Top 7 Lessons From My First Big Silverlight Project

Top 7 Lessons From My First Big Silverlight Project

Benjamin Day
Benjamin Day Consulting, Inc.

Level: Intermediate/Advanced

Benjamin Day

- Consultant, Coach, Trainer
- Professional Scrum Development Trainer
  - http://scrum.org
- Unit testing enthusiast
- Microsoft MVP for Visual Studio ALM
- Silverlight, Windows Azure, C#, Team Foundation Server
- http://blog.benday.com
- benday@benday.com

Benjamin Day Consulting

OVERVIEW

Goals

- Call out the pitfalls
- Learn from my mistakes
- Silverlight pain points
- Hopefully, avoid the pitfalls

Background

- Regulatory application for a Federal agency
- As of 1/15/2011:
  - Silverlight
    - 1,557 *.cs files
    - 11 MB of *.cs
  - Service code
    - 507 *.cs files
    - 2.3 MB of *.cs
- Approximately 12 months of development

It's reasonably large.

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

Technologies

- Silverlight 4
- GalaSoft MvvmLight
- Microsoft Unity
- .NET 4
- WCF
- Enterprise Library 5.0 Data Access Block
- SQL Server 2008 R2
- Windows 2008 R2
- Visual Studio 2010
- Team Foundation Server 2010

We’re not using RIA Services.

If you’re using RIA, that’s OK.

(You’re not necessarily a bad person.)

My $0.02 on RIA Services.

- In software, technologies that are supposed to save me time usually don't. They steal short-term time savings from long-term maintenance.

My $0.02 on RIA Services.

- On enterprise projects, time to market matters but not as much as the long-term maintenance and operational concerns of the app.

My $0.02 on RIA Services.

- If time to market matters more than maintenance or operations, RIA is an option.
- RIA introduces tight coupling between the client and server

THE LESSONS.

The 7 Lessons.

1. Client-side & Server-side: It's 2 applications.
2. Unit test, unit test, unit test.
3. Async WCF calls dictate your architecture.
4. Repository & Adapter patterns are your friend.
5. No shortcuts: Keep your ViewModels & Models separate.
6. Primitive Obsession in your ViewModel.
7. x:Name is a code smell.

Lesson 1: It's 2 applications.

Your app.

- Let's say you have a Silverlight UI
- Calls back to some WCF endpoints
- WCF services write data in to a database

- You're probably writing all this at the same time
- Feels like you're writing one application.

Client-side & Server-side: They're 2 applications

Service

- Service-Oriented Application (SOA)
- Uses a database for it's persistence
- "Domain Model" implementation
- User interface is a collection of WCF endpoints

Client

- Runs on a desktop machine
- Silverlight UI
- Has it's own object model
- Uses WCF services for persistence

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

Why do I think this?

- When data is returned via WCF, you're leaving the AppDomain
- Objects on the server are converted to XML
- XML is hierarchical

- When you're writing WCF applications, it's a mistake to think that you're returning objects.
- Your application is actually "message-oriented"

Why do I think this?

- Difficult to share code on both sides of the wire
- You're writing both apps at the same time but the server-side code doesn't need the Silverlight code.
- Silverlight definitely needs the service code.

Benefits of thinking about 2 apps

- It's like building a tunnel under a mountain.
- Meet in the middle
  - Design the endpoints
  - Server team works on server code
  - Client team works on client code
- Find problems faster
- Iterate with your customer faster

Benefits of thinking about 2 apps

- Code to abstractions
- → Fewer dependencies
- → Loosely coupled
- → Easier to test

Lesson 2: Unit test, unit test, unit test.

Disclaimer: I'm obsessed with unit testing.

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

Why unit test?

- What's "done" stays "done".
- You find problems early.
- Bugs stay dead.
- Refactoring is painless.

- If your application is large, it's tough to know when you've broken something obscure.
- Unit tests tell you when you've broken your app.

What do I test?

Silverlight

- ViewModels
  - Operations
  - Commands
  - INotifyPropertyChanged
  - Progress Bars
  - Show/Hide Logic
  - Login/Logout
- Domain Models
- Domain Model Services
- WCF logic
  - I don't test making WCF calls from Silverlight to the WCF Services
- Utility methods

Server-side

- Data access code
  - aka. Repositories
- Domain Model
- Domain Model Services
- Services
  - Direct calls without WCF
  - Calls through WCF

Which unit test frameworks?

- Server-side:
  Visual Studio 2010 (MSTest)
- Silverlight:
  Silverlight Control Toolkit Unit Test Framework
  - It's not great.
  - Better than nothing.
  - At least it runs in Silverlight.

Benefits of "2 applications" + unit testing

- Thinking of 2 apps helps isolate problems
- Are the service unit tests passing?
- Are the Silverlight unit tests passing?

Silverlight Passing?	Services Passing?	Conclusion
Yes	No	Service problem
No	Yes	Silverlight problem
Yes	Yes	New problem
No	No	Whew! $%^& is broken.

Tip: Don't write new code if the tests are broken.

- Always code on top of a known good state
- If you make changes and the tests stop passing, it's your fault.
  - Do your team a favor. Please, don't check in.

Side note: TFS 2010 Gated Check-in

- Continuous integration on steroids.
- Check in goes to a shelveset
- A build kicks off that compiles the shelved code
- Runs unit tests
- Compile fails or unit testing fails → your code doesn't get checked in
- No more broken builds!

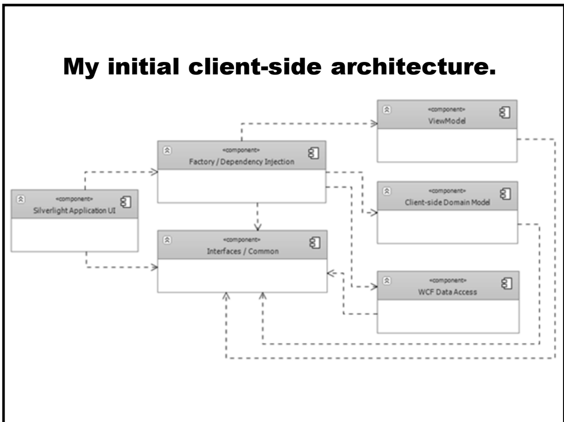
Lesson 3:
Async WCF rules your architecture.

(BTW…)

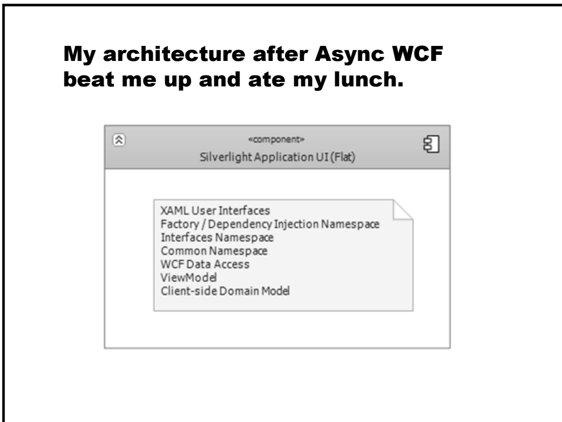
(Just between you and me.)

(This one nearly killed me.)

My initial client-side architecture.



My architecture after Async WCF beat me up and ate my lunch.



© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

Network traffic in Silverlight

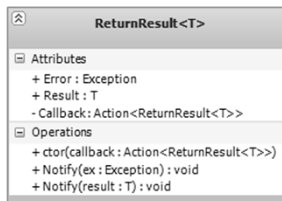
- It has to be async.
- If it isn't, the UI thread locks…forever.

Async Kills

- Your Repository methods can't return populated objects → must return void
- Exception handling is hard
  - Work happens on a different thread
  - Exceptions can't "bubble up" the stack
- You could have your *.xaml.cs handle the callbacks
  - Ugly
  - Violates "separation of concerns"

My Solution: ReturnResult<T>

- "Virtual" call stack
- Notify(Exception) or Notify(T)



The "glue" between method calls

```
public void LoadById(ReturnResult<IPerson> action, int id)
{
    try
    {
        IPerson returnValue = CreatePerson();
        // do some work...
        // yada yada yada
        action.Notify(returnValue);
    }
    catch (Exception ex)
    {
        action.Notify(ex);
    }
}
```

Alternate Solution

- Reactive Extensions for .NET
- http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx

Lesson 4:
Repository & Adapter Patterns
are your friend

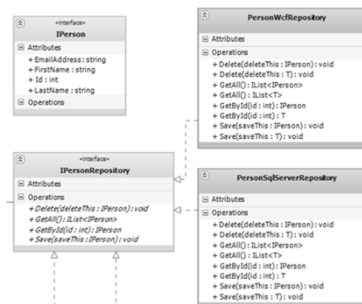
What is Repository?

The Repository Pattern

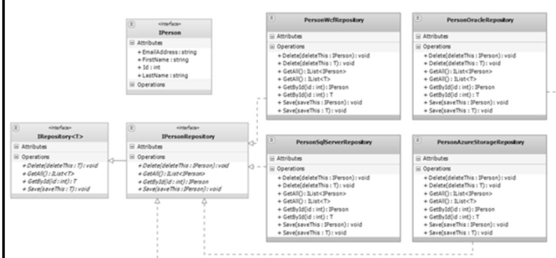
- *"Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects."*
  - http://martinfowler.com/eaaCatalog/repository.html
- Encapsulates the logic of getting things saved and retrieved

Synchronous Repository

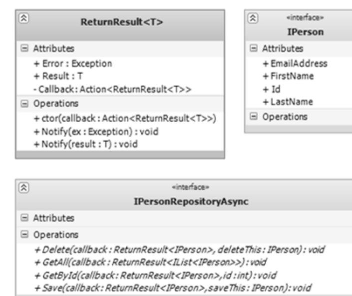
Synchronous SQL Server & WCF



A Big Picture



Async Repository



What is Adapter?

Adapter Pattern

- "…converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."
- *from "Head First Design Patterns" by Elisabeth & Eric Freeman*

My version of Adapter Pattern

- Take object of Type A and convert it in to object of Type B

Why are these patterns your friend?

- If you "Add Service Reference", these are *NOT* your Models or ViewModels
  - (I know it might be tempting.)
  - (Remember, it's 2 applications.)
- $0.02, you want your own Models and ViewModels
  - Breaks the dependency on the WCF services
- You'll convert to/from the Service Reference objects

Why are these patterns your friend?

- Help focus your mind
- Better design
- Help contain bugs
  - These conversions to/from will be buggy
- Help localize change
  - Service endpoint designs will change often
- Unit test the conversions separately
  - (Remember it's a "unit" test.)

SOLID Principles

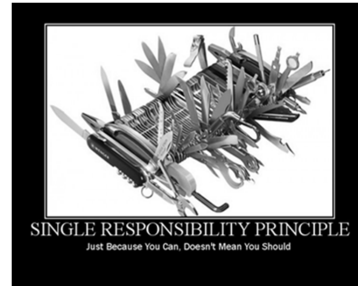
- Robert C. Martin
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

SOLID Principles of Class Design

Principle	Purpose
Single Responsibility	A class should have one, and only one, reason to change.
Open Closed	You should be able to extend a class's behavior without modifying it.
Liskov Substitution	Derived classes must be substitutable for their base classes.
Interface Segregation	Make fine grained interfaces that are client specific.
Dependency Inversion	Depend on abstractions, not on concretions.

Single Responsibility Principle



- Poster by Derick Bailey

Keep the Adapt separated from the Retrieve

- **Two classes**
  - Repository knows how to talk to the WCF service
  - Adapter knows how to turn the Service Reference types into Models

Lesson 5: No shortcuts: Keep your ViewModels & Models separate.

No shortcuts: Keep your ViewModels & Models separate.

- **It will be tempting to have your Repository/Adapter layer create ViewModels**
  - (Don't.)
- **There's a reason why it's called Model-View-ViewModel**

Why keep Model and ViewModel separated?

- **ViewModel is a user interface design**
- **Model is the state of your application**
  - aka. "Domain Model" pattern
- **ViewModel advocates for the UI**
  - 1-to-1 between a ViewModel and a *.xaml file
  - Might reference multiple Models
- **Don't have the ViewModel fields directly update the Model.**

It's all about the Cancel button.

- If you're "two way" data bound, How do you undo?

Cancel: ViewModel wraps Model

- ViewModel populates itself from the Model
- User edits the screen, ViewModel gets updated
- Model doesn't get changed until Save button is clicked.
- Model is The Boss.

Lesson 6: Primitive Obsession in your ViewModel.

Primitive Obsession

- James Shore's "Primitive Obsession"
  - Too many plain scalar values
  - Phone number isn't really just a string
  - http://www.jamesshore.com/Blog/
- Validation in the get / set properties is ok but is phone number validation really the responsibility of the Person class?

Coarse-Grained vs. Fine-Grained Object Model

- James Shore blog entry talks about Responsibilities
  - Fine-grained = More object-oriented
  - Data and properties are split into actual responsibilities
- I'm concerned about Responsibilities + Code Duplication + Simplicity

ViewModelField<T>

- Provides common functionality for a property on a ViewModel

With & Without ViewModelField<T>

PersonViewModelWithoutViewModelFields
Class

Properties
- EmailAddress : string
- EmailAddressValidationMessage : string
- FirstName : string
- FirstNameValidationMessage : string
- Id : int
- IsEmailAddressValid : bool
- IsFirstNameValid : string
- IsLastNameValid : bool
- IsSalaryValid : bool
- IsSalaryVisible : bool
- LastName : string
- LastNameValidationMessage : string
- Salary : int
- SalaryValidationMessage : string

PersonViewModelWithViewModelFields
Class

Properties
- EmailAddress : ViewModelField<string>
- FirstName : ViewModelField<string>
- Id : ViewModelField<int>
- LastName : ViewModelField<string>
- Salary : ViewModelField<int>

Are your ViewModel properties Coarse or Fine?

- **Fine-grained gives you room to grow**
- **ViewModelField<T>**
- **Create custom controls that know how to talk to your ViewModelFields**
  - Simplified binding expressions
- **Add features later**
  - Field validation later
  - Security

Lesson 7:
x:Name is a code smell.

What's a 'Code Smell'?

- **An indication that there *MIGHT* be a problem**
  - Quality
  - Maintenance
  - Performance
  - Etc…

Bad for ViewModel & UnitTesting

Named

```
<TextBox
    x:Name="m_textboxFirstName"
    Style="{StaticResource TextboxStyle}"
    Grid.Column="1"
    Grid.Row="2"
    KeyDown="TextBox_KeyDown"/>
```

- **Values populated by get/set statements**
- **Implies a lot of code in the "code behind" (*.xaml.cs)**

Not Named

```
<TextBox
    Style="{StaticResource TextboxStyle}"
    Grid.Column="1"
    Grid.Row="2"
    Text="{Binding FirstName, Mode=TwoWay}"
    KeyDown="TextBox_KeyDown"/>
```

- **Populated via binding expressions**
- **ViewModel-centric**
- **Minimal *.xaml.cs code**
- *** BETTER TESTABILITY ***

Where my opinion breaks down…

- **CodedUI Tests and Manual Test Automation in Microsoft Test Manager (MTM) & VS2010 Ultimate**
  - Visual Studio 2010 Feature Pack 2
  - Requires controls to be uniquely named in order to automate simulated user interactions
- **Animation Designers in Blend**
  - (I have been told this but have not verified this myself.)

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

Remember:
A “smell” is not necessarily bad.

Bad Smell

- This is garbage.
- Do not eat.
- Smells awful.

Good Smell

- This is Langres
- From Champagne-Adrenne region of France
- Cow’s milk
- Tastes awesome
- Smells fairly awful

Photo: Benjamin Day, 12/14/2008

Good Smell vs. Bad Smell

- Cheese is controlled rot. It’s a way of preserving food. It’s supposed to be that way.
- Garbage is garbage.

The Recap.

The 7 Lessons.

1. Client-side & Server-side: It’s 2 applications.
2. Unit test, unit test, unit test.
3. Async WCF calls dictate your architecture.
4. Repository & Adapter patterns are your friend.
5. No shortcuts: Keep your ViewModels & Models separate.
6. Primitive Obsession in your ViewModel.
7. x:Name is a code smell.

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com

Any last questions?

Goals

- Call out the pitfalls
- Learn from my mistakes
- Silverlight pain points
- Hopefully, avoid the pitfalls

Thank you.

http://blog.benday.com | http://www.benday.com | benday@benday.com

© 2011 Benjamin Day Consulting, Inc. -- www.benday.com