KING TUT
ARCHITECTURE

PYRAMIDS, PATTERNS, AND TESTS
I'M NOT SAYING IT WAS ALIENS...

BUT IT WAS ALIENS
THE (AUTOMATION) TESTING PYRAMID

Ideas from Mike Cohn’s Succeeding With Agile, graphic from http://martinfowler.com/bliki/TestPyramid.html
KING TUT
ARCHITECTURE

PYRAMIDS, PATTERNS, AND TESTS

GARY PEDRETTI

• Over 20 years in the software industry with highly cross-functional experience – DBA, Developer, BA, Application Architect
• Currently Trainer, Coach, Consultant, Owner at Sodoto Solutions
• SODOTO = See One, Do One, Teach One
• Scrum: Development Team member, Scrum Master, Coach, Professional Scrum Trainer for Scrum.org
• http://blog.GaryPedretti.com/
• http://www.linkedin.com/in/garypedretti
• Twitter: @GaryPedretti
Let’s talk some details...

To achieve these proportions and still cover the application adequately requires good application architecture and design patterns!

“Quality is built-in from the beginning!!”
“Quality is Job # 1!!”
BDD frameworks (Cucumber, SpecFlow) could use any of these test types and tools to implement their test steps.
"Real" unit tests that isolate dependencies with fakes, mocks, stubs, dummies, etc.

These tests might be written by a unit testing framework, or a service- or integration-specific tool like Fit, Fitnesse, etc.
Testing Pyramid

UI Integration & Service

Unit

Typical Manual UI/End-to-End Integration Testing

UI
Business Logic
Data Access Layer
Database

Automated UI Test

Advanced Automated UI Testing

UI
Business Logic
Data Access Layer
Database

Mocked Business Logic

e.g., testing a View with a Mock Controller in an MVC pattern
So, we talked some details...
## ON SOLID...

<table>
<thead>
<tr>
<th>SOLID</th>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Single Responsibility Principle</td>
<td>A class has one single responsibility and therefore one reason to change.</td>
</tr>
<tr>
<td>O</td>
<td>Open/Closed Principle</td>
<td>Open for extensions, closed for modifications.</td>
</tr>
<tr>
<td>L</td>
<td>Liskov Substitution Principle</td>
<td>Subtypes must be substitutable for their base type with no ill effects.</td>
</tr>
<tr>
<td>I</td>
<td>Interface Segregation Principle</td>
<td>Not one big interface for all clients, but one interface per kind of client. No methods in interface that client does not use. No “fat” interfaces.</td>
</tr>
<tr>
<td>D</td>
<td>Dependency Inversion Principle</td>
<td>High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.</td>
</tr>
</tbody>
</table>

**SINGLE RESPONSIBILITY PRINCIPLE**

*Just Because You Can, Doesn't Mean You Should*


---

**OPEN CLOSED PRINCIPLE**

*Open Chest Surgery Is Not Needed When Putting On A Coat*

Graphic from http://www.tomdalling.com/blog/software-design/solid-class-design-the-open-closed-principle/
LISKOV SUBSTITUTION PRINCIPLE
If it Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction


INTERFACE SEGREGATION
Tailor interfaces to individual clients' needs.

Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?


INVERSION OF CONTROL
ASPECT-ORIENTED PROGRAMMING
MV* FRAMEWORKS
ONION ARCHITECTURE

FOUR PATTERNS /
ORIENTATIONS /
FRAMEWORKS
INVERSION OF CONTROL

INVERSION OF CONTROL

- Presentation Layer
- Business Logic Layer
- Data Access Layer
- Data Source
INVERSION OF CONTROL

See Fowler’s “Inversion of Control Containers and the Dependency Injection pattern” at http://martinfowler.com/articles/injection.html for a great discussion on choices and tradeoffs here.

INVERSION OF CONTROL

High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend on details. Details should depend on abstractions.

INVERSION OF CONTROL

Very Nice!!
INVERSION OF CONTROL

WHY?

INVERSION OF CONTROL

SOLID Principles

• Single Responsibility Principle
• Dependency Inversion

Domain-Driven Design (DDD) ideas (a/k/a “Real” Object-Orientation)

• This just models the real world anyway
• Insurance rate computation example

Various -ilities, especially maintainability

Implementation Flexibility

Testability!!! Real Unit Tests at the Bottom of Your Pyramid!!
ASPECT-ORIENTED PROGRAMMING

Shared
Common
Not-A-Layer
Bad Idea?
public class MegaDivider
{
    private ILog logger;

    public int Divide(int numerator, int denominator)
    {
        logger.DebugFormat("MegaDivider.Divide method entered at ", DateTime.UtcNow);
        try
        {
            return numerator / denominator;
        }
        finally
        {
            logger.DebugFormat("MegaDivider.Divide method left at ", DateTime.UtcNow);
        }...
    }
}
ASPECT-ORIENTED PROGRAMMING

Very Nice!!

ASPECT-ORIENTED PROGRAMMING

WHY?
ASPECT-ORIENTED PROGRAMMING

SOLID Principles
  • Single Responsibility Principle
Various -ilities, especially maintainability
Domain Driven Design (DDD) ideas (a/k/a “Real” Object-Orientation)
  • Readability
Testability!!! Real Unit Tests at the Bottom of Your Pyramid!!

MVC, MVP, MVVM

MV* PATTERNS
MV* PATTERNS

Images from http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod

MV* PATTERNS

MV* PATTERNS

Very Nice!!
**MV* PATTERNS**

SOLID Principles

- Single Responsibility Principle

Various -ilities, especially maintainability

In-Line With Various Modern Frameworks and Tools

Testability!!! For the Pyramid:

- Model: Unit Testable
- Controllers/Presenters/View Models: Unit Testable
- View: Very Thin (IF YOU FOLLOW THE INTENDED PRINCIPLES), Allows You To Keep the Top of Your Pyramid Small

---

**ONION ARCHITECTURE**
Onion Architecture

Graphic from http://jeffreypalermo.com/blog/the-onion-architecture-part-2/

Meet a Meaningless Metaphor,

Graphic from Meet a Meaningless Metaphor, http://wakefelderman.blogspot.com
IT’S A METAPHOR

- Domain-Driven Design (DDD)
  - Domain is central to the system – true Domain Model
  - Patterns and approaches like Bounded Context, Repositories
- Separation of Concerns
- Inversion of Control or Separated Interface - Loose coupling based on interfaces at design time, concretes only at runtime
- Persistence Ignorance
- Mocking – Application core can be compiled, run, and tested with no infrastructure

ONION ARCHITECTURE TENETS

- The application is built around an independent object model
- Inner layers define interfaces
- Outer layers implement interfaces
- Direction of coupling is toward the center
- All application core code can be compiled and run separate from infrastructure
A KA HEXAGONAL ARCHITECTURE

AKA HEXAGONAL ARCHITECTURE, PORTS & ADAPTERS, OR OBJECT STRUCTURAL

Create your application to work without either a UI or a database so you can

- Run automated regression-tests against the application
- Work when the database becomes unavailable
- Link applications together without any user involvement
BRINGING IT ALL TOGETHER

TESTING PYRAMID = GOOD DESIGN

Now you have good ideas about the WHYs of

• Inversion of Control (IoC)
• Aspect-Oriented Programming (AOP)
• MVx (MVC, MVP, MVVM)
The Testing Pyramid = Good Design

- The WHYs for All 3 of These Are In Direct Support of the Testing Pyramid

- The WHYs for All 3 of These Are In Direct Support of Good Application Architecture and Design

The Testing Pyramid = Good Design

- The Testing Pyramid = Good Application Architecture and Design

- The Testing Pyramid != Testing ??

- The Testing Pyramid = Application Architecture and Design Statement !!
TESTING PYRAMID = GOOD DESIGN

• What Happens When You DON’T Use These Patterns?

WHAT ABOUT...

Are we only doing this for testing? Where’s the value?

Sure glad the hole isn’t at our end.
THE BAR IS HIGH

@GARYPEDRETTI
GARYPEDRETTI.COM
SODOTOSOLUTIONS.COM
WWW.LINKEDIN.COM/IN/GARYPEDRETTI

THANK YOU!