



Visual Studio **LIVE!** | San Diego
EXPERT SOLUTIONS FOR .NET DEVELOPERS

Unit Testing & Test-Driven Development for Mere Mortals

Benjamin Day
@benday

 Code Again for the First Time!
@benday | www.benday.com

 Microsoft Most Valuable Professional

 Visual Studio 25 YEARS OF CODING INNOVATION

Benjamin Day

Brookline, MA
Consultant & Trainer
Scrum, DevOps,
Team Foundation Server,
Software Architecture & Testing
Microsoft MVP
Pluralsight Author
Scrum.org Trainer
@benday



Benjamin Day Consulting



PLURALSIGHT



Professional
Scrum Trainer
Scrum.org



@benday | www.benday.com



PLURALSIGHT

Pluralsight.com

Online courses

DevOps with TFS2017

Scrum Master Skills

DevOps Skills for Developers with Visual Studio & TFS 2015	by Benjamin Day	Intermediate	7h 7m	19 Sep 2016
Scrum Master Skills	by Benjamin Day	Advanced	4h 7m	06 Aug 2015
Load Testing with Visual Studio 2013	by Benjamin Day	Intermediate	3h 7m	05 Dec 2014
Real World Scrum With Team Foundation Server 2013	by Benjamin Day	Intermediate	5h 2m	10 Nov 2014
Load Testing with Visual Studio 2012	by Benjamin Day	Intermediate	4h 3m	26 Aug 2013
ALM for Developers with Visual Studio 2012	by Benjamin Day	Intermediate	4h 31m	23 Sep 2012
ALM with TFS 2012 Fundamentals	by Benjamin Day	Intermediate	5h 34m	04 Sep 2012



@benday | www.benday.com

Getting Started with Visual Studio Team Services (VSTS)



PLURALSIGHT

Architecting an ASP.NET Core MVC Application for Unit Testability

Coming in ~November 2018



On with the show.

Overview

What is a unit test?

What is unit testing?

Why do I care?

How do I justify it to my self/boss/team?

Design for testability

Design Patterns

Demos



@benday | www.benday.com

**Unit tests =
Small chunks of code that test other
small chunks of code**

Typical Unit Testing Flow

You're writing a feature

You're also writing tests

Feature code = the "System Under Test"

- (aka. "SUT")

At least one unit test method per public method on the SUT

- Asserts



@benday | www.benday.com

Demo: Calculator Unit Tests

Why use unit tests?

Typical Development Flow

You write code
It appears to work
You check it in
You send it to QA
QA sends back bugs
You fix
(repeat)



@benday | www.benday.com

Typical Refactoring Flow

You refactor code

You write some new code, too

You check it in

It appears to work

You send it to QA

They send bugs

You fix bugs

You deploy to production

The world lights on fire

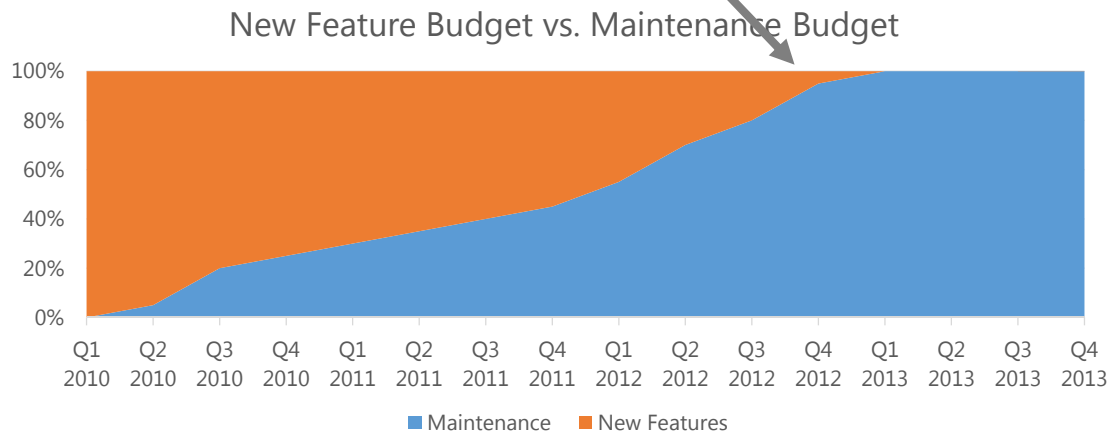


@benday | www.benday.com

\$\$\$

\$1m for IT to support an application.

That's looking grim.



@benday | www.benday.com

Technical Debt

Technical Debt

Little bits of not-quite-done

Low-quality code

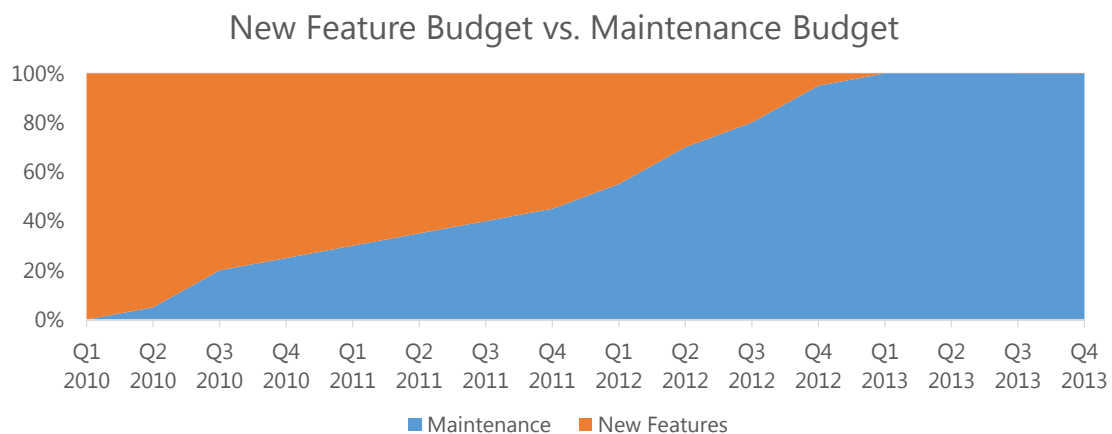
- 8 million line methods
- Coding standards violations

Classes with unclear names and intent

Buggy & Brittle

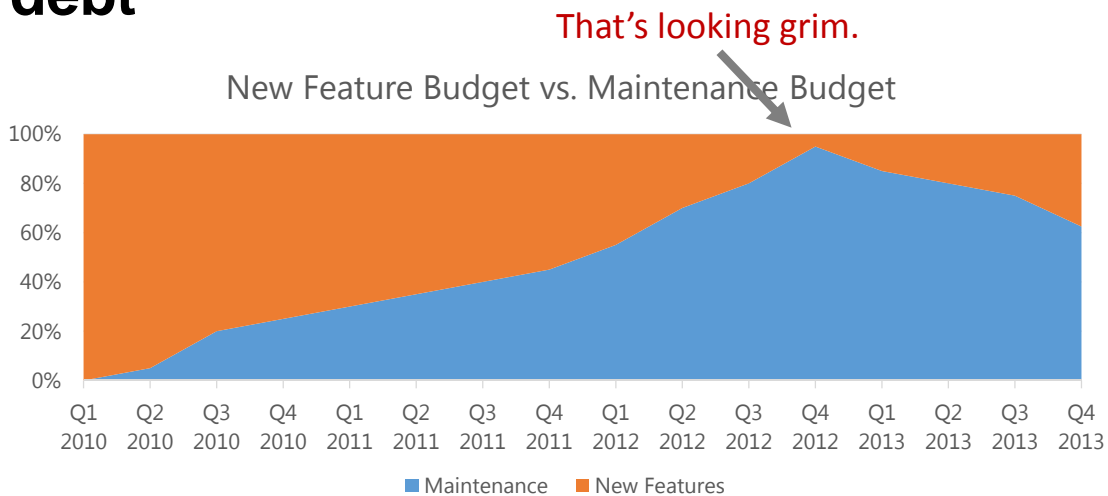
BDC @benday | www.benday.com

Unit tests can help tackle technical debt



BDC @benday | www.benday.com

Unit tests can help tackle technical debt



BDC @benday | www.benday.com

Unit Tests

Proof that the code you're writing actually works

Run-able by a computer

Proof that the code you wrote a long time ago still works

Automated regression tests

Proof that bugs are still fixed

BDC @benday | www.benday.com

Awesome side effects of unit tests

Focus on quality early

- Don't rely on QA to tell you your code is broken

Fewer low-value bugs coming back from QA

- `NullReferenceException`

QA can focus on "exploratory testing"

- Better use of QA's time

Refactor with confidence

- Keep your code clean

Clean code is easier to read

- % time reading code vs. % time writing code

Clean code is easier to build on top of



@benday | www.benday.com

How I structure a unit test

Unit test class tests a class in the system under test

- `{NameOfSystemUnderTestClass}Fixture.cs`

Has a property called `SystemUnderTest`

- `Typeof {NameOfSystemUnderTestClass}`

[`TestInitialize`] method called `OnTestInitialize()`

One or more [`TestMethod`]

- Arrange
- Act
- Assert



@benday | www.benday.com

Unit Tests vs. Integration Tests

How would you test this?



@benday | www.benday.com

What is Design For Testability?

Build it so you can test it.



How would you test this?

Do you have to take the plane up for a spin?



@benday | www.benday.com

Interfaces, Dependency Injection, & Mocks

Code against interfaces

Dependency Injection

- "Advertise" your dependencies on the constructor
- Dependencies are references to interfaces

Use mock classes to provide fake data



@benday | www.benday.com

Advertise Dependencies on Constructor

Less Awesome

```
public class PersonManagerWithoutDI
{
    private IPersonRepository m_Repository;

    public PersonManagerWithoutDI()
    {
        // create an instance of IPersonRepository
        m_Repository = new SqlPersonRepository();
    }

    private void Save(IPerson saveThis)
    {
        Validate(saveThis);

        m_Repository.Save(saveThis);
    }

    private void Validate(IPerson saveThis)
    {
        // validate it
    }
}
```



@benday | www.benday.com

Now With More Awesome

```
public class PersonManagerWithDI
{
    private IPersonRepository m_Repository;

    public PersonManagerWithDI(IPersonRepository instance)
    {
        if (instance == null)
        {
            throw new ArgumentNullException("instance",
                "instance is null.");
        }

        m_Repository = instance;
    }

    private void Save(IPerson saveThis)
    {
        Validate(saveThis);

        m_Repository.Save(saveThis);
    }

    private void Validate(IPerson saveThis)
    {
        // validate it
    }
}
```

Hard to test usually also means
hard to maintain.

Design Patterns will help you to create a more testable & maintainable application.

What's a Design Pattern?

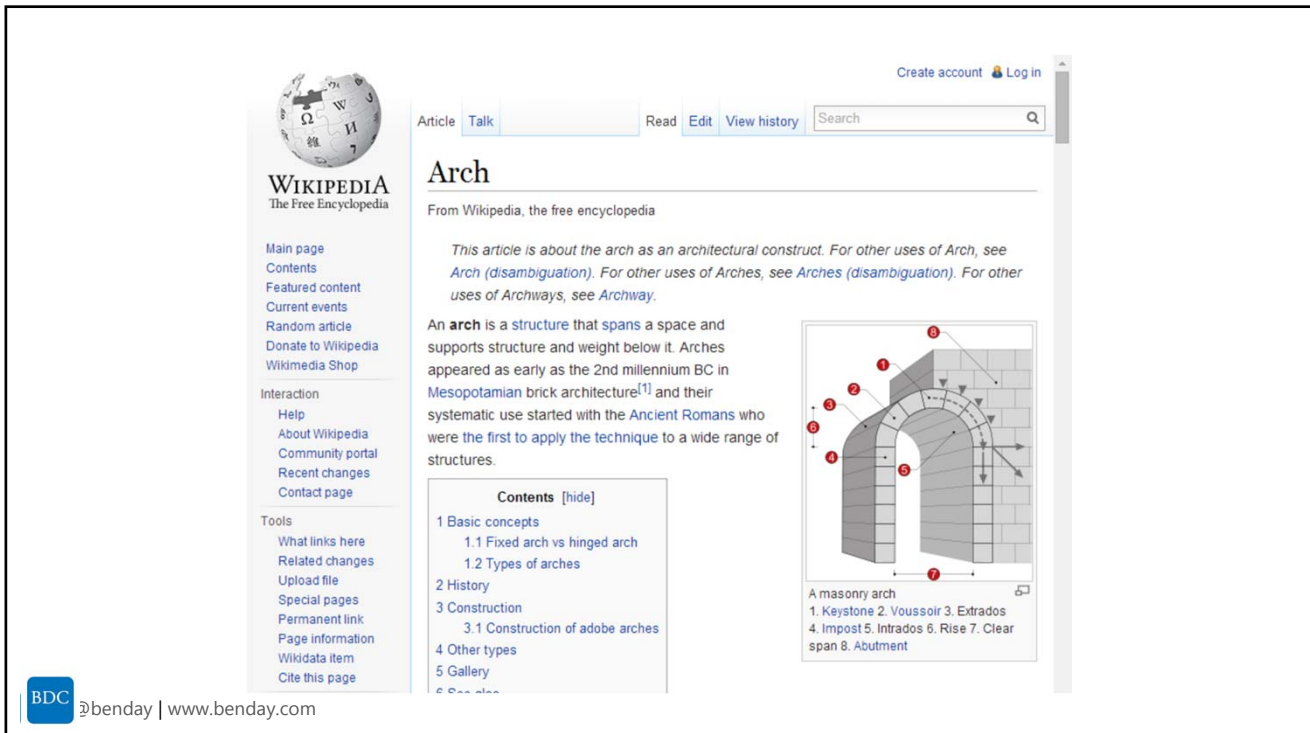
Well-known and accepted solution to a common problem

Avoid re-inventing the wheel



@benday | www.benday.com

Design patterns in architecture.



The screenshot shows the Wikipedia article for "Arch". The page includes the Wikipedia logo, navigation links, and a search bar. The article text defines an arch as a structure that spans a space and supports structure and weight below it. It mentions that arches appeared as early as the 2nd millennium BC in Mesopotamian brick architecture and that their systematic use started with the Ancient Romans. A diagram of a masonry arch is shown with numbered parts: 1. Keystone, 2. Voussoir, 3. Extrados, 4. Impost, 5. Intrados, 6. Rise, 7. Clear span, 8. Abutment. The diagram illustrates the structure of a masonry arch, showing the keystone at the top, the voussoirs (wedge-shaped stones) forming the arch, the extrados (outer curve), the impost (base of the arch), the intrados (inner curve), the rise (height), the clear span (width), and the abutment (support).

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Create account Log in

Article Talk Read Edit View history Search

Arch

From Wikipedia, the free encyclopedia

This article is about the arch as an architectural construct. For other uses of Arch, see Arch (disambiguation). For other uses of Arches, see Arches (disambiguation). For other uses of Archways, see Archway.

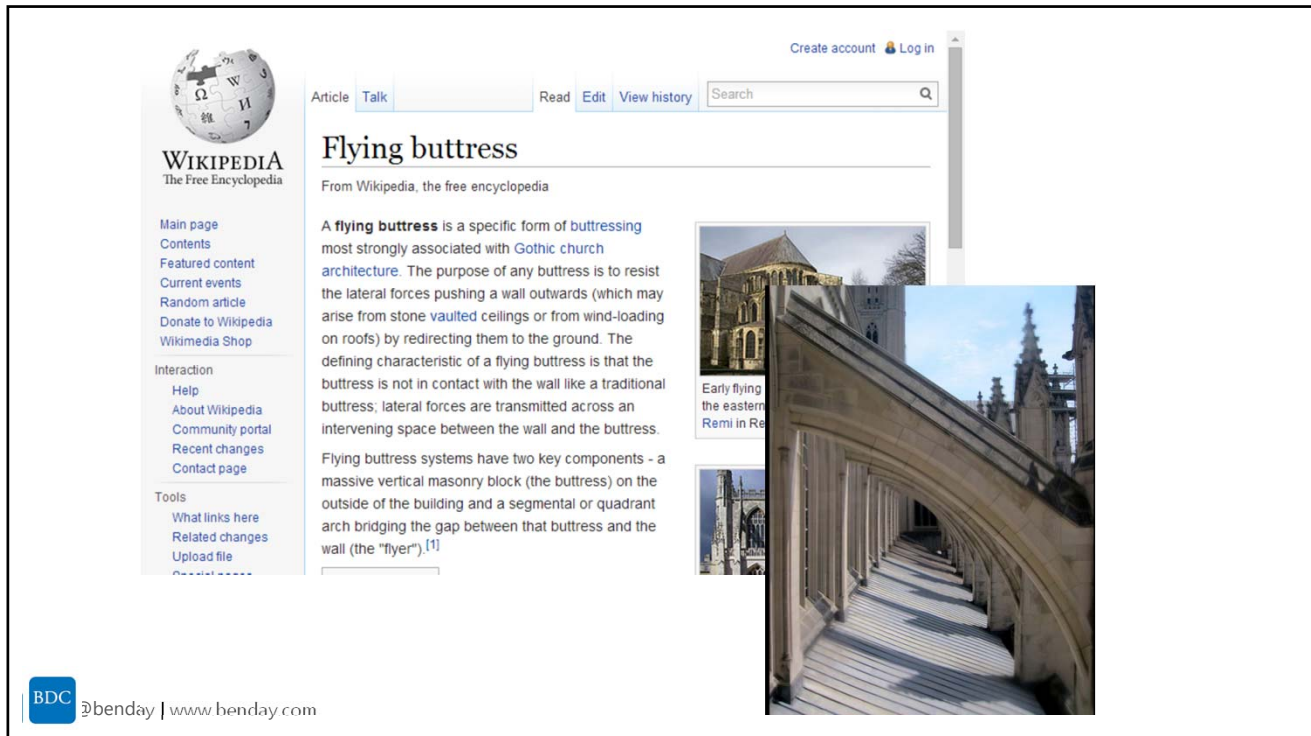
An **arch** is a structure that spans a space and supports structure and weight below it. Arches appeared as early as the 2nd millennium BC in Mesopotamian brick architecture^[1] and their systematic use started with the **Ancient Romans** who were the first to apply the technique to a wide range of structures.

Contents [hide]

- 1 Basic concepts
 - 1.1 Fixed arch vs hinged arch
 - 1.2 Types of arches
- 2 History
- 3 Construction
 - 3.1 Construction of adobe arches
- 4 Other types
- 5 Gallery
- 6 See also

A masonry arch
1. Keystone 2. Voussoir 3. Extrados
4. Impost 5. Intrados 6. Rise 7. Clear span 8. Abutment

BDC @benday | www.benday.com



Popularized in Software by this book...



Common Design Patterns for Testability

Dependency Injection

- Flexibility

Strategy

- Encapsulates algorithms & business logic

Repository

- Data access details

Model-View-Controller

- Isolates User Interface *Implementation* from the User Interface *Logic*
- Testable User Interfaces
- Model-View-ViewModel

Adapter

- Keeps tedious, bug-prone code contained



@benday | www.benday.com

Repository Pattern

Hide the existence of databases and services

Hide the existence of Entity Framework & ADO.NET

Code against an interface

Allows you to use mock data without an actual database

Separate Unit Tests from Integration Tests



@benday | www.benday.com

Adapter Pattern

Isolate the details of turning one object into another object

To / from Entity Framework entities

To / from WebAPI message types

To / from ASP.NET MVC ViewModels and Models



@benday | www.benday.com

Model-View-Controller (MVC)

Test user interfaces without having to run an actual user interface

ASP.NET MVC

WPF / XAML

- Model-View-ViewModel (MVVM)



@benday | www.benday.com

Strategy Pattern

Isolate and test algorithms

Validation

Business rules



@benday | www.benday.com

Demos

Any last questions?

Thank you.



Benjamin Day Consulting

www.benday.com | benday@benday.com