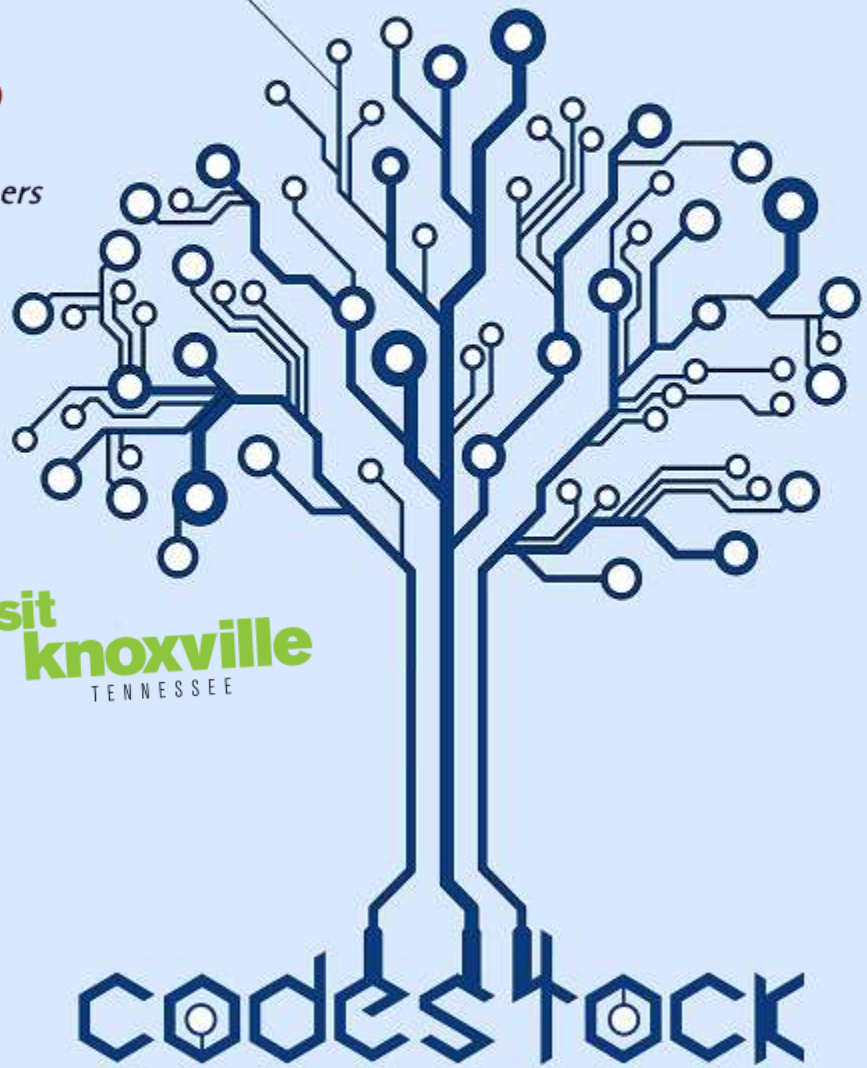
Principles of Application Architecture

9:25am/ Ballroom B



Premier Staffing Partners

RECRUITWISE

visit knoxville TENNESSEE

PRELIMINARIES

- Who am I?
  - Steve Barbour
  - @steverb
  - steverb.com
- Why should you listen to me?
  - You shouldn’t
- Ground rules for this talk
  - I do NOT have all the answers
  - Discussion and disagreement are welcome
  - Keep it civil
  - We do have a time limit

WHAT IS APPLICATION ARCHITECTURE?

- High level breakdown of system into its parts.
- Things that are hard to change.
- Mostly concerned with the public interfaces, not the private implementation.
- Whatever the important stuff is.

WHY SHOULD I CARE?

BAD ARCHITECTURE

- Overly Complex
- Error Prone
- Fragile
- Hard to understand
- Rigid

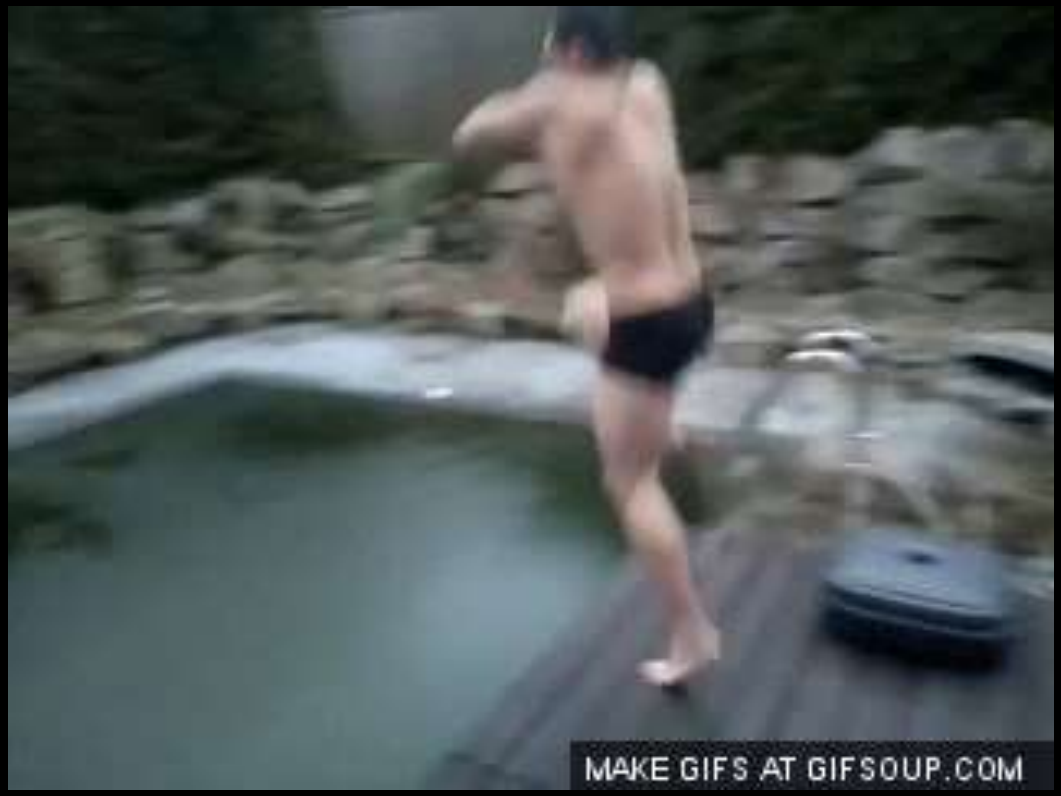
GOOD ARCHITECTURE

- Simple as possible
- Correct
- Robust
- Understandable
- Flexible

CONTINUUM

- The line between architecture and craftsmanship is blurry
  - System
  - Application
  - Modules/Classes
  - Functions

KEY PRINCIPLES / SOLID



SINGLE RESPONSIBILITY

- A component should have only one (business) reason to change.
- Having separate components that do not overlap in functionality makes the system/application easier to understand, reason about, test and change.

THE MODEM CLASS

```
Public interface Modem
{
        public void Dial (string phoneNumber);
        public void Hangup();
        public void Send (char c);
        public char Recv();
}
```

OPEN/CLOSED

- Entities should be open for extension but closed for modification
- Open for Extension – Behavior can be extended
- Closed for Modification – Extending behavior doesn't require source changes to the entity.
- In C#, think abstract classes, or implementing interfaces.

THE SQUARE CIRCLE PROBLEM

```
Void DrawAllShapes(Shapes list[])
{
        foreach(shape in Shapes)
        {
                switch(shape.type)
                {
                        case square:
                                DrawSquare(shape);
                        break;

                        case circle:
                                DrawCircle(shape);
                        break;
                }
        }
}
```

THE LISKOV SUBSTITUTION PRINCIPLE

- Subtypes must be substitutable for their base types.
- “A model, viewed in isolation cannot be validated.”
- “The validity of a model can be expressed only in terms of its clients.”
- The IS-A relationship applies to behaviors.

THE RECTANGLE AND THE SQUARE

```
public class Rectangle
{
        private Point topLeft;
        private double width;
        private double height;

        public virtual double Width{get; set;}
        public virtual double Height{get; set;}
}
```

THE RECTANGLE AND THE SQUARE- CONT

```
public class Square:Rectangle
{

        public override double Width
        {
                set
                {
                        base.Width = value;
                        base.Height = value;
                }
        }
}
```

INTERFACE SEGREGATION

- No client should be forced to depend on methods they don’t use.
- Regular door and a timed door…

DEPENDENCY INVERSION

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

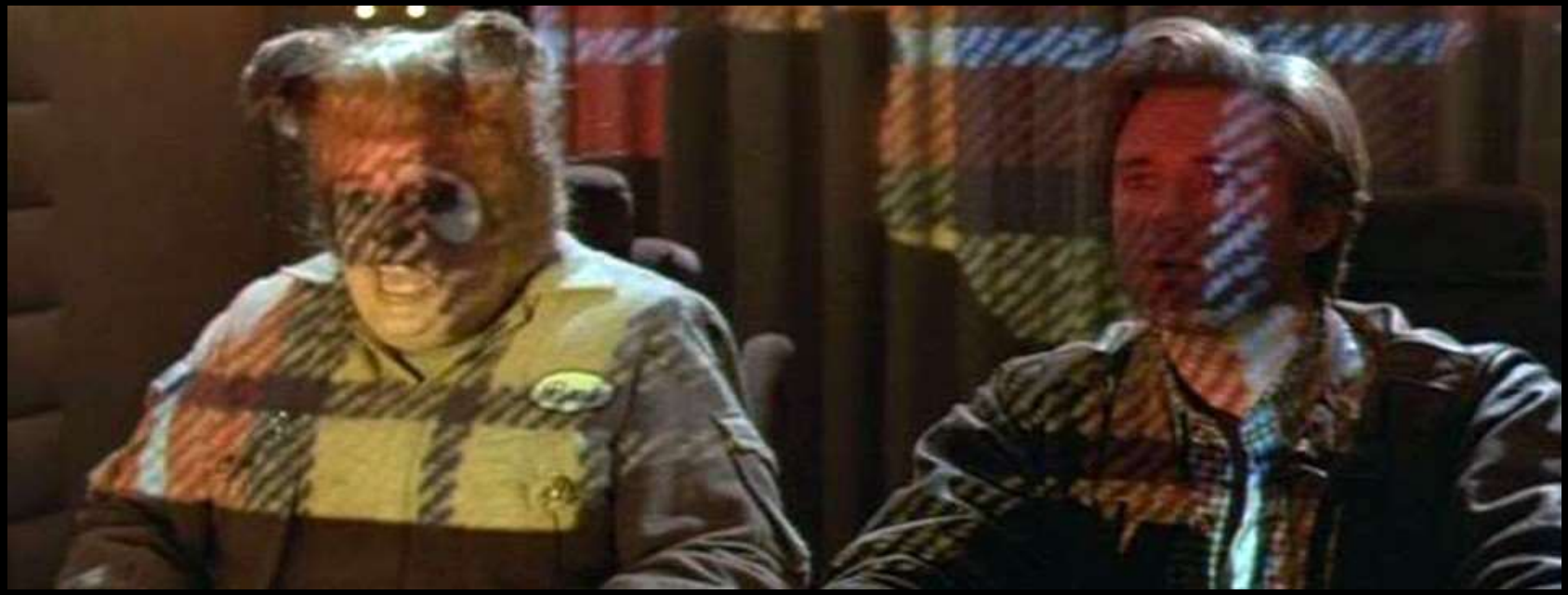
GOOD PRACTICES

- Don’t Repeat Yourself (DRY)
- You Ain’t Gonna Need It (YAGNI)
- Keep it stupid simple (KISS)
- Keep the design consistent within each layer.
- Prefer composition to inheritance.
- Be explicit about communication.
- Keep data formats consistent.
- Separate crosscutting code.
- Only talk to your immediate friends (LoD)?

BAD PRACTICES

- Premature Generalization
- Premature Optimization
- Pattern all the things!
- No non-functional requirements
- The one true way to do it
- Resume driven development
- Einstein syndrome
- Fire and forget
- Quality is job #987

PATTERNS



ARCHITECTURAL STYLES / PATTERNS

- Patterns that are mostly concerned with how applications are structured and developed.
- Principles that shape an application.
- You can and probably will mix and match styles.
- Conway's Law
  - "Software will reflect the organization that created it."

LAYERS

- Functionality in each layers is related by common responsibility.
- Communication is explicit and loosely coupled.
- Benefits
  - Abstraction
  - Isolation
  - Manageability
  - Reusability
  - Testability

CLIENT/SERVER

- Segregates the system into two applications, client makes requests to the server.
- Classically, the server is a database with application logic in stored procedures.
- Benefits
  - Security
  - Centralized Access
  - Maintainance

N-TIER / 3-TIER

- Like layers, but physically separated.
- Tiers are completely separate from other tiers and only communicate with the tiers immediately adjacent.
- Benefits
  - Maintainability
  - Scalability
  - Flexibility
  - Availability

MESSAGE BUS

- Receive and send messages using a common bus.
- Communication is explicit and loosely coupled.
- Pub/Sub, Message Queueing.
- Benefits
  - Extensibility
  - Low Complexity
  - Flexibility
  - Loose Coupling
  - Scalability
  - Application Simplicity

MICROKERNEL

- Core System + Plug-ins
- Core traditionally contains minimal functionality
- Benefits
  - Extensibility
  - Ease of Deployment
  - Testability

SERVICE ORIENTED ARCHITECTURE

- Boundaries are explicit.
- Services are autonomous.
- Services share schema and contract, not class.
- Service compatibility is based on policy (transport, protocol and security).
- Benefits
  - Loose Coupling
  - Ease of deployment (kinda)
  - Testable
  - Scalable
- Now for something completely different.

MICROSERVICES

- Functionality provided as (web) services
- Boundaries are explicit.
- Services are autonomous.
- Might not expose schema or contract.
- Generally little, restish services.
- Benefits
  - Loose Coupling
  - Ease of deployment (kinda)
  - Testable
  - Scalable

SO MANY ARCHITECTURAL PATTERNS

- Monolithic (Big Ball of Mud)
- Blackboard (Shared Memory)
- Hexagonal
- CQRS
- Event Driven
- Peer to Peer
- REST
- Shared Nothing Architecture (Sharding)
- Space Based Architecture (Good Luck)

ARCHITECTURAL DESIGN

- How to document and communicate?
- How much documentation do we need?
- When do we design the architecture?
- How much time should we spend architecting?

WE HAVE ONLY SCRATCHED THE SURFACE

- Software Architecture and Design from Microsoft Patterns and Practices.
- Patterns of Enterprise Application Architecture by Martin Fowler.
- Martin Fowler's Catalog of Patterns
- Martin Fowler's Site, anything by Martin Fowler really.
- Agile Principles, Patterns and Practices in C# by Robert C. Martin
- Head First Design Patterns - A little light, but a good intro.
- Wikipedia
- O'Reilly

SERIOUSLY? YOU'RE NOT DONE YET?

- @steverb
- steverb.com
- me@steverb.com

EXTRA SLIDES IN CASE I TALK WAY TOO FAST

DOMAIN LOGIC PATTERNS

- Transaction Script - Organizes business logic by procedures where each procedure handles a single request from the presentation layer.
- Domain Model - An object model of the domain that incorporates both behavior and data.
- Table Module - A single instance that handles the business logic for all rows in a database table or view.
- Service Layer - Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

DATA ACCESS PATTERNS

- Row Data Gateway - An object that acts as a Gateway to a single record in a data source. There is one instance per row.
- Table Data Gateway - An object that acts as a gateway to a database table. One instance handles all the rows in the table.
- Data Mapper - A layer of mappers that moves data between objects and a database while keeping them independent of one another and of the mapper itself.
- Active Record - An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.
- Command Query Responsibility Segregation - Use a model to update data that is different from the model used to read data.

PRESENTATION PATTERNS

- Model View Controller - Splits user interface interaction into three distinct roles.
- Page Controller - An object that handles a request for a specific page or action on a web site.
- Front Controller - A controller that handles all requests for a web site.
- Template View - Renders information into HTML by embedding markers in an HTML page.
- Transform View - A view that process domain data element by element and transforms it into HTML.
- Two Step View - Turns domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.
- Application Controller - A centralized point for handling screen navigation and the flow of an application.