



38

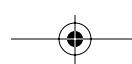
The Payroll User Interface: MODEL VIEW PRESENTER

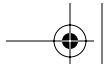


As far as the customer is concerned, the Interface is the product.

—Jef Raskin

Our payroll application is coming together nicely at this point. It supports adding hourly, salary, and commissioned employees. Each employee's payments may be delivered by mail, direct deposit, or held in the office. The system can calculate the pay for each





employee and have it delivered on a variety of schedules. Furthermore, all the data created and used by the system is persisted in a relational database.

In its current state, the system supports all the needs of our customer. In fact, it was put into production last week. It was installed on a computer in the Human Resources department, and Joe was trained to use it. Joe receives companywide requests to add new employees or change existing employees. He enters each request by adding the appropriate transaction text to a text file that is processed every night. Joe has been very grumpy recently, but he became very happy when he heard that we're going to build a user interface for the payroll system. This UI should make the payroll system easier to use. Joe is happy about this because everyone will be able to enter his or her own transactions instead of sending them to Joe to type into the transaction file.

Deciding what type of interface to build required a long discussion with the customer. One option proposed was a text-based interface whereby users would traverse menus, using keystrokes and entering data via the keyboard. Although text interfaces are easy to build, they can be less than easy to use. Besides, most users today consider them to be "clunky."

A Web interface was also considered. Web applications are great because they don't usually require any installation on the user's machines and can be used from any computer connected to the office intranet. But building Web interfaces is complicated because they appear to tie the application to a large and complex infrastructure of Web servers, application servers, and tiered architectures.¹ This infrastructure needs to be purchased, installed, configured, and administered. Web systems also tie us to such technologies as HTML, CSS, and JavaScript and force us into a somewhat stilted user model reminiscent of the 3270 green-screen applications of the 1970s.

Our users, and our company, wanted something simple to use, build, install, and administer. So in the end, we opted for a GUI desktop application. GUI desktop applications provide a more powerful set of UI functionality and can be less complicated to build than a Web interface. Our initial implementation won't be deployed over a network, so we won't need any of the complex infrastructure that Web systems seem to require.

Of course, desktop GUI applications have some disadvantages. They are not portable and are not easily distributed. However, since all the users of the payroll system work in the same office and use company computers, it was agreed that these disadvantages don't cost us as much as the Web architecture would. So we decided to use Windows Forms to build our UI.

Since UIs can be tricky, we'll limit our first release to adding employees. This first small release will give us some valuable feedback. First, we'll find out how complicated the UI is to build. Second, Joe will use the new UI and will tell us how much easier life is—we hope. Armed with this information, we will know better how to proceed to build

1. Or so it seems to the unwary software architect. In many unfortunate cases, this extra infrastructure provides much more benefit to the vendors than to the users.



the rest of the UI. It is also possible that the feedback from this first small release might suggest that a text-based or Web interface would be better. If that happens, it would be better to know before we invested effort in the whole application.

The form of the UI is less important than the internal architecture. Whether desktop or Web, UIs are usually volatile and tend to change more often than the business rules beneath them. Thus, it will behoove us to carefully separate the business logic from the user interface. Toward that end, we'll write as little code in the Windows Forms as possible. Instead, we'll put the code in plain C# classes that will work together with the Windows Forms. This separation strategy protects the business rules from the volatility of the UI. Changes to the UI code won't affect the business rules. Moreover, if one day we decide to switch to a Web interface, the business rule code will already have been separated.

The Interface

Figure 38-1 shows the general idea for the UI that we'll build. The menu named Action contains a list of all the supported actions. Selecting an action opens an appropriate form for creating the selected action. For example, Figure 38-2 shows the form that appears when Add Employee is selected. For the time being, Add Employee is the only action we're interested in.

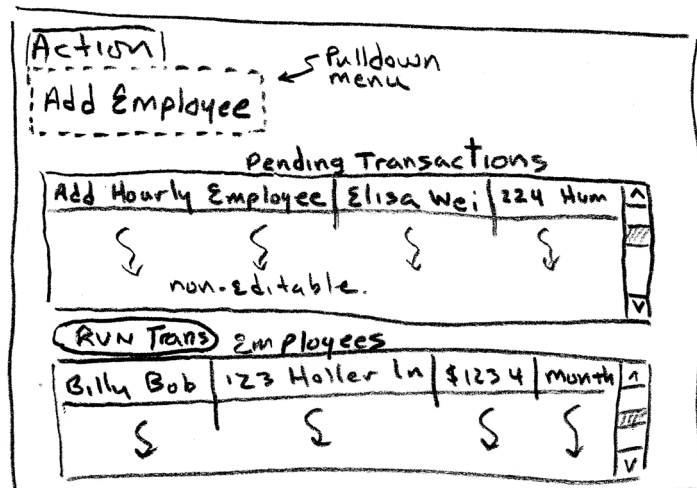


Figure 38-1
Initial payroll user interface



Figure 38-2
Add Employee transaction form

Near the top of the Payroll window is a text box labeled Pending Transactions. Payroll is a batch system. Transactions are entered throughout the day but are not executed until night, when they are all executed together as a batch. This top text box is a list of all the pending transaction that have been collected but not yet executed. In Figure 38-1 we can see that there is one pending transaction to add an hourly employee. The format of this list is readable, but we'll probably want to make it look prettier down the road. For now, this should do.

The bottom text box is labeled Employees and contains a list of employees who already exist in the system. Executing `AddEmployeeTransactions` will add more employees to this list. Again, we can imagine a much better way to display the employees. A tabular format would be nice. There could be a column for each bit a data, along with a column for the date of the last paycheck, amount paid to date, and so on. Records for hourly and commissioned employees would include a link to a new window that would list their time cards and sales receipts, respectively. That will have to wait, though.

In the middle is a button labeled Run Transactions, which does just as it suggests. Clicking it will invoke the batch, executing all the pending transactions and updating the employee list. Unfortunately, someone will have to click this button to initiate the batch. This is a temporary solution until we create an automatic schedule to do it.

Implementation

We can't get very far with the payroll window without being able to add transactions, so we'll start with the form to add an employee transaction, shown in Figure 38-2. Let's think



about the business rules that have to be achieved through this window. We need to collect all the information to create a transaction. This can be achieved as the user fills out the form. Based on the information, we need to figure out what type of `AddEmployeeTransaction` to create and then put that transaction in the list to be processed later. This will all be triggered by a click of the Submit button.

That about covers it for the business rules, but we need other behaviors to make the UI more usable. The Submit button, for example, should remain disabled until all the required information is supplied. Also, the text box for hourly rate should be disabled unless the Hourly radio button is on. Likewise, the Salary, Base Salary, and Commission text boxes should remain disabled until the appropriate radio button is clicked.

We must be careful to separate the business behavior from the UI behavior. To do so, we'll use the MODEL VIEW PRESENTER design pattern. Figure 38-3 shows a UML design for how we'll use MODEL VIEW PRESENTER for our current task. You can see that the design has three components: the model, the view, and the presenter. The model in this case represents the `AddEmployeeTransaction` class and its derivatives. The view is a Windows Form called `AddEmployeeWindow`, shown in Figure 38-2. The presenter, a class called `AddEmployeePresenter`, glues the UI to the model. `AddEmployeePresenter` contains all the business logic for this particular part of the application, whereas `AddEmployeeWindow` contains none. Instead, `AddEmployeeWindow` confines itself to the UI behavior, delegating all the business decisions to the presenter.

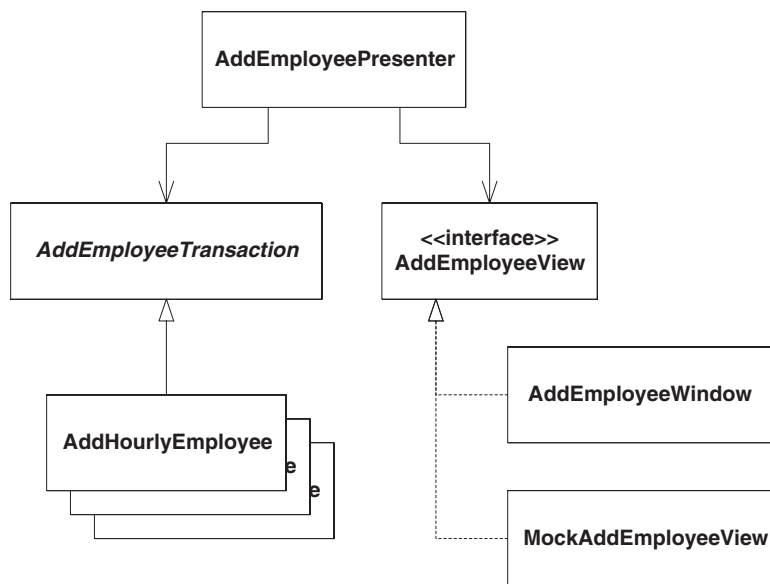


Figure 38-3
MODEL VIEW PRESENTER pattern for adding an employee transaction



An alternative to using MODEL VIEW PRESENTER is to push all the business logic into the Windows Form. In fact, this approach is very common but quite problematic. When the business rules are embedded in the UI code, not only do you have an SRP violation, but also the business rules are much more difficult to automatically test. Such tests would have to involve clicking buttons, reading labels, selecting items in a combo box, and fidgeting with other types of controls. In other words, in order to test the business rules, we'd have to actually *use* the UI. Tests that use the UI are fragile because minor changes to the UI controls have a large impact on the tests. They're also tricky because getting UIs in a test harness is a challenge in and of itself. Also, later down the road, we may decide that a Web interface is needed, and business logic embedded in the Windows form code would have to be duplicated in the ASP.NET code.

The observant reader will have noticed that the `AddEmployeePresenter` does not directly depend on the `AddEmployeeWindow`. The `AddEmployeeView` interface inverts the dependency. Why? Most simply, to make testing easier. Getting user interfaces under test is challenging. If `AddEmployeePresenter` was directly dependent upon `AddEmployeeWindow`, `AddEmployeePresenterTest` would also have to depend on `AddEmployeeWindow`, and that would be unfortunate. Using the interface and `MockAddEmployeeView` greatly simplifies testing.

Listing 38-1 and Listing 38-2 show the `AddEmployeePresenterTest` and `AddEmployeePresenter`, respectively. This is where the story begins.

Listing 38-1**AddEmployeePresenterTest.cs**

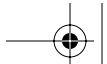
```
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
    [TestFixture]
    public class AddEmployeePresenterTest
    {
        private AddEmployeePresenter presenter;
        private TransactionContainer container;
        private InMemoryPayrollDatabase database;
        private MockAddEmployeeView view;

        [SetUp]
        public void SetUp()
        {
            view = new MockAddEmployeeView();
            container = new TransactionContainer(null);
            database = new InMemoryPayrollDatabase();
            presenter = new AddEmployeePresenter(
                view, container, database);
        }

        [Test]
        public void Creation()
```



**Listing 38-1 (Continued)****AddEmployeePresenterTest.cs**

```
{
    Assert.AreSame(container,
        presenter.TransactionContainer);
}

[Test]
public void AllInfoIsCollected()
{
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.EmpId = 1;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.Name = "Bill";
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.Address = "123 abc";
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.IsHourly = true;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.HourlyRate = 1.23;
    Assert.IsTrue(presenter.AllInformationIsCollected());

    presenter.IsHourly = false;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.IsSalary = true;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.Salary = 1234;
    Assert.IsTrue(presenter.AllInformationIsCollected());

    presenter.IsSalary = false;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.IsCommission = true;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.CommissionSalary = 123;
    Assert.IsFalse(presenter.AllInformationIsCollected());
    presenter.Commission = 12;
    Assert.IsTrue(presenter.AllInformationIsCollected());
}

[Test]
public void ViewGetsUpdated()
{
    presenter.EmpId = 1;
    CheckSubmitEnabled(false, 1);

    presenter.Name = "Bill";
    CheckSubmitEnabled(false, 2);

    presenter.Address = "123 abc";
    CheckSubmitEnabled(false, 3);

    presenter.IsHourly = true;
    CheckSubmitEnabled(false, 4);

    presenter.HourlyRate = 1.23;
    CheckSubmitEnabled(true, 5);
}
```



**Listing 38-1 (Continued)****AddEmployeePresenterTest.cs**

```
private void CheckSubmitEnabled(bool expected, int count)
{
    Assert.AreEqual(expected, view.submitEnabled);
    Assert.AreEqual(count, view.submitEnabledCount);
    view.submitEnabled = false;
}

[Test]
public void CreatingTransaction()
{
    presenter.EmpId = 123;
    presenter.Name = "Joe";
    presenter.Address = "314 Elm";

    presenter.IsHourly = true;
    presenter.HourlyRate = 10;
    Assert.IsTrue(presenter.CreateTransaction()
        is AddHourlyEmployee);

    presenter.IsHourly = false;
    presenter.IsSalary = true;
    presenter.Salary = 3000;
    Assert.IsTrue(presenter.CreateTransaction()
        is AddSalariedEmployee);

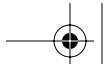
    presenter.IsSalary = false;
    presenter.IsCommission = true;
    presenter.CommissionSalary = 1000;
    presenter.Commission = 25;
    Assert.IsTrue(presenter.CreateTransaction()
        is AddCommissionedEmployee);
}

[Test]
public void AddEmployee()
{
    presenter.EmpId = 123;
    presenter.Name = "Joe";
    presenter.Address = "314 Elm";
    presenter.IsHourly = true;
    presenter.HourlyRate = 25;

    presenter.AddEmployee();

    Assert.AreEqual(1, container.Transactions.Count);
    Assert.IsTrue(container.Transactions[0]
        is AddHourlyEmployee);
}
}
```



**Listing 38-2****AddEmployeePresenter.cs**

```
using Payroll;

namespace PayrollUI
{
    public class AddEmployeePresenter
    {
        private TransactionContainer transactionContainer;
        private AddEmployeeView view;
        private PayrollDatabase database;

        private int empId;
        private string name;
        private string address;
        private bool isHourly;
        private double hourlyRate;
        private bool isSalary;
        private double salary;
        private bool isCommission;
        private double commissionSalary;
        private double commission;

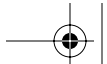
        public AddEmployeePresenter(AddEmployeeView view,
            TransactionContainer container,
            PayrollDatabase database)
        {
            this.view = view;
            this.transactionContainer = container;
            this.database = database;
        }

        public int EmpId
        {
            get { return empId; }
            set
            {
                empId = value;
                UpdateView();
            }
        }

        public string Name
        {
            get { return name; }
            set
            {
                name = value;
                UpdateView();
            }
        }

        public string Address
        {
            get { return address; }
        }
    }
}
```





Listing 38-2 (Continued)
AddEmployeePresenter.cs

```
        set
        {
            address = value;
            UpdateView();
        }
    }

    public bool IsHourly
    {
        get { return isHourly; }
        set
        {
            isHourly = value;
            UpdateView();
        }
    }

    public double HourlyRate
    {
        get { return hourlyRate; }
        set
        {
            hourlyRate = value;
            UpdateView();
        }
    }

    public bool IsSalary
    {
        get { return isSalary; }
        set
        {
            isSalary = value;
            UpdateView();
        }
    }

    public double Salary
    {
        get { return salary; }
        set
        {
            salary = value;
            UpdateView();
        }
    }

    public bool IsCommission
    {
        get { return isCommission; }
        set
        {
            isCommission = value;
            UpdateView();
        }
    }
}
```



**Listing 38-2 (Continued)****AddEmployeePresenter.cs**

```
}

public double CommissionSalary
{
    get { return commissionSalary; }
    set
    {
        commissionSalary = value;
        UpdateView();
    }
}

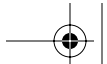
public double Commission
{
    get { return commission; }
    set
    {
        commission = value;
        UpdateView();
    }
}

private void UpdateView()
{
    if(AllInformationIsCollected())
        view.SubmitEnabled = true;
    else
        view.SubmitEnabled = false;
}

public bool AllInformationIsCollected()
{
    bool result = true;
    result &= empId > 0;
    result &= name != null && name.Length > 0;
    result &= address != null && address.Length > 0;
    result &= isHourly || isSalary || isCommission;
    if(isHourly)
        result &= hourlyRate > 0;
    else if(isSalary)
        result &= salary > 0;
    else if(isCommission)
    {
        result &= commission > 0;
        result &= commissionSalary > 0;
    }
    return result;
}

public TransactionContainer TransactionContainer
{
    get { return transactionContainer; }
}
```



**Listing 38-2 (Continued)****AddEmployeePresenter.cs**

```
public virtual void AddEmployee()
{
    transactionContainer.Add(CreateTransaction());
}

public Transaction CreateTransaction()
{
    if(isHourly)
        return new AddHourlyEmployee(
            empId, name, address, hourlyRate, database);
    else if(isSalary)
        return new AddSalariedEmployee(
            empId, name, address, salary, database);
    else
        return new AddCommissionedEmployee(
            empId, name, address, commissionSalary,
            commission, database);
}
}
```

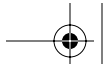
Starting with the `SetUp` method of the test, we see what's involved in instantiating the `AddEmployeePresenter`. It takes three parameters. The first is an `AddEmployeeView`, for which we use a `MockAddEmployeeView` in the test. The second is a `TransactionContainer` so that it has a place to put the `AddEmployeeTransaction` that it will create. The last parameter is a `PayrollDatabase` instance that won't be used directly but is needed as a parameter to the `AddEmployeeTransaction` constructors.

The first test, `Creation`, is almost embarrassingly silly. When first sitting down to write code, it can be difficult to figure out what to test first. It often helps to test the simplest thing you can think of. Doing so gets the ball rolling, and subsequent tests come much more naturally. The `Creation` test is an artifact of this practice. It makes sure that the container parameter was saved, and it could probably be deleted at this point.

The next test, `AllInfoIsCollected`, is much more interesting. One of the responsibilities of the `AddEmployeePresenter` is to collect all the information required to create a transaction. Partial data won't do, so the presenter has to know when *all* the necessary data has been collected. This test says that the presenter needs a method called `AllInformationIsCollected`, that returns a `boolean` value. The test also demonstrates how the presenter's data is entered through properties. Each piece of data is entered here one by one. At each step, the presenter is asked whether it has all the data it needs and asserts the expected response. In `AddEmployeePresenter`, we can see that each property simply stores the value in a field. `AllInformationIsCollected` does a bit of Boolean algebra as it checks that each field has been provided.

When the presenter has all the information it needs, the user may submit the data adding the transaction. But not until the presenter is content with the data provided should the user be able to submit the form. So it is the presenter's responsibility to inform the user





Chapter 38: The Payroll User Interface

649

when the form can be submitted. This is tested by the method `ViewGetsUpdated`. This test provides data, one piece at a time to the presenter. Each time, the test checks to make sure that the presenter properly informs the view whether submission should be enabled.

Looking in the presenter, we can see that each property makes a call to `UpdateView`, which in turn calls the `SaveEnabled` property on the view. Listing 38-3 shows the `AddEmployeeView` interface with `SubmitEnabled` declared. `AddEmployeePresenter` informs that submitting should be enabled by calling the `SubmitEnabled` property. Now we don't particularly care what `SubmitEnabled` does at this point. We simply want to make sure that it is called with the right value. This is where the `AddEmployeeView` interface comes in handy. It allows us to create a mock view to make testing easier. In `MockAddEmployeeView`, shown in Listing 38-4, there are two fields: `submitEnabled`, which records the last values passed in, and `submitEnabledCount`, which keeps track of how many times `SubmitEnabled` is invoked. These trivial fields make the test a breeze to write. All the test has to do is check the `submitEnabled` field to make sure that the presenter called the `SubmitEnabled` property with the right value and check `submitEnabledCount` to make sure that it was invoked the correct number of times. Imagine how awkward the test would have been if we had to dig into forms and window controls.

Listing 38-3

AddEmployeeView.cs

```
namespace PayrollUI
{
    public interface AddEmployeeView
    {
        bool SubmitEnabled { set; }
    }
}
```

Listing 38-4

MockAddEmployeeView.xs

```
namespace PayrollUI
{
    public class MockAddEmployeeView : AddEmployeeView
    {
        public bool submitEnabled;
        public int submitEnabledCount;

        public bool SubmitEnabled
        {
            set
            {
                submitEnabled = value;
                submitEnabledCount++;
            }
        }
    }
}
```



Something interesting happened in this test. We were careful to test how `AddEmployeePresenter` behaves when data is entered into the view rather than testing what happens when data is entered. In production, when all the data is entered, the Submit button will become enabled. We could have tested that; instead, we tested how the presenter behaves. We tested that when all the data is entered, the presenter will send a message to the view, telling it to enable submission.

This style of testing is called behavior-driven development. The idea is that you should not think of tests as tests, where you make assertions about state and results. Instead, you should think of tests as *specifications of behavior*, in which you describe how the code is supposed to behave.

The next test, `CreatingTransaction`, demonstrates that `AddEmployeePresenter` creates the proper transaction, based on the data provided. `AddEmployeePresenter` uses an `if/else` statement, based on the payment type, to figure out which type of transaction to create.

That leaves one more test, `AddEmployee`. When all the data is collected and the transaction is created, the presenter must save the transaction in the `TransactionContainer` so that it can be used later. This test makes sure that this happens.

With `AddEmployeePresenter` implemented, we have all the business rules in place to create `AddEmployeeTransactions`. Now all we need is user interface.

Building a Window

Designing the Add Employee window GUI code was a breeze. With Visual Studio's designer, it's simply a matter of dragging some controls around to the right place. This code is generated for us and is not included in the following listings. Once the window is designed, we have more work to do. We need to implement some behavior in the UI and wire it to the presenter. We also need a test for it all. Listing 38-5 shows `AddEmployeeWindowTest`, and Listing 38-6 shows `AddEmployeeWindow`.

Listing 38-5

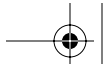
`AddEmployeeWindowTest.cs`

```
using NUnit.Framework;

namespace PayrollUI
{
    [TestFixture]
    public class AddEmployeeWindowTest
    {
        private AddEmployeeWindow window;
        private AddEmployeePresenter presenter;
        private TransactionContainer transactionContainer;

        [SetUp]
        public void SetUp()
        {
            window = new AddEmployeeWindow();
```



**Listing 38-5 (Continued)****AddEmployeeWindowTest.cs**

```
        transactionContainer = new TransactionContainer(null);
        presenter = new AddEmployeePresenter(
            window, transactionContainer, null);

        window.Presenter = presenter;
        window.Show();
    }

    [Test]
    public void StartingState()
    {
        Assert.AreSame(presenter, window.Presenter);
        Assert.IsFalse(window.submitButton.Enabled);
        Assert.IsFalse(window.hourlyRateTextBox.Enabled);
        Assert.IsFalse(window.salaryTextBox.Enabled);
        Assert.IsFalse(window.commissionSalaryTextBox.Enabled);
        Assert.IsFalse(window.commissionTextBox.Enabled);
    }

    [Test]
    public void PresenterValuesAreSet()
    {
        window.empIdTextBox.Text = "123";
        Assert.AreEqual(123, presenter.EmpId);

        window.nameTextBox.Text = "John";
        Assert.AreEqual("John", presenter.Name);

        window.addressTextBox.Text = "321 Somewhere";
        Assert.AreEqual("321 Somewhere", presenter.Address);

        window.hourlyRateTextBox.Text = "123.45";
        Assert.AreEqual(123.45, presenter.HourlyRate, 0.01);

        window.salaryTextBox.Text = "1234";
        Assert.AreEqual(1234, presenter.Salary, 0.01);

        window.commissionSalaryTextBox.Text = "123";
        Assert.AreEqual(123, presenter.CommissionSalary, 0.01);

        window.commissionTextBox.Text = "12.3";
        Assert.AreEqual(12.3, presenter.Commission, 0.01);

        window.hourlyRadioButton.PerformClick();
        Assert.IsTrue(presenter.IsHourly);

        window.salaryRadioButton.PerformClick();
        Assert.IsTrue(presenter.IsSalary);
        Assert.IsFalse(presenter.IsHourly);

        window.commissionRadioButton.PerformClick();
        Assert.IsTrue(presenter.IsCommission);
        Assert.IsFalse(presenter.IsSalary);
    }
}
```



**Listing 38-5 (Continued)****AddEmployeeWindowTest.cs**

```
[Test]
public void EnablingHourlyFields()
{
    window.hourlyRadioButton.Checked = true;
    Assert.IsTrue(window.hourlyRateTextBox.Enabled);

    window.hourlyRadioButton.Checked = false;
    Assert.IsFalse(window.hourlyRateTextBox.Enabled);
}

[Test]
public void EnablingSalaryFields()
{
    window.salaryRadioButton.Checked = true;
    Assert.IsTrue(window.salaryTextBox.Enabled);

    window.salaryRadioButton.Checked = false;
    Assert.IsFalse(window.salaryTextBox.Enabled);
}

[Test]
public void EnablingCommissionFields()
{
    window.commissionRadioButton.Checked = true;
    Assert.IsTrue(window.commissionTextBox.Enabled);
    Assert.IsTrue(window.commissionSalaryTextBox.Enabled);

    window.commissionRadioButton.Checked = false;
    Assert.IsFalse(window.commissionTextBox.Enabled);
    Assert.IsFalse(window.commissionSalaryTextBox.Enabled);
}

[Test]
public void EnablingAddEmployeeButton()
{
    Assert.IsFalse(window.submitButton.Enabled);

    window.SubmitEnabled = true;
    Assert.IsTrue(window.submitButton.Enabled);

    window.SubmitEnabled = false;
    Assert.IsFalse(window.submitButton.Enabled);
}

[Test]
public void AddEmployee()
{
    window.empIdTextBox.Text = "123";
    window.nameTextBox.Text = "John";
    window.addressTextBox.Text = "321 Somewhere";
    window.hourlyRadioButton.Checked = true;
    window.hourlyRateTextBox.Text = "123.45";

    window.submitButton.PerformClick();
}
```



**Listing 38-5 (Continued)****AddEmployeeWindowTest.cs**

```
Assert.IsFalse(window.Visible);
Assert.AreEqual(1,
    transactionContainer.Transactions.Count);
}
}
```

Listing 38-6**AddEmployeeWindow.cs**

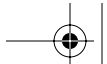
```
using System;
using System.Windows.Forms;

namespace PayrollUI
{
    public class AddEmployeeWindow : Form, AddEmployeeView
    {
        public System.Windows.Forms.TextBox empIdTextBox;
        private System.Windows.Forms.Label empIdLabel;
        private System.Windows.Forms.Label nameLabel;
        public System.Windows.Forms.TextBox nameTextBox;
        private System.Windows.Forms.Label addressLabel;
        public System.Windows.Forms.TextBox addressTextBox;
        public System.Windows.Forms.RadioButton hourlyRadioButton;
        public System.Windows.Forms.RadioButton salaryRadioButton;
        public System.Windows.Forms.RadioButton commissionRadioButton;
        private System.Windows.Forms.Label hourlyRateLabel;
        public System.Windows.Forms.TextBox hourlyRateTextBox;
        private System.Windows.Forms.Label salaryLabel;
        public System.Windows.Forms.TextBox salaryTextBox;
        private System.Windows.Forms.Label commissionSalaryLabel;
        public System.Windows.Forms.TextBox commissionSalaryTextBox;
        private System.Windows.Forms.Label commissionLabel;
        public System.Windows.Forms.TextBox commissionTextBox;
        private System.Windows.Forms.TextBox textBox2;
        private System.Windows.Forms.Label label1;
        private System.ComponentModel.IContainer components = null;
        public System.Windows.Forms.Button submitButton;
        private AddEmployeePresenter presenter;

        public AddEmployeeWindow()
        {
            InitializeComponent();
        }

        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if(components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }
    }
}
```



**Listing 38-6 (Continued)****AddEmployeeWindow.cs**

```
}

#region Windows Form Designer generated code
// snip
#endregion

public AddEmployeePresenter Presenter
{
    get { return presenter; }
    set { presenter = value; }
}

private void hourlyRadioButton_CheckedChanged(
    object sender, System.EventArgs e)
{
    hourlyRateTextBox.Enabled = hourlyRadioButton.Checked;
    presenter.IsHourly = hourlyRadioButton.Checked;
}

private void salaryRadioButton_CheckedChanged(
    object sender, System.EventArgs e)
{
    salaryTextBox.Enabled = salaryRadioButton.Checked;
    presenter.IsSalary = salaryRadioButton.Checked;
}

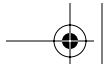
private void commissionRadioButton_CheckedChanged(
    object sender, System.EventArgs e)
{
    commissionSalaryTextBox.Enabled =
        commissionRadioButton.Checked;
    commissionTextBox.Enabled =
        commissionRadioButton.Checked;
    presenter.IsCommission =
        commissionRadioButton.Checked;
}

private void empIdTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.EmpId = AsInt(empIdTextBox.Text);
}

private void nameTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.Name = nameTextBox.Text;
}

private void addressTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.Address = addressTextBox.Text;
}
}
```



**Listing 38-6 (Continued)****AddEmployeeWindow.cs**

```
private void hourlyRateTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.HourlyRate = AsDouble(hourlyRateTextBox.Text);
}

private void salaryTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.Salary = AsDouble(salaryTextBox.Text);
}

private void commissionSalaryTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.CommissionSalary =
        AsDouble(commissionSalaryTextBox.Text);
}

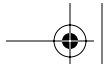
private void commissionTextBox_TextChanged(
    object sender, System.EventArgs e)
{
    presenter.Commission = AsDouble(commissionTextBox.Text);
}

private void addEmployeeButton_Click(
    object sender, System.EventArgs e)
{
    presenter.AddEmployee();
    this.Close();
}

private double AsDouble(string text)
{
    try
    {
        return Double.Parse(text);
    }
    catch (Exception)
    {
        return 0.0;
    }
}

private int AsInt(string text)
{
    try
    {
        return Int32.Parse(text);
    }
    catch (Exception)
    {
        return 0;
    }
}
```



**Listing 38-6 (Continued)****AddEmployeeWindow.cs**

```
public bool SubmitEnabled
{
    set { submitButton.Enabled = value; }
}
}
```

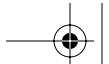
Despite all my griping about how painful it is to test GUI code, testing Windows Form code is relatively easy. There are some pitfalls, however. For some silly reason, known only to programmers at Microsoft, half of the functionality of the controls does not work unless they are *displayed* on the screen. It is for this reason that you'll find the call `window.Show()` in the `SetUp` of the test fixture. When the tests are executed, you can see the window appearing and quickly disappearing for each test. This is annoying but bearable. Anything that slows down the tests or otherwise makes them clumsy makes it more likely that the tests will not be run.

Another limitation is that you cannot easily invoke all the events on a control. With buttons and buttonlike controls, you can call `PerformClick`, but events such as `MouseOver`, `Leave`, `Validate`, and others are not so easy. An extension for NUnit, called `NUnitForms`, can help with these problems and more. Our tests are simple enough to get by without extra help.

In the `SetUp` of our test, we create an instance of `AddEmployeeWindow` and give it an instance of `AddEmployeePresenter`. Then in the first test, `StartingState`, we make sure that several controls are disabled: `hourlyRateTextBox`, `salaryTextBox`, `commissionSalaryTextBox`, and `commissionTextBox`. Only one or two of these fields are needed, and we don't know which ones until the user chooses the payment type. To avoid confusing the user by leaving all the fields enabled, they'll remain disabled until needed. The rules for enabling these controls are specified in three tests: `EnablingHourlyFields`, `EnablingSalaryField`, and `EnablingCommissionFields`. `EnablingHourlyFields`, for example, demonstrates how the `hourlyRateTextBox` is enabled when the `hourlyRadioButton` is turned on and disabled when the radio button is turned off. This is achieved by registering an `EventHandler` with each `RadioButton`. Each `EventHandler` enables and disables the appropriate text boxes.

The test, `PresenterValuesAreSet`, is an important one. The presenter knows what to do with the data, but it's the view's responsibility to populate the data. Therefore, whenever a field in the view is changed, it calls the corresponding property on the presenter. For each `TextBox` in the form, we use the `Text` property to change the value and then check to make sure that the presenter is properly updated. In `AddEmployeeWindow`, each `TextBox` has an `EventHandler` registered on the `TextChanged` event. For the `RadioButton` controls, we call the `PerformClick` method in the test and again make sure that the presenter is informed. The `RadioButton`'s `EventHandlers` take care of this.





Chapter 38: The Payroll User Interface

657

`EnablingAddEmployeeButton` specifies how the `submitButton` is enabled when the `SubmitEnabled` property is set to `true`, and reverse. Remember that in `AddEmployeePresenterTest`, we didn't care what this property did. Now we do care. The view must respond properly when the `SubmitEnabled` property is changed; however, `AddEmployeePresenterTest` was not the right place to test it. `AddEmployeeWindowTest` focuses on the behavior of the `AddEmployeeWindow`, and it *is* the right place to test this unit of code.

The final test here is `AddEmployee`, which fills in a valid set of fields, clicks the `Submit` button, asserts that the window is no longer visible, and makes sure that a transaction was added to the `transactionContainer`. To make this pass, we register an `EventHandler`, on the `submitButton`, that calls `AddEmployee` on the presenter and then closes the window. If you think about it, the test is doing a lot of work just to make sure that the `AddEmployee` method was called. It has to populate all the fields and then check the `transactionContainer`. Some might argue that instead, we should use a mock presenter so we can easily check that the method was called. To be honest, I wouldn't put up a fight if my pair partner were to bring it up. But the current implementation doesn't bother me too much. It's healthy to include a few high-level tests like this. They help to make sure that the pieces can be integrated properly and that the system works the way it should when put together. Normally, we'd have a suite of acceptance tests that do this at an even higher level, but it doesn't hurt to do it a bit in the unit tests—just a bit, though.

With this code in place, we now have a working form for creating `AddEmployeeTransactions`. But it won't get used until we have the main `Payroll` window working and wired up to load our `AddEmployeeWindow`.

The Payroll Window

In building the `Payroll` view, shown in Figure 38-4, we'll use the same `MODEL VIEW PRESENTER` pattern used in the `Add Employee` view.

Listings 38-7 and 38-8 show all the code in this part of the design. Altogether, the development of this view is very similar to that of the `Add Employee` view. For that reason, we will not pay it much attention. Of particular note, however, is the `ViewLoader` hierarchy.

Sooner or later in developing this window, we'll get around to implementing an `EventHandler` for the `Add Employee MenuItem`. This `EventHandler` will call the `AddEmployeeActionInvoked` method on `PayrollPresenter`. At this point, the `AddEmployeeWindow` needs to pop up. Is `PayrollPresenter` supposed to instantiate `AddEmployeeWindow`? So far, we have done well to decouple the UI from the application. Were it to instantiate the `AddEmployeeWindow`, `PayrollPresenter` would be violating `DIP`. Yet someone must create `AddEmployeeWindow`.



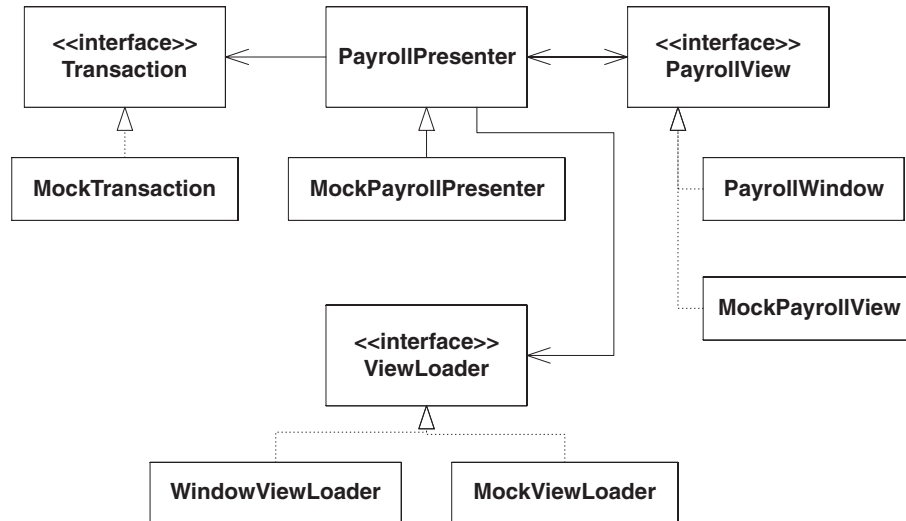
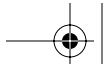


Figure 38-4
Design of the Payroll view

FACTORY pattern to the rescue! This is the exact problem that FACTORY was designed to solve. `ViewLoader`, and its derivatives, are in fact an implementation of the FACTORY pattern. It declares two methods: `LoadPayrollView` and `LoadAddEmployeeView`. `WindowsViewLoader` implements these methods to create Windows Forms and display them. The `MockViewLoader`, which can easily replace the `WindowsViewLoader`, makes testing much easier.

With the `ViewLoader` in place, `PayrollPresenter` need not depend on any Windows form classes. It simply makes a call to the `LoadAddEmployeeView` on its reference to `ViewLoader`. If the need ever arises, we can change the whole user interface for Payroll by swapping the `ViewLoader` implementation. No code needs to change. That's power! That's OCP!

Listing 38-7

PayrollPresenterTest.cs

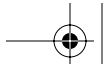
```

using System;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
    [TestFixture]
    public class PayrollPresenterTest
    {
        private MockPayrollView view;
        private PayrollPresenter presenter;
        private PayrollDatabase database;
    }
}

```



**Listing 38-7 (Continued)****PayrollPresenterTest.cs**

```
private MockViewLoader viewLoader;

[SetUp]
public void SetUp()
{
    view = new MockPayrollView();
    database = new InMemoryPayrollDatabase();
    viewLoader = new MockViewLoader();
    presenter = new PayrollPresenter(database, viewLoader);
    presenter.View = view;
}

[Test]
public void Creation()
{
    Assert.AreSame(view, presenter.View);
    Assert.AreSame(database, presenter.Database);
    Assert.IsNotNull(presenter.TransactionContainer);
}

[Test]
public void AddAction()
{
    TransactionContainer container =
        presenter.TransactionContainer;
    Transaction transaction = new MockTransaction();

    container.Add(transaction);

    string expected = transaction.ToString()
        + Environment.NewLine;
    Assert.AreEqual(expected, view.transactionsText);
}

[Test]
public void AddEmployeeAction()
{
    presenter.AddEmployeeActionInvoked();

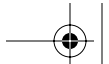
    Assert.IsTrue(viewLoader.addEmployeeViewWasLoaded);
}

[Test]
public void RunTransactions()
{
    MockTransaction transaction = new MockTransaction();
    presenter.TransactionContainer.Add(transaction);
    Employee employee =
        new Employee(123, "John", "123 Baker St.");
    database.AddEmployee(employee);

    presenter.RunTransactions();

    Assert.IsTrue(transaction.wasExecuted);
}
```



**Listing 38-7 (Continued)****PayrollPresenterTest.cs**

```
Assert.AreEqual("", view.transactionsText);
string expectedEmployeeTest = employee.ToString()
    + Environment.NewLine;
Assert.AreEqual(expectedEmployeeTest, view.employeesText);
    }
}
}
```

Listing 38-8**PayrollPresenter.cs**

```
using System;
using System.Text;
using Payroll;

namespace PayrollUI
{
    public class PayrollPresenter
    {
        private PayrollView view;
        private readonly PayrollDatabase database;
        private readonly ViewLoader viewLoader;
        private TransactionContainer transactionContainer;

        public PayrollPresenter(PayrollDatabase database,
            ViewLoader viewLoader)
        {
            this.view = view;
            this.database = database;
            this.viewLoader = viewLoader;
            transactionContainer.AddAction addAction =
                new TransactionContainer.AddAction(TransactionAdded);
            transactionContainer = new TransactionContainer(addAction);
        }

        public PayrollView View
        {
            get { return view; }
            set { view = value; }
        }

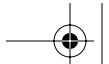
        public TransactionContainer TransactionContainer
        {
            get { return transactionContainer; }
        }

        public void TransactionAdded()
        {
            UpdateTransactionsTextBox();
        }

        private void UpdateTransactionsTextBox()
        {
            StringBuilder builder = new StringBuilder();

```



**Listing 38-8 (Continued)****PayrollPresenter.cs**

```
        foreach(Transaction transaction in
            transactionContainer.Transactions)
        {
            builder.Append(transaction.ToString());
            builder.Append(Environment.NewLine);
        }
        view.TransactionsText = builder.ToString();
    }

    public PayrollDatabase Database
    {
        get { return database; }
    }

    public virtual void AddEmployeeActionInvoked()
    {
        viewLoader.LoadAddEmployeeView(transactionContainer);
    }

    public virtual void RunTransactions()
    {
        foreach(Transaction transaction in
            transactionContainer.Transactions)
            transaction.Execute();

        transactionContainer.Clear();
        UpdateTransactionsTextBox();
        UpdateEmployeesTextBox();
    }

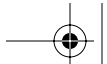
    private void UpdateEmployeesTextBox()
    {
        StringBuilder builder = new StringBuilder();
        foreach(Employee employee in database.GetAllEmployees())
        {
            builder.Append(employee.ToString());
            builder.Append(Environment.NewLine);
        }
        view.EmployeesText = builder.ToString();
    }
}
```

Listing 38-9**PayrollView.cs**

```
namespace PayrollUI
{
    public interface PayrollView
    {
        string TransactionsText { set; }

        string EmployeesText { set; }
    }
}
```





Listing 38-9 (Continued)

PayrollView.cs

```
    PayrollPresenter presenter { set; }  
  }  
}
```

Listing 38-10

MockPayrollView.cs

```
namespace PayrollUI  
{  
    public class MockPayrollView : PayrollView  
    {  
        public string transactionsText;  
        public string employeesText;  
        public PayrollPresenter presenter;  
  
        public string TransactionsText  
        {  
            set { transactionsText = value; }  
        }  
  
        public string EmployeesText  
        {  
            set { employeesText = value; }  
        }  
  
        public PayrollPresenter presenter  
        {  
            set { presenter = value; }  
        }  
    }  
}
```

Listing 38-11

ViewLoader.cs

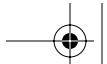
```
namespace PayrollUI  
{  
    public interface ViewLoader  
    {  
        void LoadPayrollView();  
        void LoadAddEmployeeView(  
            TransactionContainer transactionContainer);  
    }  
}
```

Listing 38-12

MockViewLoader.cs

```
namespace PayrollUI  
{  
    public class MockViewLoader : ViewLoader
```



**Listing 38-12 (Continued)****MockViewLoader.cs**

```
{
    public bool addEmployeeViewWasLoaded;
    private bool payrollViewWasLoaded;

    public void LoadPayrollView()
    {
        payrollViewWasLoaded = true;
    }

    public void LoadAddEmployeeView(
        TransactionContainer transactionContainer)
    {
        addEmployeeViewWasLoaded = true;
    }
}
```

Listing 38-13**WindowViewLoaderTest.cs**

```
using System.Windows.Forms;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
    [TestFixture]
    public class WindowViewLoaderTest
    {
        private PayrollDatabase database;
        private WindowViewLoader viewLoader;

        [SetUp]
        public void Setup()
        {
            database = new InMemoryPayrollDatabase();
            viewLoader = new WindowViewLoader(database);
        }

        [Test]
        public void LoadPayrollView()
        {
            viewLoader.LoadPayrollView();

            Form form = viewLoader.LastLoadedView;
            Assert.IsTrue(form is PayrollWindow);
            Assert.IsTrue(form.Visible);

            PayrollWindow payrollWindow = form as PayrollWindow;
            PayrollPresenter presenter = payrollWindow.Presenter;
            Assert.IsNotNull(presenter);
            Assert.AreSame(form, presenter.View);
        }
    }
}
```



**Listing 38-13 (Continued)****WindowViewLoaderTest.cs**

```
[Test]
public void LoadAddEmployeeView()
{
    viewLoader.LoadAddEmployeeView(
        new TransactionContainer(null));

    Form form = viewLoader.LastLoadedView;
    Assert.IsTrue(form is AddEmployeeWindow);
    Assert.IsTrue(form.Visible);

    AddEmployeeWindow addEmployeeWindow =
        form as AddEmployeeWindow;
    Assert.IsNotNull(addEmployeeWindow.Presenter);
}
}
```

Listing 38-14**WindowViewLoader.cs**

```
using System.Windows.Forms;
using Payroll;

namespace PayrollUI
{
    public class WindowViewLoader : ViewLoader
    {
        private readonly PayrollDatabase database;
        private Form lastLoadedView;

        public WindowViewLoader(PayrollDatabase database)
        {
            this.database = database;
        }

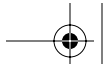
        public void LoadPayrollView()
        {
            PayrollWindow view = new PayrollWindow();
            PayrollPresenter presenter =
                new PayrollPresenter(database, this);

            view.Presenter = presenter;
            presenter.View = view;

            LoadView(view);
        }

        public void LoadAddEmployeeView(
            TransactionContainer transactionContainer)
        {
            AddEmployeeWindow view = new AddEmployeeWindow();
            AddEmployeePresenter presenter =
                new AddEmployeePresenter(view,
                    transactionContainer, database);
        }
    }
}
```



**Listing 38-14 (Continued)****WindowViewLoader.cs**

```
        view.Presenter = presenter;
        LoadView(view);
    }

    private void LoadView(Form view)
    {
        view.Show();
        lastLoadedView = view;
    }

    public Form LastLoadedView
    {
        get { return lastLoadedView; }
    }
}
```

Listing 38-15**PayrollWindowTest.cs**

```
using NUnit.Framework;

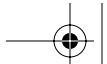
namespace PayrollUI
{
    [TestFixture]
    public class PayrollWindowTest
    {
        private PayrollWindow window;
        private MockPayrollPresenter presenter;

        [SetUp]
        public void SetUp()
        {
            window = new PayrollWindow();
            presenter = new MockPayrollPresenter();
            window.Presenter = this.presenter;
            window.Show();
        }

        [TearDown]
        public void TearDown()
        {
            window.Dispose();
        }

        [Test]
        public void TransactionsText()
        {
            window.TransactionsText = "abc 123";
            Assert.AreEqual("abc 123",
                window.transactionsTextBox.Text);
        }
    }
}
```



**Listing 38-15 (Continued)****PayrollWindowTest.cs**

```
[Test]
public void EmployeesText()
{
    window.EmployeesText = "some employee";
    Assert.AreEqual("some employee",
        window.employeesTextBox.Text);
}

[Test]
public void AddEmployeeAction()
{
    window.addEmployeeMenuItem.PerformClick();
    Assert.IsTrue(presenter.addEmployeeActionInvoked);
}

[Test]
public void RunTransactions()
{
    window.runButton.PerformClick();
    Assert.IsTrue(presenter.runTransactionCalled);
}
}
```

Listing 38-16**PayrollWinow.cs**

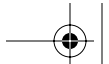
```
namespace PayrollUI
{
    public class PayrollWindow : System.Windows.Forms.Form,
        PayrollView
    {
        private System.Windows.Forms.MainMenu mainMenu1;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label employeeLabel;
        public System.Windows.Forms.TextBox employeesTextBox;
        public System.Windows.Forms.TextBox transactionsTextBox;
        public System.Windows.Forms.Button runButton;
        private System.ComponentModel.Container components = null;
        private System.Windows.Forms.MenuItem actionMenuItem;
        public System.Windows.Forms.MenuItem addEmployeeMenuItem;
        private PayrollPresenter presenter;

        public PayrollWindow()
        {
            InitializeComponent();
        }

        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if(components != null)
                {

```



**Listing 38-16 (Continued)****PayrollWinow.cs**

```
        components.Dispose();
    }
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code
//snip
#endregion

private void addEmployeeMenuItem_Click(
    object sender, System.EventArgs e)
{
    presenter.AddEmployeeActionInvoked();
}

private void runButton_Click(
    object sender, System.EventArgs e)
{
    presenter.RunTransactions();
}

public string TransactionsText
{
    set { transactionsTextBox.Text = value; }
}

public string EmployeesText
{
    set { employeesTextBox.Text = value; }
}

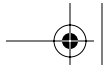
public PayrollPresenter Presenter
{
    get { return presenter; }
    set { presenter = value; }
}
}
}
```

Listing 38-17**TransactionContainerTest.cs**

```
using System.Collections;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
    [TestFixture]
    public class TransactionContainerTest
    {
        private TransactionContainer container;
    }
}
```





Listing 38-17 (Continued)

TransactionContainerTest.cs

```
private bool addActionCalled;
private Transaction transaction;

[SetUp]
public void Setup()
{
    TransactionContainer.AddAction action =
        new TransactionContainer.AddAction(SillyAddAction);
    container = new TransactionContainer(action);
    transaction = new MockTransaction();
}

[Test]
public void Construction()
{
    Assert.AreEqual(0, container.Transactions.Count);
}

[Test]
public void AddingTransaction()
{
    container.Add(transaction);

    IList transactions = container.Transactions;
    Assert.AreEqual(1, transactions.Count);
    Assert.AreSame(transaction, transactions[0]);
}

[Test]
public void AddingTransactionTriggersDelegate()
{
    container.Add(transaction);

    Assert.IsTrue(addActionCalled);
}

private void SillyAddAction()
{
    addActionCalled = true;
}
}
```

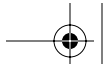
Listing 38-18

TransactionContainer.cs

```
using Payroll;

namespace PayrollUI
{
    public class TransactionContainer
    {
        public delegate void AddAction();
    }
}
```



**Listing 38-18 (Continued)****TransactionContainer.cs**

```
private IList transactions = new ArrayList();
private AddAction addAction;

public TransactionContainer(AddAction action)
{
    addAction = action;
}

public IList Transactions
{
    get { return transactions; }
}

public void Add(Transaction transaction)
{
    transactions.Add(transaction);
    if(addAction != null)
        addAction();
}

public void Clear()
{
    transactions.Clear();
}
}
```

The Unveiling

A lot of work has gone into this payroll application, and at last we'll see it come alive with its new graphical user interface. Listing 38-19 contains the `PayrollMain` class, the entry point for the application. Before we can load the Payroll view, we need an instance of the database. In this code listing, an `InMemoryPayrollDatabase` is being created. This is for demonstration purposes. In production, we'd create an `SqlPayrollDatabase` that would link up to our SQL Server database. But the application will happily run with the `InMemoryPayrollDatabase` while all the data is saved in memory and loaded in memory.

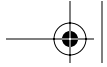
Next, an instance of `WindowViewerLoader` is created. `LoadPayrollView` is called, and the application is started. We can now compile, run it, and add as many employees to the system as we like.

Listing 38-19**PayrollMain.cs**

```
using System.Windows.Forms;
using Payroll;

namespace PayrollUI
```



**Listing 38-19 (Continued)****PayrollMain.cs**

```
{
    public class PayrollMain
    {
        public static void Main(string[] args)
        {
            PayrollDatabase database =
                new InMemoryPayrollDatabase();
            WindowViewer viewLoader =
                new WindowViewer(database);

            viewLoader.LoadPayrollView();
            Application.Run(viewLoader.LastLoadedView);
        }
    }
}
```

Conclusion

Joe will be happy to see what we've done for him. We'll build a production release and let him try it out. Surely he'll have comments about how the user interface is crude and unpolished. There will be quirks that slow him down or aspects that he finds confusing. User interfaces are difficult to get right. So we'll pay close attention to his feedback and go back for another round. Next, we'll add actions to change employee details. Then we'll add actions to submit time cards and sales receipts. Finally we'll handle payday. This is all left to you, of course.

Bibliography

<http://daveastels.com/index.php?p=5>

www.martinfowler.com/eaDev/ModelViewPresenter.html

<http://nunitforms.sourceforge.net/>

www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf

