PooleConsulting

When GUI Tests are Hard

And why that's a Good Thing!

Charlie Poole

- Software Development
  - City of Seattle, Microsoft, other NW firms
  - .Net, C#, J#, C++, COM, Win32
  - Independent developer, trainer, coach and consultant working in the US and Europe
- XP & Agile Methods
  - Specializing in TDD, XP and Scrum
  - Seattle XP Users Group – www.seattlexp.org
  - Open Source
    - NUnit – www.nunit.com/nunit
    - NUnitLite – www.nunit.com/nunitlite
    - VsUnit – www.codeplex.com/VSUnit

PooleConsulting

Outline

- Is GUI Testing Hard?
  - The GUI Message
- Principles of GUI Testing
  - Three Simple Rules
- Examples
  - Event Generation Example
  - Control Example
- Design Patterns

PooleConsulting

Is GUI Testing Hard?

What Makes GUI Testing Hard?

- Programmer tests aim at test independence

But…

- Testing connected sets of objects is hard
  - Tests become dependent on one another
  - Objects have to be put in a known state
  - It’s hard to figure out what failed

Problems of GUI Testing

- Tight coupling to other parts of the application – may include business logic
- UI is implemented as a single large program handling all user interaction
- Design is driven by IDE code generation
- Technical issues of the platform

Problems of GUI Testing

- Tight coupling to other parts of the application – may include business logic
- Possible Solutions
  - Separate business logic from the UI
  - Use interfaces to permit substitution
  - Use factories to create business objects

PooleConsulting

Problems of GUI Testing

- UI is implemented as a single large program handling all user interaction
- Possible Solutions
  - Use separate classes for UI logic
  - Use separate classes for validation
  - Use smart controls where feasible

Problems of GUI Testing

- Design is driven by IDE code generation
- Possible Solutions
  - Resist the temptation to insert code into the UI just because the IDE makes it easy.
  - Explore the limits of your tools
  - Where possible, split the code into two parts

Problems of GUI Testing

- Technical issues of the platform

- Possible solutions
  - Learn your platform and language thoroughly
  - Explore key aspects needed for testing UI
    - Event generation and capture
    - Instantiation of UI elements as part of a test
    - Access to contained UI elements
  - Be clear on what testing is needed

PooleConsulting

Test-Driven Development in .NET

The GUI Message

The GUI Message

GUI testing may be hard, but it’s no harder than many other aspects of delivering great software.

The GUI Message

What initially presents as a testing problem usually turns out to be a design problem.

The GUI Message

The design techniques we use to solve GUI testing problems are the same ones we use for business objects.

The GUI Message

GUI testing presents unique technical – as opposed to conceptual – issues.

The GUI Message

The general principles used for GUI testing are platform-independent.

The GUI Message

Some of the lower-level techniques require platform-specific knowledge, even though similar things must be accomplished on every platform.

The “Hardness” Paradox

- Perhaps it’s a good thing that GUIs seem hard to test
  - Most of the problems that make GUI testing “hard” are problems of design
  - “Hardness” can be a signal to look more deeply at how our application is designed
- Most GUIs seem to need design improvement
  - The influence of various IDEs and other vendor tools is the biggest culprit.

An Objection?

Should we be modifying the design merely to make the software easier to test?

Yes, because…

- Testability is just as valid an objective as security, safety, ease of deployment or any other desirable characteristic
- Poor testability is almost always due to other problems in the design, particularly poor separation of responsibilities, and excessive coupling between components.

The GUI Message

Questions?

PooleConsulting

Test-Driven Development in .NET

Principles of GUI Testing

Principles

- Know exactly what you’re testing
  - Each test should have one purpose
  - Do you really need to test it?
    - Too simple to fail
    - Part of the platform

Principles

- Keep domain logic out of the GUI
  - Use separate domain objects
  - Test them separately, without the GUI
  - Make the GUI as thin as possible
  - Consider NOT testing it!

Principles

- Avoid tight coupling with the domain
  - Use creational patterns that allow substitution
  - GUI should not create domain objects directly
  - Use interfaces in languages that require them
  - Mock the domain when testing the GUI

Principles

- Apply separation of concerns to the UI
  - Separate behavior from display
  - Use separate classes for validation
  - Use smart controls judiciously

Principles

- Avoid code generation pitfalls
  - Resist the temptation to insert code into the UI just because the IDE makes it easy.
  - Explore the limits of your tools
  - Where possible, split the code into two parts

Principles

Learn your platform and language thoroughly

- Explore key aspects needed for testing UI
  - Event generation and capture
  - Instantiation of UI elements as part of a test
  - Access to contained UI elements
- Be clear on what testing is needed

Principles

Understand your own common mistakes

- Don’t test getters and setters…

  unless you habitually mess them up

- Don’t test whether events are hooked up…

  unless you often forget to hook them up

PooleConsulting

Test-Driven Development in .NET

Three Simple Rules

Three Simple Rules of GUI Testing

- Get business functions out of the UI
  - Use domain or controller objects
  - Test them separately
- Get UI logic out of the Form
  - Separate controller objects
  - Derived or User Controls
  - Test them separately
- Know what you’re testing
  - Do you really need to test it?
  - Don’t test non-essentials

GUI Non-Essentials

Things that are usually non-essential

- Precise positioning of controls
- Shades of color
- Exact sizes

They become essential

- When they are hard requirements
- When code depends on them

PooleConsulting

Testing Events

1. If you generate the events, test that they are generated correctly
2. Test that you handle all possible sequences of events

But NOT at the same time!

The GUI Message

Questions?

PooleConsulting

Case Study

Event Generation

Event Example

- TestLoader loads and runs tests
- It is the source of many different events
  - TestLoaded, RunStarting, …
- UI objects react to these events
- How can we test one class at a time?

What We Want

```
[Test]
public void TestLoader()
{
       loader.LoadProject( assembly );
       Assert.Equals( 2, someObj.EventsSent );
       …
}
```

PooleConsulting

ITestEvents

```
public delegate void TestEventHandler(
           object sender, TestEventArgs args );
…
public interface ITestEvents
{
    event TestEventHandler ProjectLoading
    event TestEventHandler RunStarting
    …
}
```

TestEventDispatcher

```
public class TestEventDispatcher : ITestEvents
{
    // Implementations of each event
    event TestEventHandler ProjectLoading
    event TestEventHandler RunStarting
    …
    // Public methods to fire events
    …
}
```

TestLoader

```
// Simplified for presentation!
public class TestLoader
{
    private TestEventDispatcher events;

    public TestEventDispatcher Events
    {
        get { return events; }
    }
    …
}
```

TestEventCatcher

```
public class TestEventCatcher
{
    …
    public TestEventCatcher( ITestEvents source )
    {
        // Initializes collection and adds itself
        // to all the events the source provides.
    }

    public TestEventArgsCollection Events
    {
        // returns reference to internal collection
    }
}
```

TestLoaderTests

```
// Vastly oversimplified!
[TestFixture]
public class TestLoaderTests : TestEventDispatcher
{
    private TestLoader loader;
    private TestEventCatcher catcher;
    …
    [SetUp]
    public void SetUp()
    {
        loader = new TestLoader(…)
        catcher = new TestEventCatcher( loader.Events );
    }
    …
}
```

TestLoaderTests

```
// Vastly oversimplified!
[TestFixture]
public class TestLoaderTests : TestEventDispatcher
{
    …
    [Test]
    public void TestLoader()
    {
        loader.LoadProject( assembly );
        Assert.Equals( 2, catcher.Events.Count );
    }
    …
}
```

Event Example

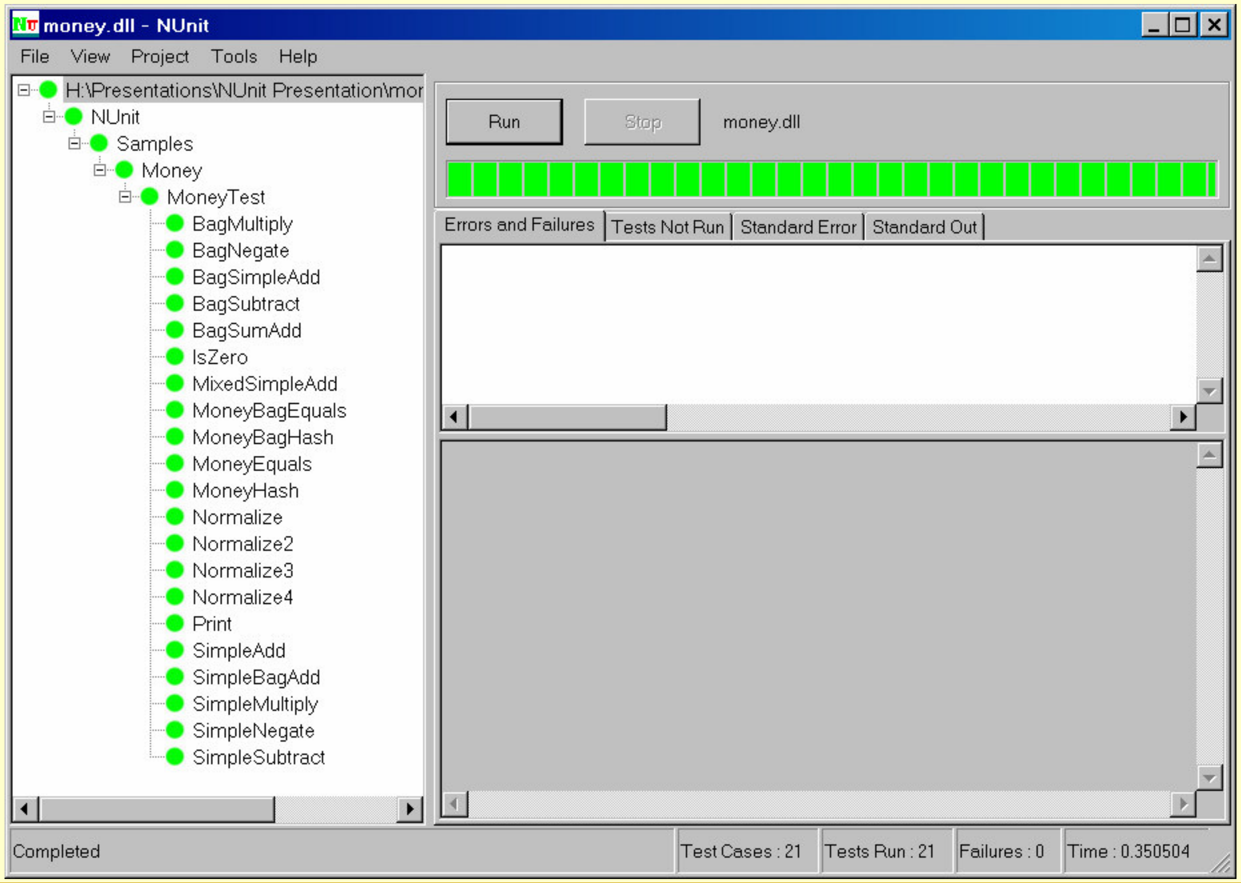
Questions?

PooleConsulting

Case Study

The Status Bar

The NUnit StatusBar



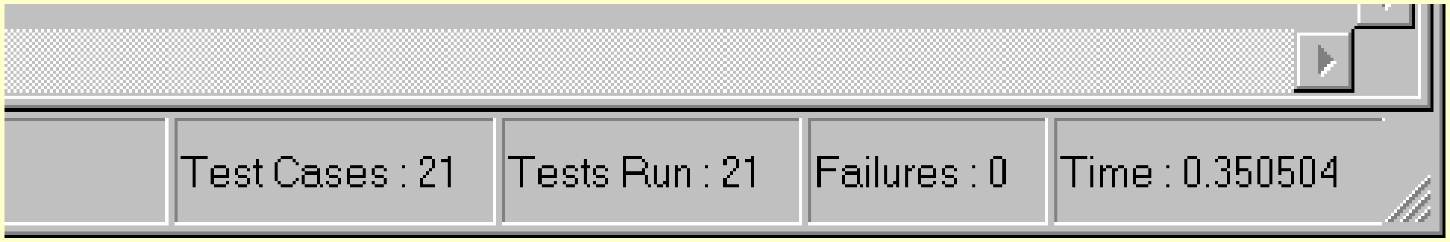
H:\Presentations\NUnit Presentation\mor
NUnit
Samples
Money
MoneyTest
BagMultiply
BagNegate
BagSimpleAdd
BagSubtract
BagSumAdd
IsZero
MixedSimpleAdd
MoneyBagEquals
MoneyBagHash
MoneyEquals
MoneyHash
Normalize
Normalize2
Normalize3
Normalize4
Print
SimpleAdd
SimpleBagAdd
SimpleMultiply
SimpleNegate
SimpleSubtract

money.dll

Completed | Test Cases : 21 | Tests Run : 21 | Failures : 0 | Time : 0.350504

PooleConsulting

The NUnit StatusBar


Test Cases : 21 | Tests Run : 21 | Failures : 0 | Time : 0.350504

Five panes

- Initialized as a test is loaded or selected
- Updated as a test run proceeds

StatusBar Test List

- Construct with five panes
- Initial content of panes
- Reinitialize for a different set of tests
- Display final result of a test run correctly
- Display each stage of a test as it progresses

Original Design

- Form creates and adds StatusBar to itself
- Form creates and adds panels to StatusBar
- Form sets values in StatusBar panels
- Form handles events and updates StatusBar

Original Design: Main Form

- Create StatusBar
- Position StatusBar on Form
- Create and initialize StatusBar panels
- Subscribe to all events related to StatusBar
- Handle each event, updating StatusBar

Original Design: StatusBar

- Just do what it’s told

Problems With Original Design

- Form knows too much
  - What the StatusBar contains
  - What the StatusBar displays
  - What events concern the StatusBar
  - What the StatusBar should do for each event
- Difficult to test StatusBar separately
- Reuse requires copy and paste
- Note: These problems are repeated for each control on the form

A More Testable Design

- Customize the StatusBar, so it
  - Knows it’s own contents
  - Knows what to display
  - Knows what events it needs
  - Knows how to react to events
  - May be re-used
  - May be independently tested

Re-design: Form Responsibilities

- Create StatusBar
- Position StatusBar on Form
- Pass event source to StatusBar

Re-design: StatusBar Responsibilities

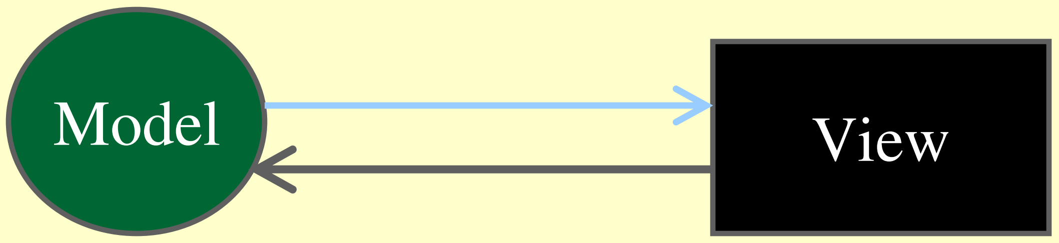
- Create and add panels to self
- Set initial panel display values
- Subscribe to events it cares about
- Process each event, updating its own display as appropriate

PooleConsulting

Test-Driven Development in .NET

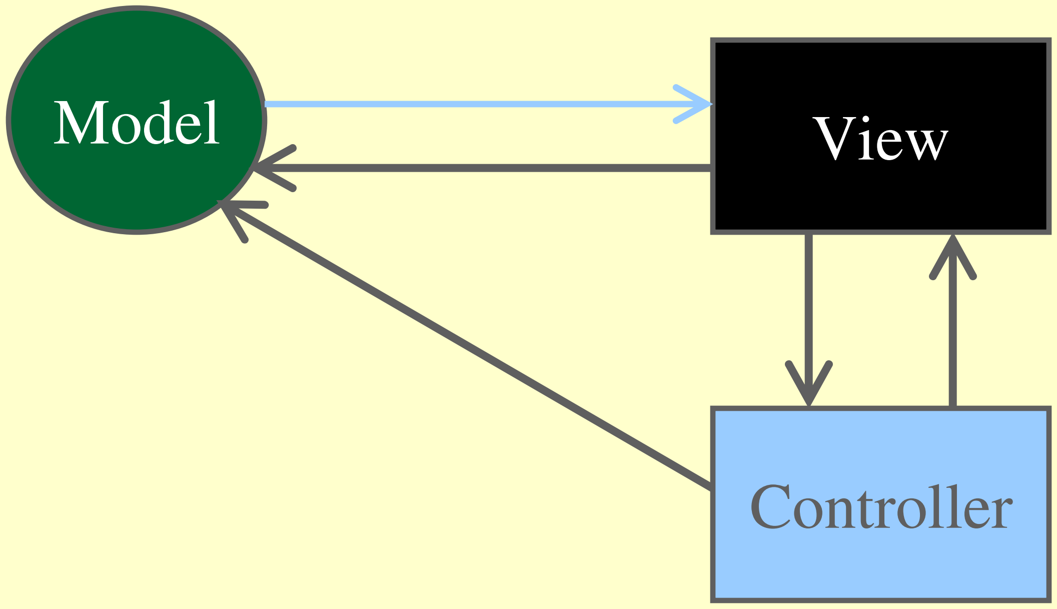
Design Patterns

Model-View



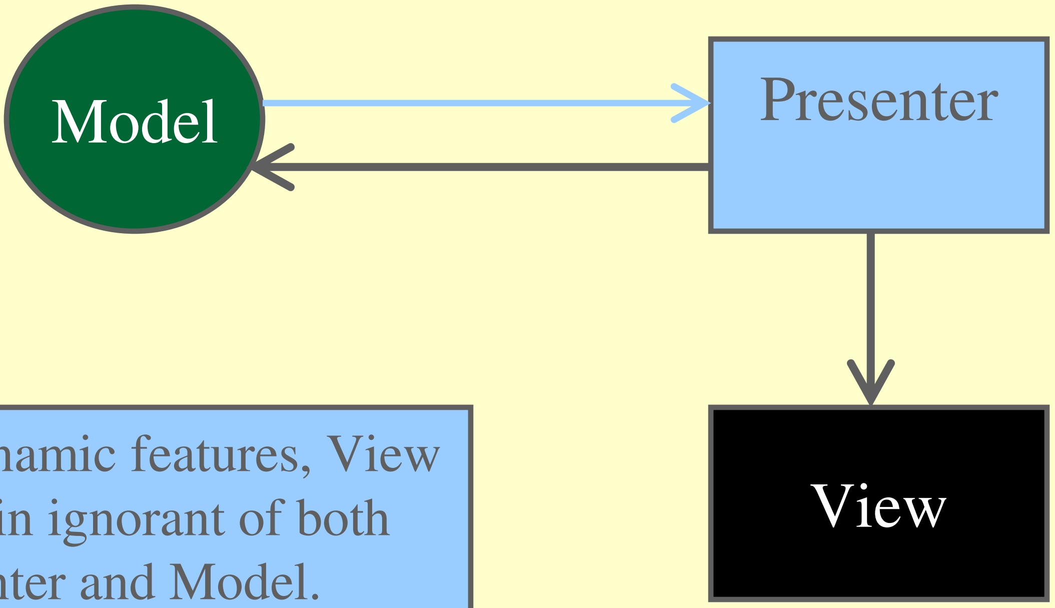
PooleConsulting

MVC



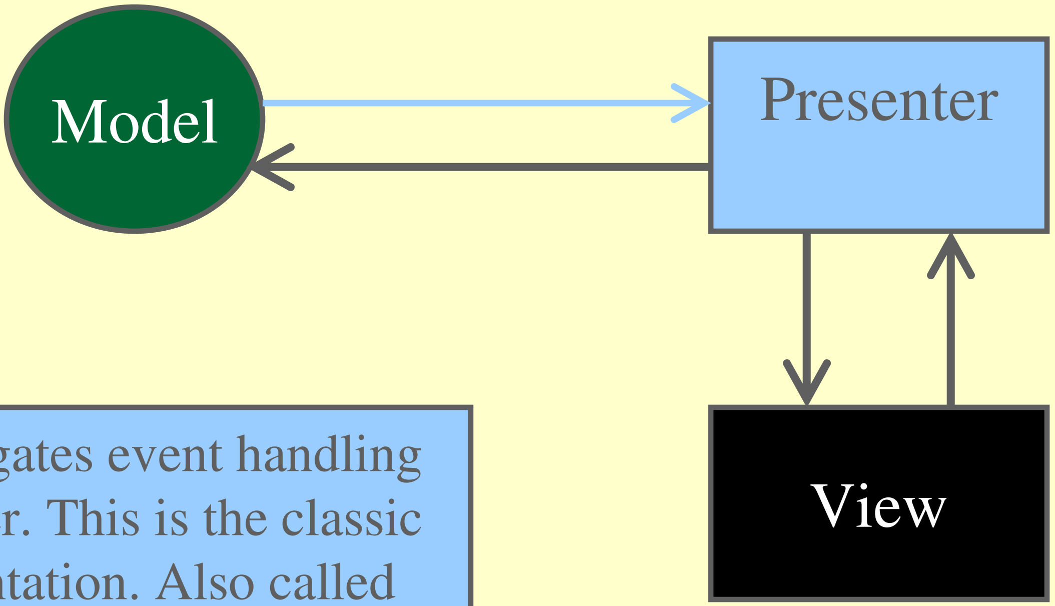
PooleConsulting

MVP



With no dynamic features, View can remain ignorant of both Presenter and Model.

MVP



View delegates event handling to Presenter. This is the classic implementation. Also called HumbleDialog.

Design Patterns

Questions?

Code Walkthrough

Conclusion

Conclusions

- "Hard to Test" often signals a need for change in the underlying code and/or areas we need to investigate further
- Improving testability tends to improve the application in terms of coupling and maintainability
- Testing of GUIs isn't all that hard[1] when approached carefully – we anticipate that this applies to other "hard" areas for testing.

Note 1: *That is, as compared to developing excellent software smoothly and rapidly in the first place.*

Questions?

Contact Info

- Email
  - charlie@pooleconsulting.com
  - charlie@nunit.com
- Web
  - www.pooleconsulting.com
  - www.charliepoole.org
  - www.nunit.com