Avoiding Five Common Architectural Pitfalls

Anthony Sneed

About Me

- Personal
  - Married, three children
  - Los Angeles, Dallas, Slovakia
- Work
  - Wintellect: Author, Instructor, Consultant
  - Training Videos: wintellectnow.com
- Open Source on GitHub
  - Simple MVVM Toolkit
  - Trackable Entities

Contact Me

- Email
  - tony@tonysneed.com
- Blog
  - blog.tonysneed.com
- Social Media
  - Twitter: @tonysneed
  - Google+: AnthonySneedGuru
  - Facebook: anthony.sneed
  - LinkedIn: tonysneed

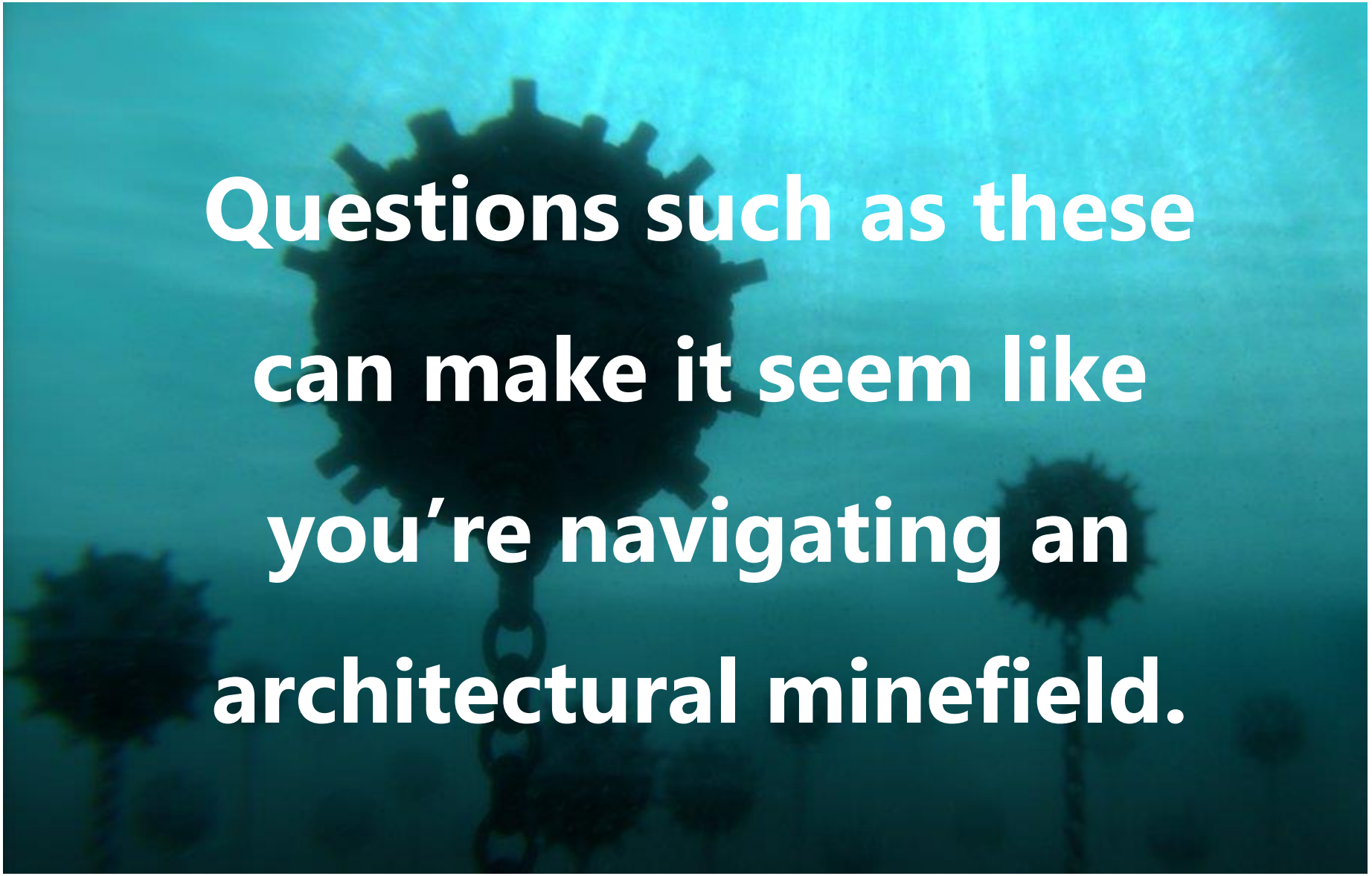
Have You Ever Wondered

- "How do I know when a particular technology is **dead** or will **soon die**?"

- "How do I know whether a specific **design pattern** will simplify or complicate development?"

Have You Ever Wondered

- "How open should I be to **open-source**?"
- "How much should I be concerned with **tight coupling**?"
- "Which approach to **testing** makes most sense for this project?"


Questions such as these
can make it seem like
you're navigating an
architectural minefield.

Today We're Going To Discuss

- Five common architectural **pitfalls**

- Steps you can take to **avoid** them

Common Architectural Pitfalls

1. Relying on **deprecated** technologies
2. Blindly adopting "**what's hot now**"
3. Failing to apply the right level of **abstraction**
4. One-size-fits-all approach to **testing**
5. Being closed to **open source**

Pitfall #1

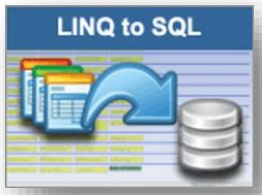
Relying on Deprecated Technologies

A Brief Survey …

- Have you ever selected a platform or framework only to find out later it was already **dead** or **dying**?

Have You Used These Recently?

- LINQ to SQL
- Windows Forms
- Silverlight
- Windows Communication Foundation



Case in Point: The Rise of Silverlight

- First there was **Adobe Flash**
- Microsoft wanted a **cross-platform** solution
- Better **deployment** model than Click Once
- Browser **plug-in** seemed like natural choice
- **WCF RIA Services** positioned SL for LOB apps

Case in Point: The Fall of Silverlight

- Steve Jobs **killed Flash** (battery, perf, security, etc)
- **HTML5** offers an x-plat *web* solution
- **Xamarin** offers an x-plat *native* solution
- **App stores** provide the deployment vehicle
- Those who **bet** on SL for LOB apps **lost** big time.

Case in Point: The Rise of WCF

- Single **unified API** (Sockets, Remoting, ASMX, MSMQ)
- Promoted **transport independence**
- Implemented **SOAP** standards (WS-*)
- Emphasized **contracts** and **metadata** (WSDL)
- Lots of **extensibility** hooks, Visual Studio **tooling**

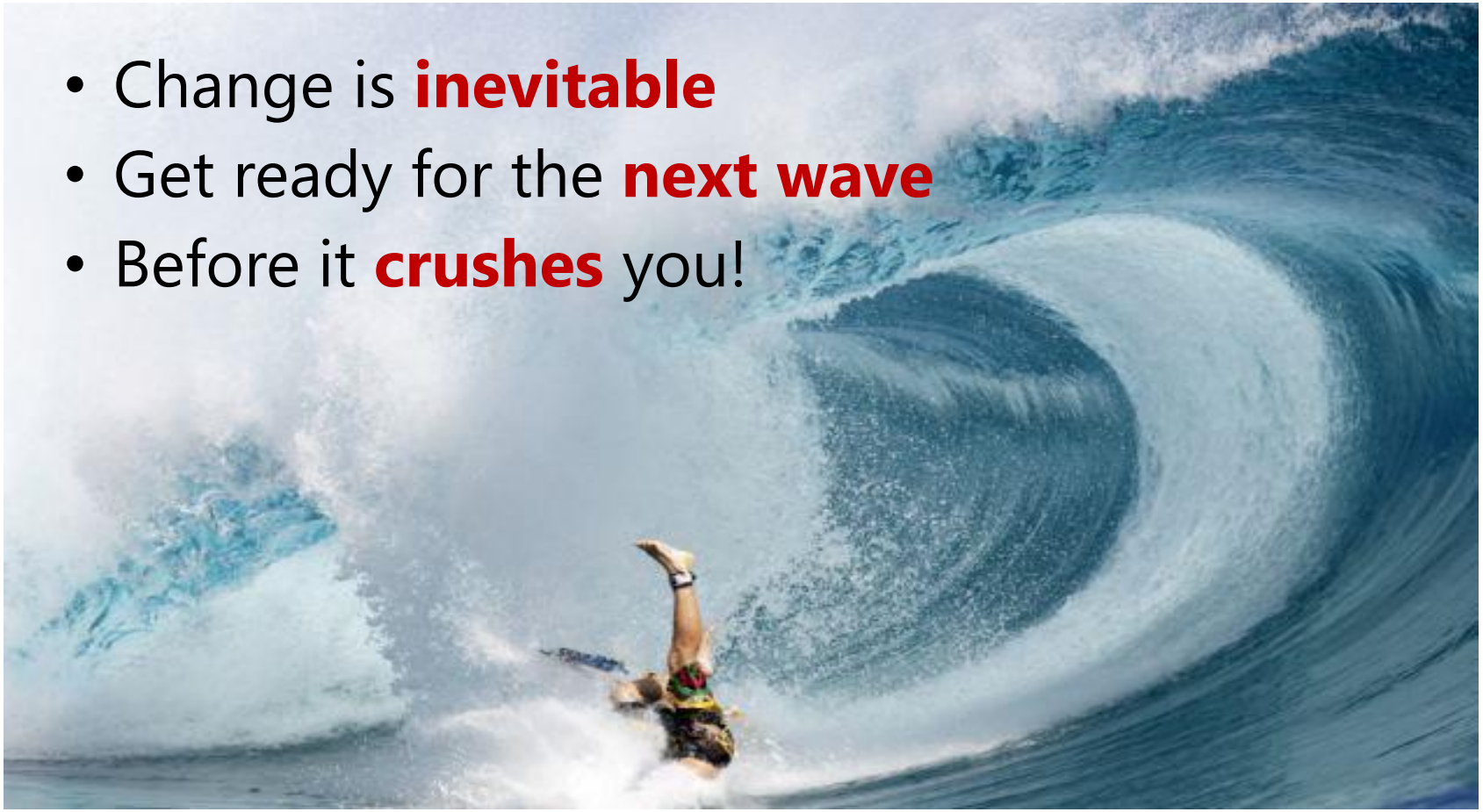
Case in Point: The Fall of WCF

- Some **clients** don't love **SOAP** (browsers, mobile)
- **SOAP specs** never caught on (WS-Security, etc)
- Good idea to leverage **HTTP** (RESTful apps)
- Had to be an **expert** to use WCF properly
- WCF not friendly to **Dependency Injection**

Pitfall #1: How to Mitigate the Risk?

- Don't rely on **official pronouncements** from Microsoft
  - They're behind the curve too!
- Thoroughly **evaluate** alternate technologies
  - For ex, **WPF** can live in IE on Windows (XBAP's)
  - You can host **Web API** apps without IIS
- Apply **patterns** for looser coupling
  - Separate presentation logic using **MVVM**
  - Isolate data layer with **Dependency Injection**

Conclusion: Roll with the Changes



Pitfall #2

Blindly Adopting

"What's Hot Now"

Drinking the Kool-Aid

MVVM!

MEF!

Dependency Injection!

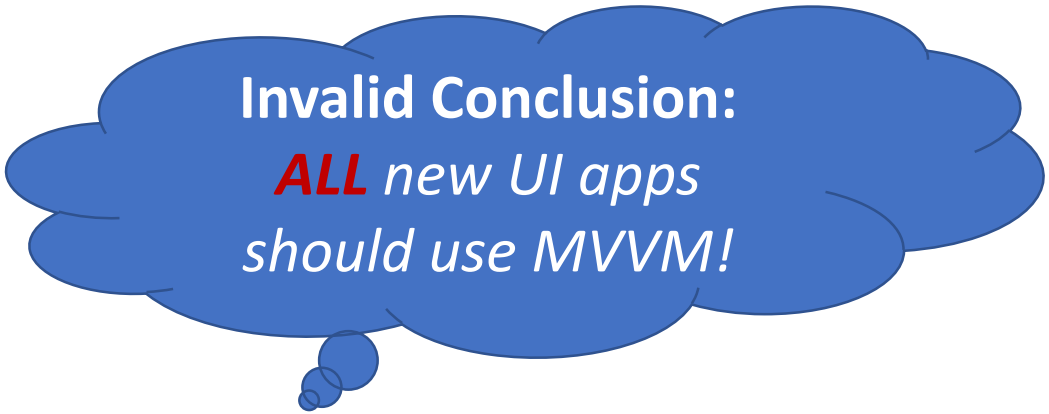
Rx!

It's Hot! What's the Problem?

- There's usually a **productivity** hit
  - There can be a steep **learning curve**
  - **Tooling** can lag behind new technologies
- There can be a **performance** hit
  - Defaults tend to favor **"Hello World"** scenarios
  - Tuning requires **advanced skills** (threading, security)
- There is often a **maintenance** hit
  - Blind adoption leads to **adulteration**
  - Tend to either **overuse** or **sabotage** patterns

Case in Point: Benefits of MVVM

- **Maintainability:** encapsulate presentation logic
- **Blendability:** graphic designers apply behaviors
- **Testability:** unit tests can drive view models



Case in Point: Misuse of MVVM

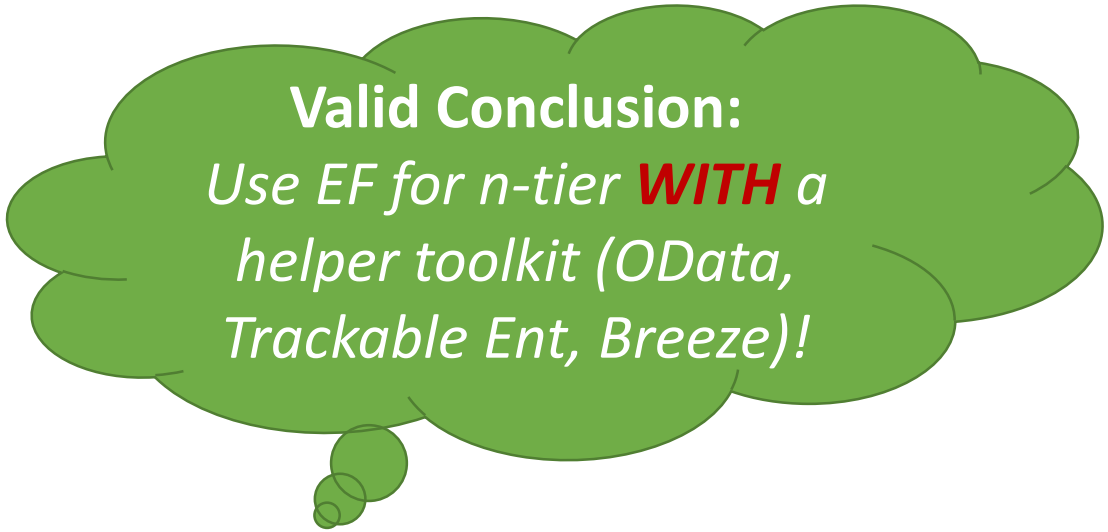
- Overusing MVVM can lead to **code bloat**
- Misunderstanding MVVM can result in **sabotaging** the pattern
  - Showing dialogs from view models render them **untestable**
- Using MVVM without a **toolkit** is painful

Case in Point: Entity Framework Defaults

- **Least** common usage is **most** supported
  - **2-tier** with client-server
  - For ex, change-tracking, dynamic proxies, lazy loading
- **Most** common usage is **least** supported
  - **N-tier** with web service

Case in Point: Entity Framework Gotchas

- Dynamic proxies are not **serializable**
- Lazy loading is not **appropriate** for n-tier apps
- Serializers must be configured to handle **cycles**
- Changes not **tracked** across service boundaries



Pitfall #2: How to Mitigate the Risk?

- Don’t select a technology based on **popularity**
- Make sure a technology benefits **your** use cases
- Don’t blindly accept the common **defaults**
- Architect for **scalability**, **performance**, **security**
- Seek objective **feedback** whenever possible

Conclusion: Architectural Reviews

- Review architecture **early** and **often**
- Know the **why** you select specific technologies
- Distinguish emerging **trends** from passing **fads**
- Design for things like **scalability** and **security**

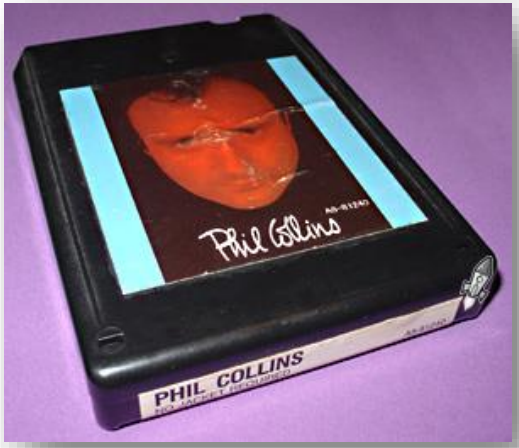
Pitfall #3

Failing to Apply the

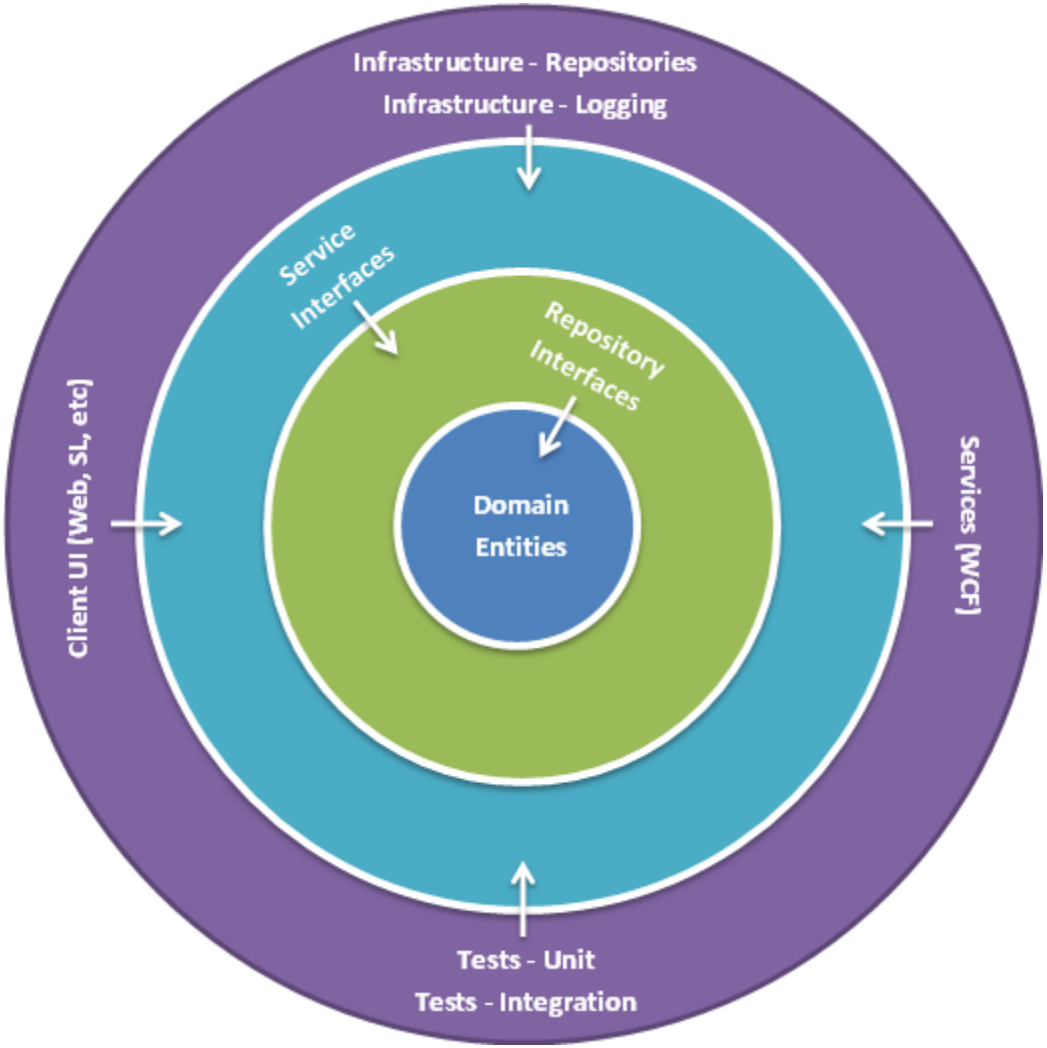
Right Level of Abstraction

The Only Constant Is Change

- Obsolete frameworks can make an entire application **obsolete**
- Abstracting away **infrastructure concerns** can help insure against obsolescence



Like the Layers of an Onion

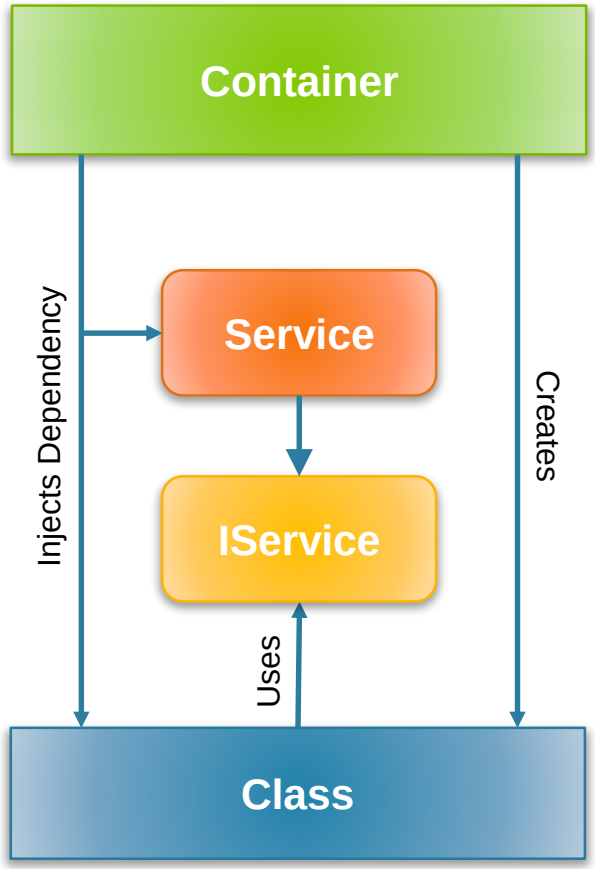


Common Patterns for Abstracting Data

- **Repository**
  - Interfaces for accessing a **data store**
  - Can expose **CRUD** operations
- **Unit of Work**
  - Operations which span **multiple** repositories
  - Can provide support for **transactions**

Dependency Injection

- Class **uses** **IService** interface
- DI container **creates** concrete **Service** class
- DI container **configured** on app **startup**

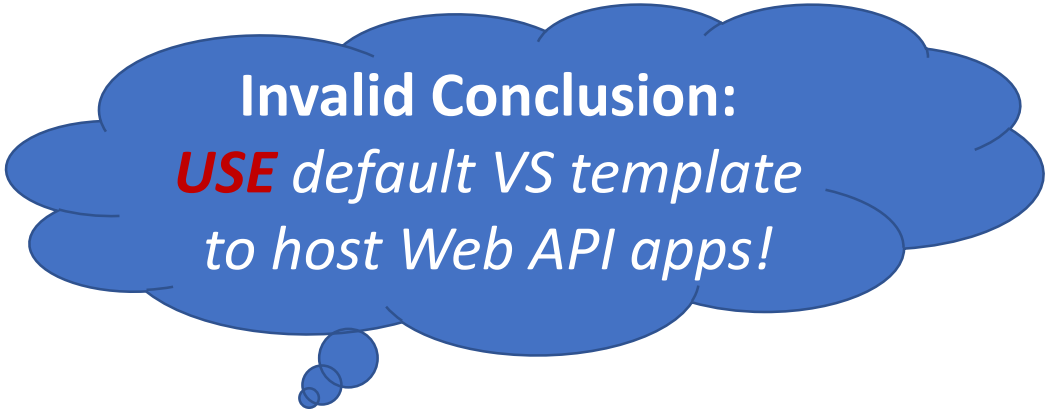


Coupling Web Services to a Host

- Ways in which a Web API service can be **coupled** to its host
  - IIS
  - ASP.NET
  - WCF
  - Windows

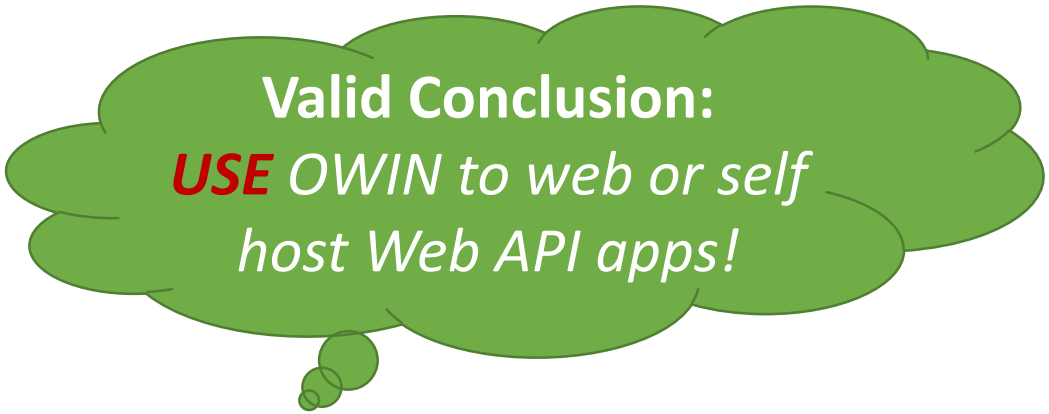
Case in Point: Outdated Web API Hosting

- Visual Studio templates for Web API are **outdated**
  - **Web-hosting** coupled to IIS and **ASP.NET**
  - **Self-hosting** coupled to **WCF**!
- Hosting option determines **security** configuration
  - Problems applying security with **mixed** UI and services



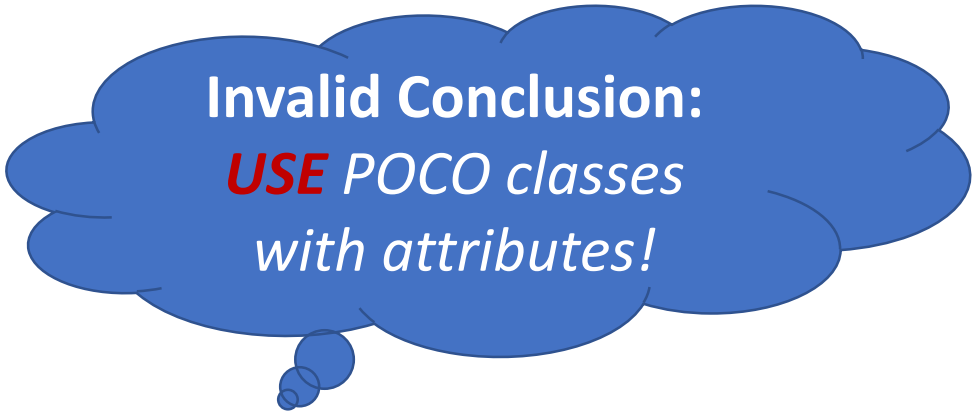
Case in Point: Updated Web API Hosting

- OWIN spec allows greater **decoupling** between web apps and their host
- Middleware is configured in a **Startup** class
- Self-hosting does not rely on **WCF**
- Easier migration to **ASP.NET 5** (vNext)



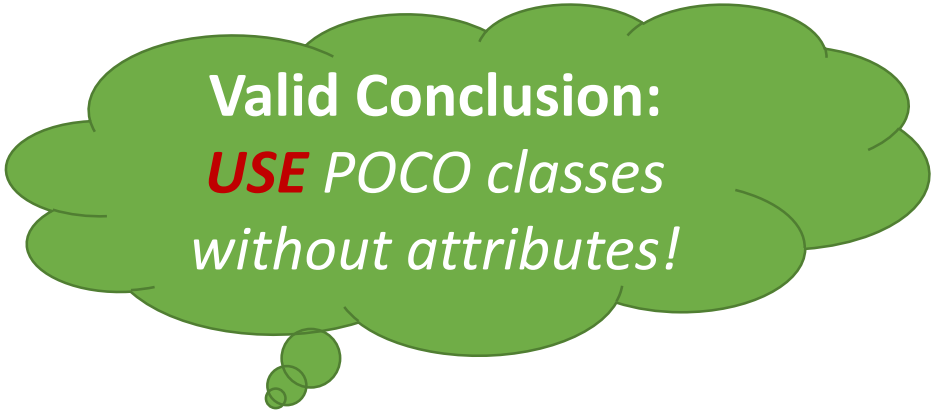
Case in Point: Dirty POCO Classes

- Entities which serve as **Data Transfer Objects** should be stripped of all persistence concerns
- Entities sometimes contain **serialization** attributes
- EF-generated entities decorate properties with **Data Annotation** attributes



Case in Point: Clean POCO Classes

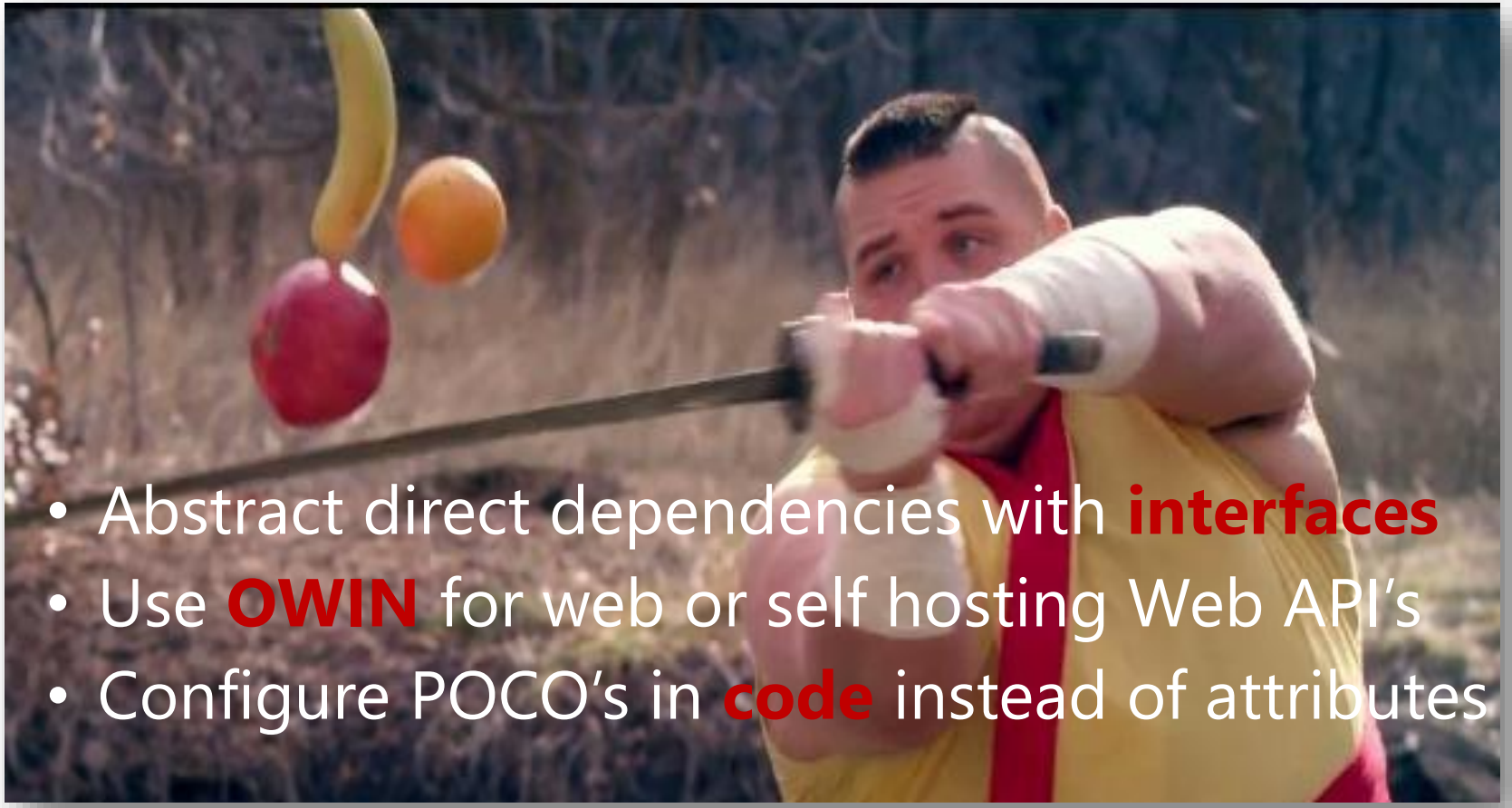
- Customize **T4 templates** for generating POCO's
- Extend generated classes with separate **partial classes** and **partial methods**
- Configure Json and Xml **serializers** in code
- Use **fluent validation** to encapsulate rules



Pitfall #3: How to Mitigate the Risk?

- Use **interfaces** to abstract from an app from infrastructure concerns (data, logging, etc)
- Use **Repository** and **Unit of Work** patterns
- Declare dependencies via **constructors**
- Use **Dependency Injection** for greater flexibility
- Use **OWIN** for Web API hosting today for easier **migration** to ASP.NET 5 tomorrow
- Separate POCO classes from serialization and validation **attributes**

Conclusion: Cut the Ties that Bind



Pitfall #4

One-Size-Fits-All

Approach to Testing

Why We Don't Test

- There never seems to be **enough time** ($, €, £)
- Under-estimate system **complexity**
- Code not written to be **testable**
  - Direct **dependencies** on databases or web services
  - Lots of **static** or **sealed** classes
- Tests are **poorly written**
  - **Inter-dependencies** among tests
  - Not designed to run in **parallel**
  - Not using a **mocking** framework

Why Write Unit Tests?

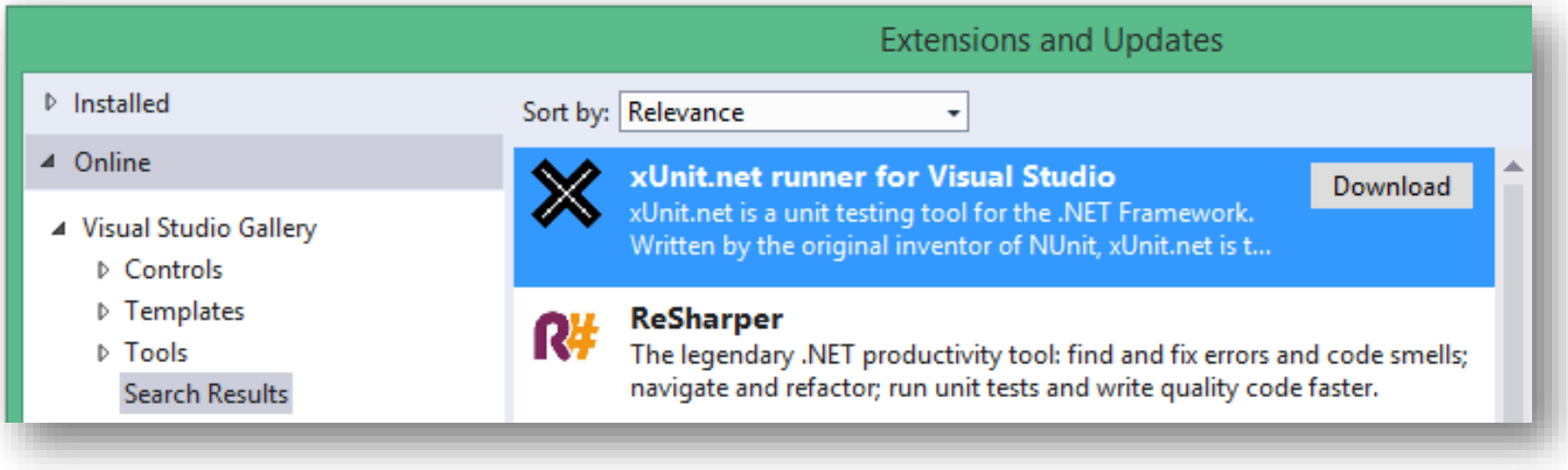
- Document **expected** behaviors
- Validate bug **fixes**
- Ensure a bug fix doesn't **break** something else

Pick Your Poison: Testing Approaches

- Plain Old Unit Testing (**POUT**-ing)
  - Write tests **after** writing code
  - Focus is on defect discovery
- Defect Driven Testing (**DDT**)
  - Fix a defect by writing a **failing test**
  - Normal part of both POUT and TDD
- Test Driven Development (**TDD**)
  - Define how piece of code is **expected to behave**
  - **Refactoring** is an integral part of the process
- Behavior Driven Development (**BDD**)
  - Define **acceptance tests** for features: Given-When-Then

Testing and Mocking Frameworks

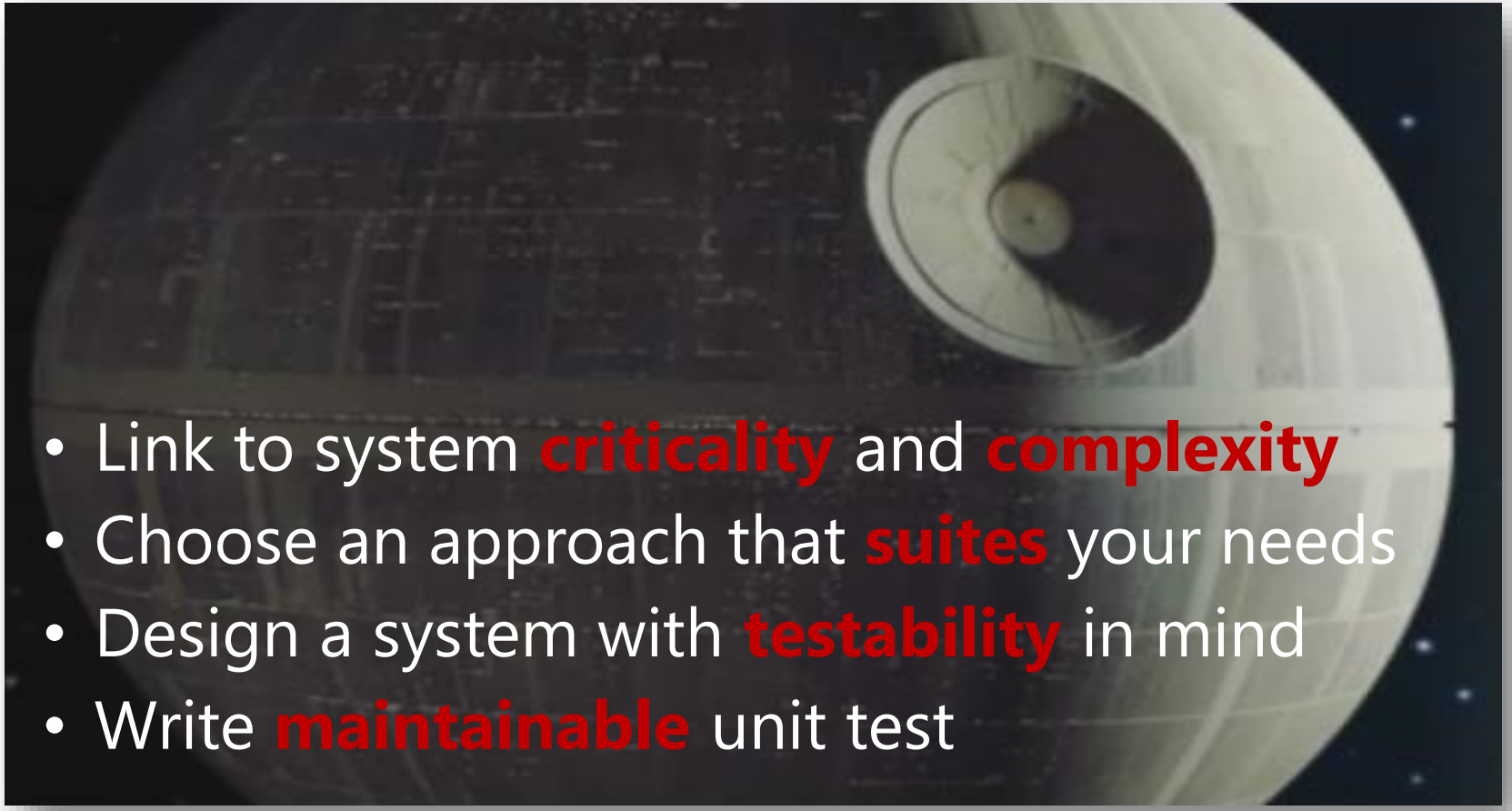
- **xUnit.Net** is the de facto standard for ASP.NET
- **Moq** or **Rhino Mocks** are popular for mocking
- Test **runners** include VS Test Explorer, ReSharper



Pitfall #4: How to Mitigate the Risk?

- Avoid **all-or-nothing** approach
- Select a testing method **suited** to your needs
  - POUT, DDT, TDD, or a **combination**
- Consider a **phased** approach
  - **Mission-critical** algorithms require more rigor
- Follow guidelines for writing **effective** tests
- Don't forget about **integration** tests

Conclusion: Select a Testing Strategy



- Link to system **criticality** and **complexity**
- Choose an approach that **suites** your needs
- Design a system with **testability** in mind
- Write **maintainable** unit test

Pitfall #5

Being Closed to

Open Source

Reinventing the Wheel

Reasons for Using Open-Source

- **Costs**: Hey, it's free!
- **Security**: Hey, it's open!
- **Quality**: Community review and testing
- **Agility**: Don't like it? You fix it!

Reasons for Not Using Open-Source

- When proprietary software has better **support**
- When **hardware** requires proprietary software
- When you need certain types of **warranties**
- When choice of **vendor** is a factor

The World Has Changed

- Microsoft's **use** of **open-source** in Visual Studio:

Microsoft Goes Open-Source

- Microsoft's **open-source** projects on **GitHub**:
  - **CoreCLR**: x-plat runtime
  - **CoreFX**: x-plat framework libraries
  - **Roslyn**: C# compiler platform
  - **Entity Framework**: object/relational mapper
  - **ASP.NET 5**, MVC 6, SignalR: web platform

It's More Than Just Code

- Design meeting **notes**
- Weekly community **standups**
  - Google **hangouts** streamed live on **YouTube**
- GitHub **issues** and **pull requests**
- Kanban style **task board** via Huboard
- Real-time **collaboration** via JabbR

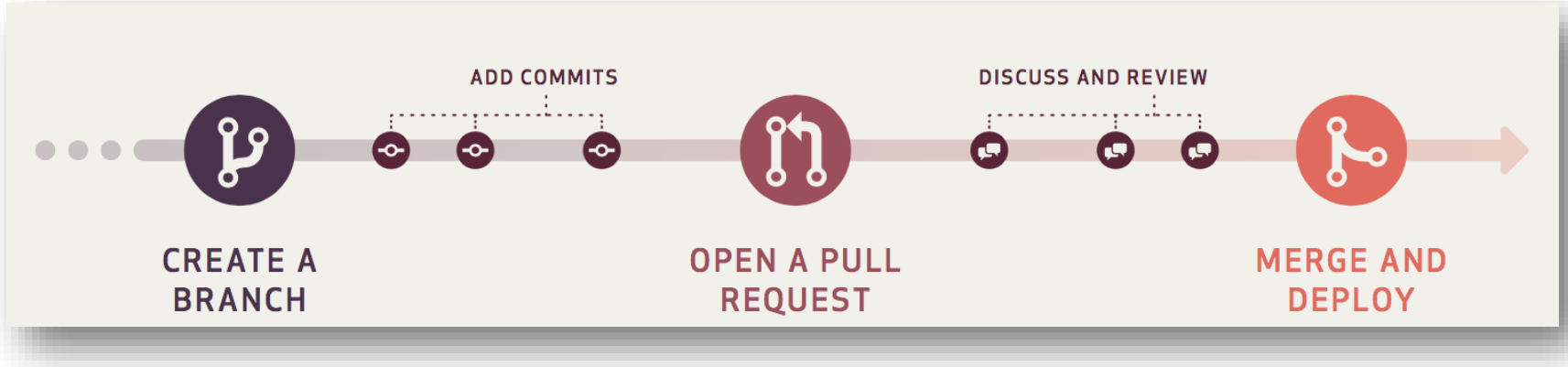
Get to Know Git

- Git is a **distributed** version control system
- Git makes **branching** a first-class citizen
- Fair to say that Git has **replaced** TFS & SVN for source code control
  - Even TFS **supports Git** for version control!
- **Visual Studio** is a great Git client
  - Check out the **Team Explorer** tab
  - Some Git tasks still performed on the **command line**

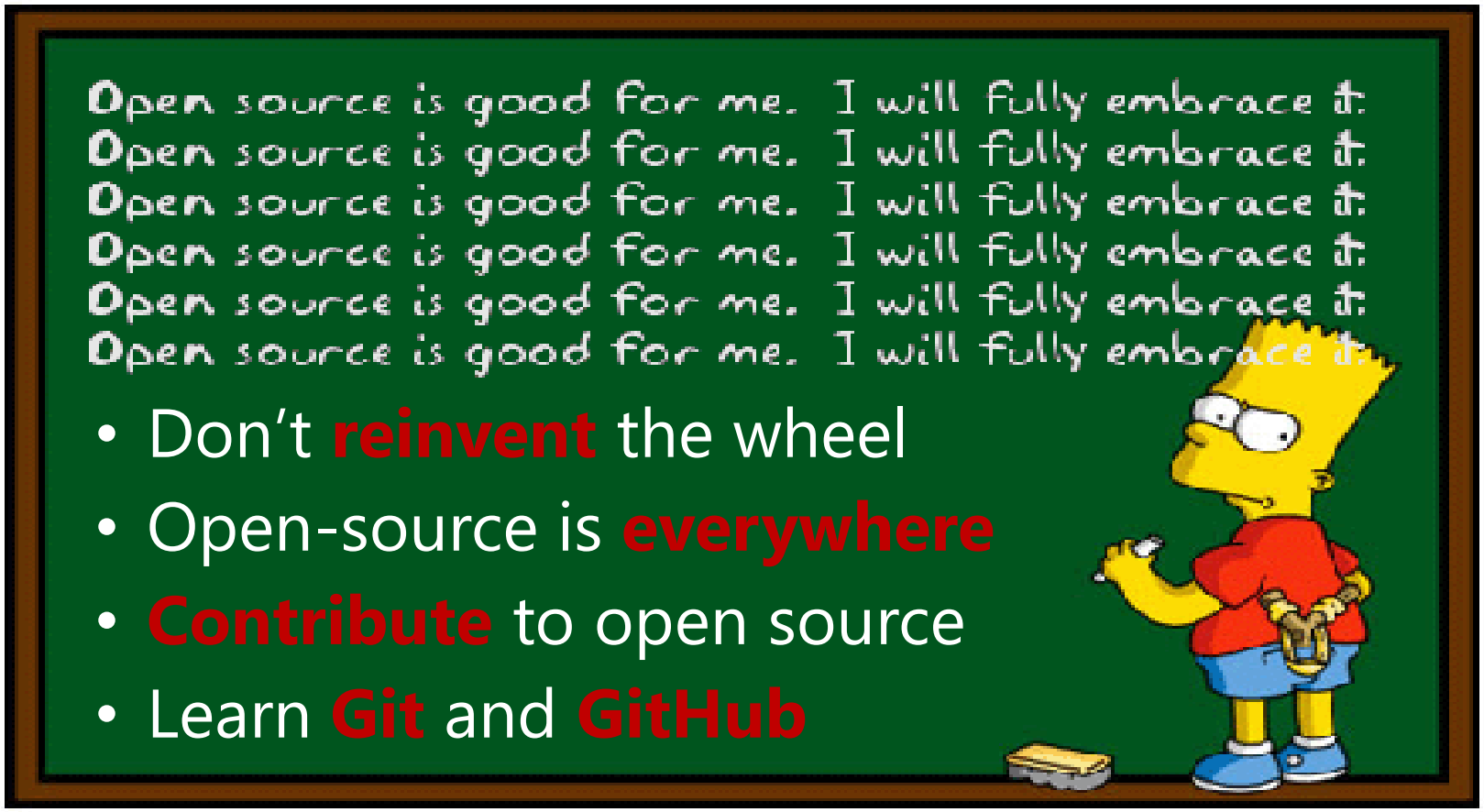
Get On GitHub

GitHub

- GitHub makes learning Git **workflows** easier
  - Fork, branch, commit, pull request, review, deploy, merge
- **Visual Studio** has a GitHub extension



Conclusion: Resistance Is Futile



Recap: Common Architectural Pitfalls

1. Relying on **deprecated** technologies
2. Blindly adopting "**what's hot now**"
3. Failing to apply the right level of **abstraction**
4. One-size-fits-all approach to **testing**
5. Being closed to **open source**

Join In

- Twitter
  - @tonysneed
  - #sapitfalls
- Dropbox
  - bit.ly/pitfalls
- GitHub
  - tonysneed/ SoftwareArchitect.Pitfalls