concerned with at this moment, keeping our focus on our customer’s story. The code below shows the methods required by the presenter, as well as the `set_contacts` method, which lets us test our model’s behavior to our satisfaction. We leave the problems of when `set_contacts` gets called in the deployed system—and who calls it—for later.

```

class ContactListModel
  can_fire :contacts_changed

  def initialize
    @contacts = []
  end

  def names
    @contacts.map { |c| c[:name] }
  end
end
  
```

```

def set_contacts(contacts)
  @contacts = contacts
  fire :contacts_changed
end

def select_by_index(index)
  # Awaiting more user stories to this effect
end
end
  
```

The view is equally straightforward:

```

class ContactListView < Fox::FXVerticalFrame
  ...
  can_fire :selection_changed

  def initialize(parent)
    ...
    @list = FXList.new(self);
    # Adapt the FXRuby List widget SEL_CHANGED event
    # to our own selection_changed event
    @list.connect(SEL_CHANGED) do |*ignored_args|
      fire :selection_changed
    end
  end

  def clear
    @list.clearItems(true)
  end

  def set_contact_names(names)
    names.each do |name|
      @list.appendItem(name)
    end
    @list.selectItem(0,true) unless names.empty?
  end

  def selected_index
    @list.currentItem
  end
end
  
```

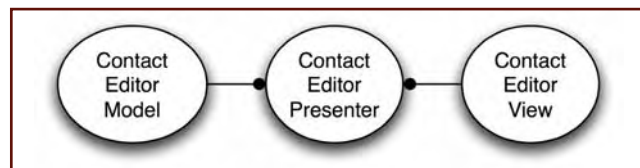
Notice that the model is completely separate from the view, having no knowledge of its interface or even its existence. The reverse is also true: The view knows nothing about the model. Each may be implemented independently of the other. Also, by making use of the Observer pattern, both objects may produce information without having to know what other objects will consume it. The presenter ensures that the consumers do not know where their information comes from, leaving them simply to operate on data as it is provided. All three classes—presenter, model, and view—have simple, easy-to-understand behaviors. This makes them easy to test.

Notice that our view is essentially a leaf node in the system;

it does not directly reference or utilize other objects in our collaboration scheme. This eliminates the common pitfall of coding too much smarts into the view by triggering complex logic in direct response to UI events. By forcing the view to hand off user events to our presenter, we keep it in a subordinate role, and it will never get a chance to unduly influence the rest of our design.

At this point we've taken our contact list as far as we can without some resolution from the customer with regard to the contact editor itself. At our next meeting, she shows us a sketch of how she wants to edit the details of her contacts. "When I select a contact from the list, I want the details for that contact to appear in the form. And when I click 'Save' at the bottom of the form, the updated contact info should be saved to the database."

At her mention of "the form," we're prompted to explore the nature of editing contact info a little further, and we generate an additional pair of finer-grained requirements: "When the user selects a new contact in the contact list view, repopulate the contact editor view" and "When the user saves an edited contact, apply those changes to whatever contact is selected in the contact list view." These "whens" outline the process of contact editing, which we can answer in the form of a new ContactEditorPresenter, along with its model and view (see figure 3).



```
class ContactEditorPresenter
  ...
  def setup
    @model.when :contact_info_changed do
      @view.set_name @model.contact_name
      @view.set_email @model.contact_email
    end
    @view.when :user_saved do
      @model.contact_name = @view.get_name
      @model.contact_email = @view.get_email
      @model.save
    end
  end
end
```

Figure 3: Contact editor objects in Presenter First triad

Without going into too much detail, suffice it to say the ContactEditorModel is not a contact data structure in and of itself but rather a layer of indirection that we can use to interact with the data of the currently selected contact.

We have two model/view/presenter triads that need to be coordinated—one for the contact list and another for the contact editor. For example, when an edited contact is to be

saved, the contact list model somehow has to get the changes so that it can update its list to use them.

Also, when a new contact is selected in the list, its data must appear in the contact editor view. One alternative is for the contact list MVP triad to be in control. The contact list model could shove the new contact into the contact editor model, like this:

```
class ContactListModel
  ...
  def select_by_index(index)
    @current_index = index
    @contact_editor_model.contact = @contacts[@current_index]
  end
end
```

If we try this, we'll run into two problems. First, we'll be unable to develop the two models independently. The symptom of this will be an annoying amount of setup code in our unit tests. While the presenter lets us develop the model and view independently, that decoupling hasn't carried over to any other code. To accomplish `select_by_index`, `ContactListModel` must know how to send contacts to the `ContactEditorModel`. (Eventually the list model will also have to react to changes in the editor model.)

Second, we've (perhaps unconsciously) decided to make `ContactListModel` responsible for knowing about *all* consumers of contact selection knowledge. We have to actively push the selected contact into those consumers. Each new consumer will require us to add code to the `ContactListModel`. Our tests are starting to look unnaturally heavy.

Amidst our growing dread, we notice one more aspect of the things we're trying to accomplish inside the `ContactListModel`. "When it's time to save the contact . . . When the contact becomes selected . . ." That's presenter-speak! But instead of connecting a model and a view, we're essentially linking two triplets, as in figure 4.

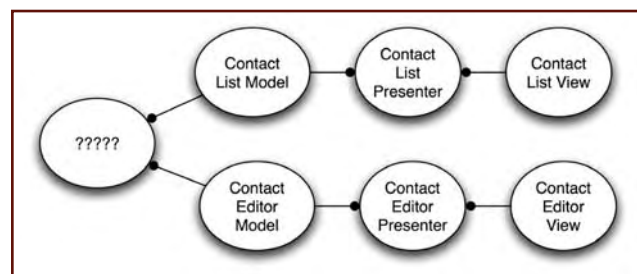
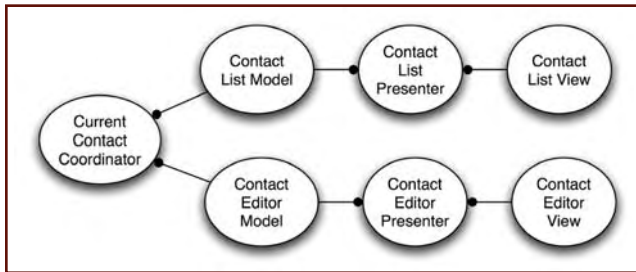


Figure 4: Integrating contact list and contact editor, and listing and editing features, with looser coupling

At the moment, we're at a loss for the name of that mystery object, but just by sketching it into our design we can start to see what it will do for us. The list model no longer has a reference to the editor model, so it doesn't need to know how to listen for save events or how to send the selected contact. Our new object—a presenter—will take that off our hands.

We name our mystery object for what it's going to do for us: It will coordinate the transfer of selected contact data in and out of our models, so it should be named `CurrentContactCoor-`

dinator. (There's no law that says a presenter has to be called a presenter—such objects are best identified by their roles in the composition of our system.)



```

class CurrentContactCoordinator
  ...
  def setup
    @contact_list_model.when :selection_changed do
      contact = @contact_list_model.selected_contact
    end

    @contact_editor_model.when :save_requested do
      @contact_list_model.save_contact(@contact_editor_model.contact)
    end
  end
end
end

```

Figure 5: Improved contact list and editor integration. Feature insulation is provided by CurrentContactCoordinator presenter object.

With their reduced responsibilities, the model implementations now look more like this:

```

class ContactListModel
  can_fire :selection_changed
  ...
  def select_by_index(index)
    @selected_index = index
    fire :selection_changed
  end
  def selected_contact
    @contacts[@selected_index]
  end
  def save_contact(contact)
    @contacts[@current_index] = contact
  end
end
end

```

This example demonstrates how we can extend the principle of a presenter beyond mere model/view integration into the realm of feature-level integration. Any time we find a grouping of objects getting bogged down with too much knowledge of each other, we can seek out the hidden “whens,” wrap them up in a presenter, and give ourselves some breathing room to write our tests. We’re empowered to keep our components small and narrow-purposed, opening them up to extension while leaving them mostly closed to change.

We’ve taken a few things for granted in the interest of brevity. For instance, the time you spend with the customer resolving user stories usually involves a lot of view prototyping, scribbling, and gesturing while you say “when” at each other. Depending on the customer it may be prudent to write some views first, in order to provide a playground in which to

further define the stories, and then adapt their interfaces to the subsequent presenters.

Similarly, coding the model side of your application isn’t directly addressed by Presenter First. You still can spend days deciphering legacy binary file formats, designing XML-processing pipelines, or building authentication mechanisms. Presenter First doesn’t tell you how to solve those problems; it just helps you know when to solve them and where. By keeping the focus on user stories you avoid the risk of building more of this infrastructure than you need.

While the technical impact on our code is satisfying in and of itself—smaller, testable, extensible, and de-coupled—the real magic here is our ability to continue responding to the voice of the customer. By understanding how to translate new and changing user requirements into these atomic triplets and by knowing how to loosely couple the triplets together, we can grow the system indefinitely, one story at a time, without first needing to establish an all-encompassing architecture. Presenter First is a simple technique that can be repeated as many times as needed to get the job done. The “micro-architecture” of Presenter First is like a carbohydrate chain in organic chemistry—interesting functionality arises

from applying the same basic pattern of object composition. The mechanisms and domain may vary, but the core principles do not: When the customer says “when,” write it into a presenter. If you can’t test something, refactor until you can. Develop infrastructure (models and views) only as demanded by presenters. **{end}**

David Crosby has been wrestling with fundamental questions of design and testing since joining Atomic Object in 2001. After graduating with a BS in computer science from Grand Valley State University, he worked at Lockheed Martin. As the first Atomic employee recognized as Software Craftsman, David mentors and yells instructively about testability and process. David can be reached at david@atomicobject.com.

Carl Erickson is the president of Atomic Object, which practices software craftsmanship on contract for customers in automotive testing, color measurement, materials handling, aerospace, healthcare, financial services, and e-commerce. Having a PhD in computer engineering and teaching a course on software craftsmanship at Uppsala University in Uppsala, Sweden, does not make Carl immune to Dave’s yelling.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware.

- Presenter First
- FXRuby
- Mock objects