Better Unit Tests through Design Patterns: Repository, Adapter, Mocks, and more…

Benjamin Day

Benjamin Day

- Brookline, MA
- Consultant, Coach, & Trainer
- Microsoft MVP for Visual Studio ALM
- Scrum, Software Testing & Architecture, Team Foundation Server
- Scrum.org Trainer
  - Professional Scrum Developer (PSD)
  - Professional Scrum Foundations (PSF)
  - Professional Scrum Master (PSM)
- Scrum.org Evidence-Based Management Engagement Manager
- Pluralsight.com Author
- www.benday.com, benday@benday.com, @benday

Online courses at Pluralsight.com

BDC

Scrum with TFS 2013 coming soon.

BDC

On with the show.

Agenda / Overview

- Quick Review
- Design for Testability
- What's a Design Pattern?
- Design Patterns for Testability
- Patterns for User Interface Testing
  - Server-side web apps
  - JavaScript apps

BDC

This talk is about unit testing & test-driven development.

BDC

What is Test-Driven Development?

- Develop code with proof that it works
  - Code that validates other code
  - Small chunks of “is it working?”

- Small chunks = Unit Tests

- “Never write a single line of code unless you have a failing automated test.”
  - Kent Beck, “Test-Driven Development”, Addison-Wesley

BDC

Why Use TDD?

- High-quality code
  - Fewer bugs
  - Bugs are easier to diagnose
- Encourages you to think about…
  - …what you're building
  - …how you know you're done
  - …how you know it works
- Less time in the debugger
- Tests that say when something works →
  - Easier maintenance, refactoring
  - Self-documenting
- Helps you to know if it's working a lot faster.
- Tends to push you into better/cleaner architecture.

BDC

You shouldn’t need QA to
tell you that your stuff doesn’t work.

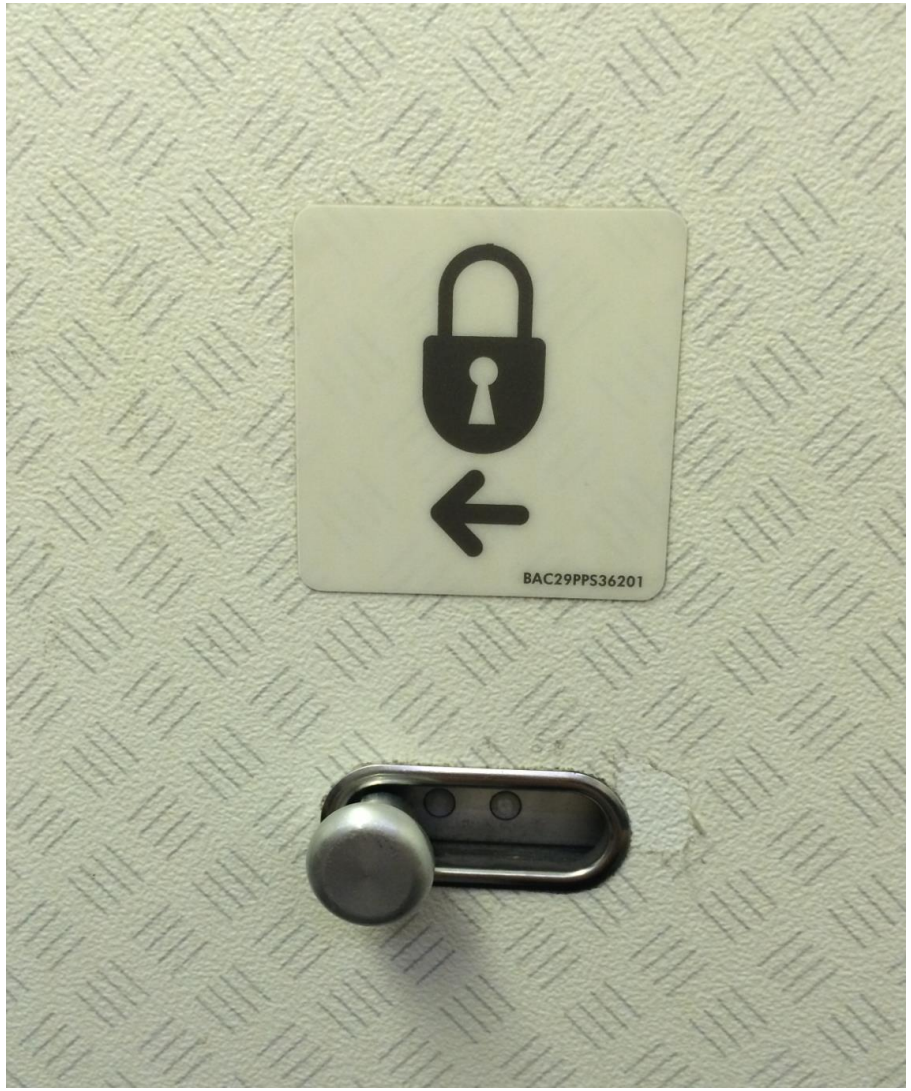
BDC

Your apps need to be tested.

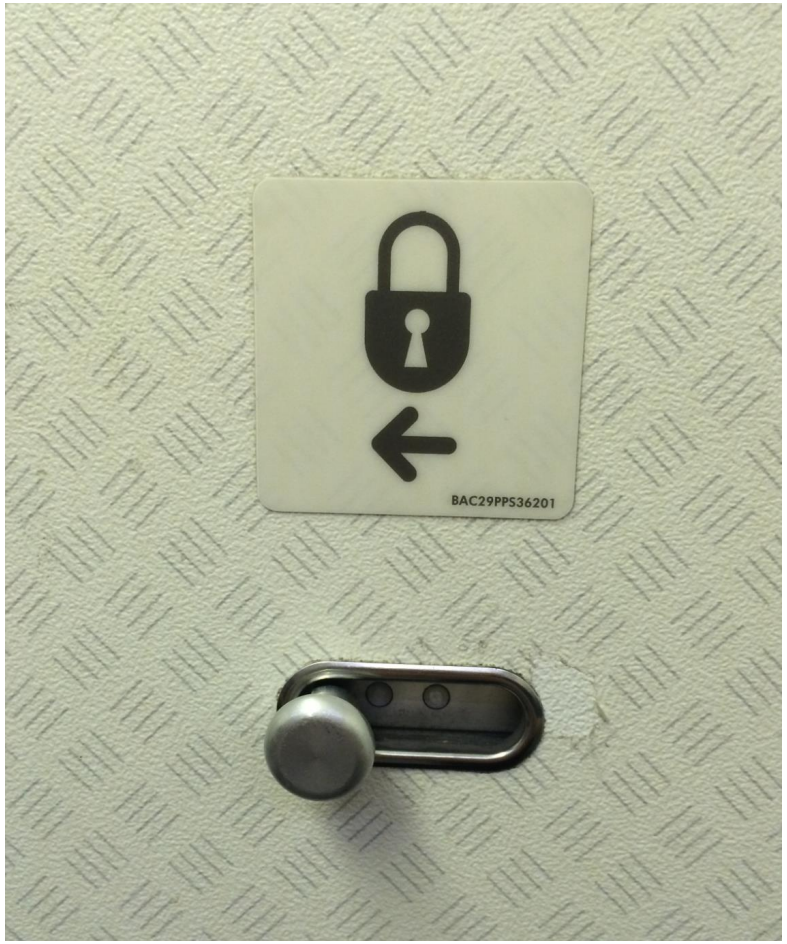
BDC

Your apps need to be testable.

How would you test this?



What is Design For Testability?



- How would you test this?
- Do you have to take the plane up for a spin?
- Build it so you can test it.

BDC

Your apps need to be testable.
You need to design for testability.

A *unit* test is not the same as an *integration* test.

BDC

Avoid End-to-End Integration Tests

Does a good test…

- …really have to write all the way to the database?
- …really have to have a running REST service on the other end of that call?
- …really need to make a call to the mainframe?

BDC

It’s called a *unit* test.

- Small units of functionality
- Tested in isolation
- If you designed for testability, you (probably) can test in isolation.
- If you *didn’t*, you probably have a monolithic app.

BDC

“How am I supposed to test *THAT*?!”

http://www.pdphoto.org/PictureDetail.php?mat=&pg=8307

PUBLIC DOMAIN

It’ll be a lot easier if you design for testability.

BDC

What makes an app hard to test?

- Tightly coupled
- Hidden or embedded dependencies
- Required data & databases
- Insane amounts of setup code for the test

BDC

Hard to test usually also means hard to maintain.

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

BDC

What’s a Design Pattern?

- Well-known and accepted solution to a common problem
- Avoid re-inventing the wheel

BDC

Design patterns in architecture.

BDC

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

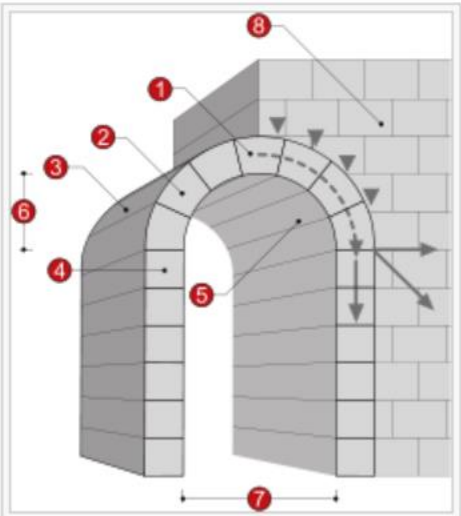
Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

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.



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

Contents [hide]

BDC

Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk Read Edit View history Search

Flying buttress

From 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

A **flying buttress** is a specific form of buttressing most strongly associated with Gothic church architecture. The purpose of any buttress is to resist the lateral forces pushing a wall outwards (which may arise from stone vaulted ceilings or from wind-loading on roofs) by redirecting them to the ground. The defining characteristic of a flying buttress is that the buttress is not in contact with the wall like a traditional buttress; lateral forces are transmitted across an intervening space between the wall and the buttress.

Flying buttress systems have two key components - a massive vertical masonry block (the buttress) on the outside of the building and a segmental or quadrant arch bridging the gap between that buttress and the wall (the "flyer").[1]

Early flying
the eastern
Remi in Re

Popularized in Software by this book…

BDC

Design Patterns for this talk

- Dependency Injection
  - Flexibility
- Repository
  - Data Access
- Adapter
  - Single-Responsibility Principle
  - Keeps tedious, bug-prone code contained
- Strategy
  - Encapsulates algorithms & business logic
- Model-View-Controller
  - Isolates User Interface *Implementation* from the User Interface *Logic*
  - Testable User Interfaces
  - Model-View-ViewModel

BDC

Design goals in a testable system

- (Testable, obviously.)
- Well-organized
- Flexible

BDC

Design goals in a testable system

- (Testable, obviously.)
- Well-organized
  - Single Responsibility Principle (SRP)
  - Layered (example: n-tier)
- Flexible
  - Code to interfaces rather than concrete types
  - Dependency Injection
  - Interface Segregation Principle (ISP)

BDC

Single Responsibility Principle:
An object should have only
one reason to change.

BDC

http://lostechies.com/derickbailey/files/2011/03/SingleResponsibilityPrinciple2_71060858.jpg

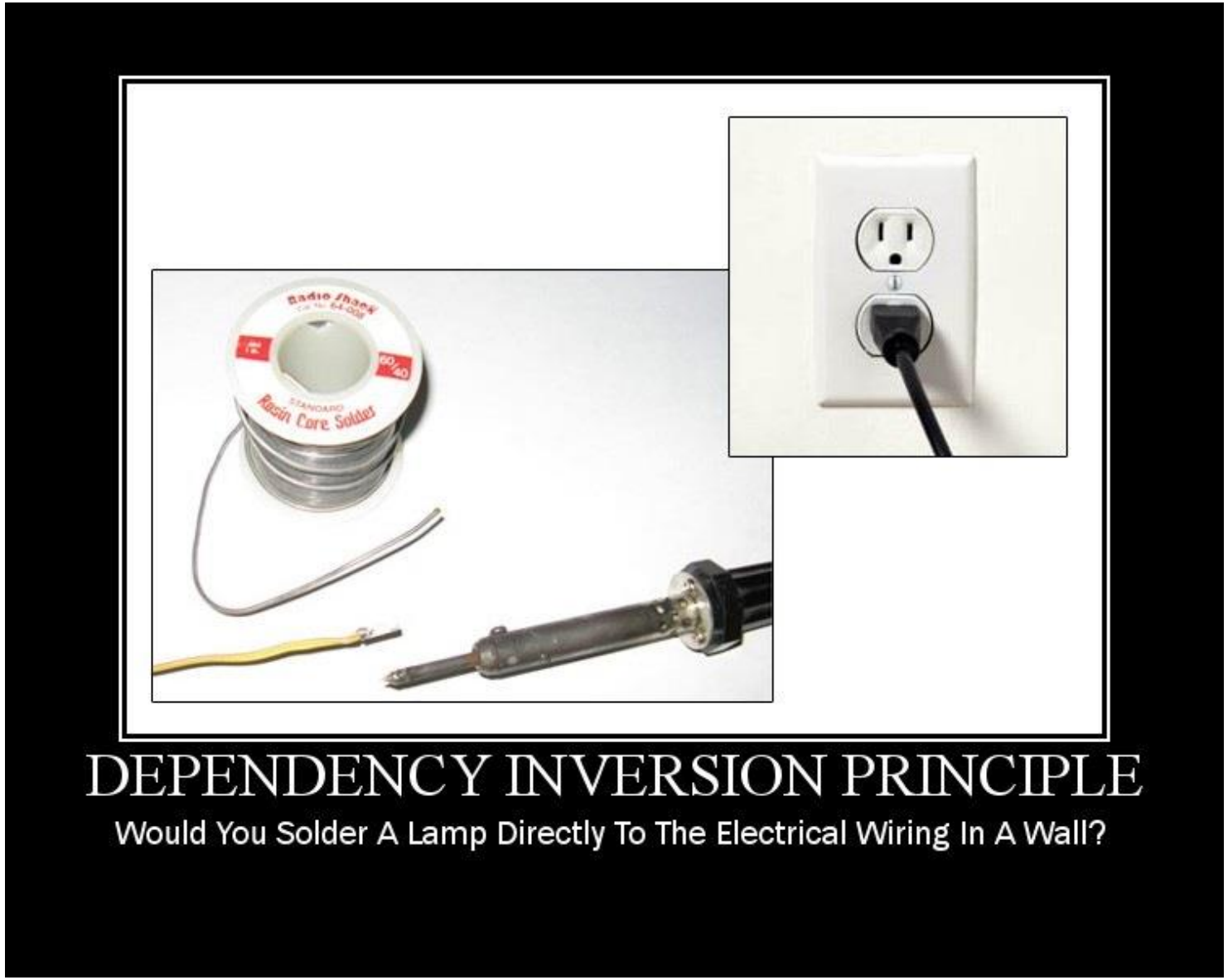
BDC
CC BY SA

Interface Segregation Principle:
“no client should be forced to depend on methods it does not use.”

BDC

Dependency Injection:
“Don’t get too attached.”

BDC



http://lostechies.com/derickbailey/files/2011/03/DependencyInversionPrinciple_0278F9E2.jpg

BDC

CC BY SA

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
    }
}
```

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
    }
}
```

BDC

Why does DI help with testability?

- Helps you focus on the testing task at hand
  - Only test what you're trying to test. Skip everything else.
- Makes interface-driven programming simple
- Interface-driven programming + DI lets you use mocks and stubs in your tests.

BDC

“Mocks & Stubs?”

BDC

Mocks vs. Stubs vs. Dummies vs. Fakes

- Martin Fowler
  http://martinfowler.com/articles/mocksArentStubs.html

- Dummy = passed but not used

- Fake = “shortcut” implementation

- Stub = Only pretends to work, returns pre-defined answer

- Mock = Used to test expectations, requires verification at the end of test

BDC

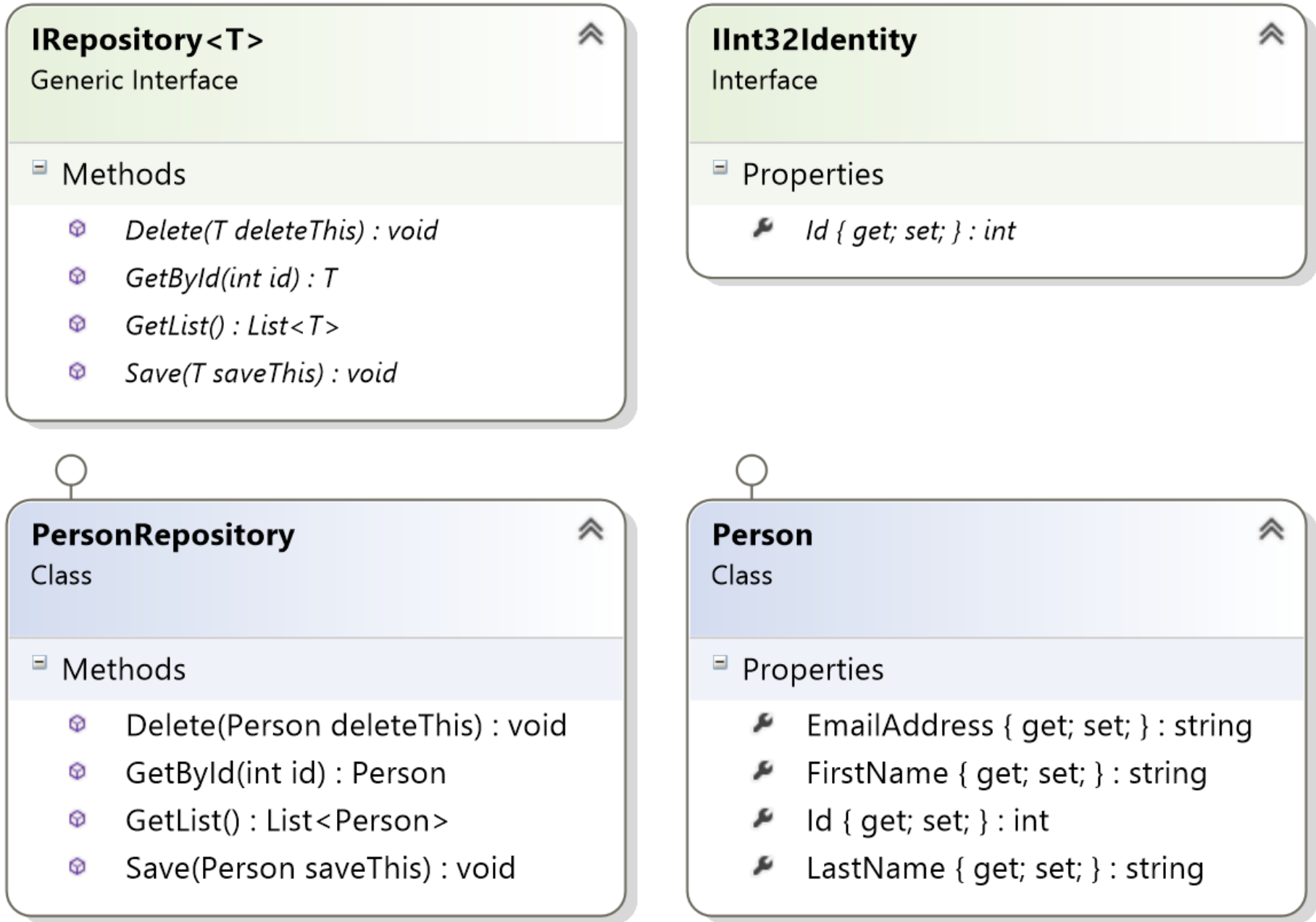
Demos:
PersonService saves
valid person objects with
unique user names.

Strategy Pattern
encapsulates an algorithm
behind an interface.

BDC

Repository Pattern
encapsulates data access logic.

BDC


IRepository<T>
Generic Interface
Methods
Delete(T deleteThis) : void
GetById(int id) : T
GetList() : List<T>
Save(T saveThis) : void
IInt32Identity
Interface
Properties
Id { get; set; } : int
PersonRepository
Class
Methods
Delete(Person deleteThis) : void
GetById(int id) : Person
GetList() : List<Person>
Save(Person saveThis) : void
Person
Class
Properties
EmailAddress { get; set; } : string
FirstName { get; set; } : string
Id { get; set; } : int
LastName { get; set; } : string

BDC

Demo:
Mocks for code coverage

BDC

User Interface Testing

BDC

User Interfaces: The Redheaded Stepchild of the Unit Testing World

- Not easy to automate the UI testing
- Basically, automating button clicks
- UI's almost have to be tested by a human
  - Computers don't understand the "visual stuff"
  - Colors, fonts, etc are hard to unit test for
  - "This doesn't look right" errors
- The rest is:
  - Exercising the application
  - Checking that fields have the right data
  - Checking field visibility

BDC

My $0.02.

- Solve the problem by not solving the problem
- Find a way to minimize what you can’t automate

BDC

The Solution.

- Keep as much logic as possible out of the UI
  - Shouldn’t be more than a handful of assignments
  - Nothing smart
  - Real work is handled by the “business” tier

- Test the UI separate from everything else

BDC

Design Patterns for UI Testability

- Model-View-Controller (MVC)
  - ASP.NET MVC
- Model-View-Presenter (MVP)
  - Windows Forms
  - ASP.NET Web Forms
- Model-View-ViewModel (MVVM)
  - AngularJS
  - Silverlight
  - WPF
  - Windows Phone

BDC

The idea is that the user interface becomes an abstraction.

BDC

Demo:
Search for a president using Model-View-Controller (MVC)

BDC

This is also relevant
in the JavaScript world.

MVC / MVVM with AngularJS, tested by Jasmine

BDC

What is AngularJS?

- JavaScript library for data binding
- Logic goes into Controllers
  - (ViewModel?)
- HTML becomes a thin layer over the Controllers
  - “Views”
- Testing effort is focused on the Controller

BDC

AngularJS is easily, readily tested by Jasmine tests.

Demo:
A simple calculator with AngularJS and Jasmine Tests

BDC

“Ok. Great.
But what about something
useful with data?”

BDC

Tip:
Service-oriented applications
are *two* apps.

BDC

Dependency Injection is built in to AngularJS.

Calls to back-end services get wrapped in classes called “Services”.

BDC

Demo:
President Search with a REST-based service, AngularJS, & Jasmine

BDC

Summary, Part 1: The Patterns

- Dependency Injection
  - Flexibility
- Repository
  - Data Access
- Adapter
  - Single-Responsibility Principle
  - Keeps tedious, bug-prone code contained
- Strategy
  - Encapsulates algorithms & business logic
- Model-View-Controller
  - Isolates User Interface *Implementation* from the User Interface *Logic*
  - Testable User Interfaces
  - Model-View-ViewModel

BDC

Summary, Part 2: The Big Picture

- Quick Review
- Design for Testability
- What's a Design Pattern?
- Design Patterns for Testability
- Patterns for User Interface Testing
  - Server-side web apps
  - JavaScript apps

BDC

Any last questions?

Thanks.

benday@benday.com | www.benday.com