Chicago.NET cnug Users Group

Test-Driven Development

Code testing, mocking, dependency injection, benefits and recent criticism

Mayank Srivastava

Manager and Evangelist

www.MayankSrivastava.com

http://www.mpspartners.com/blogs

@MayankSri

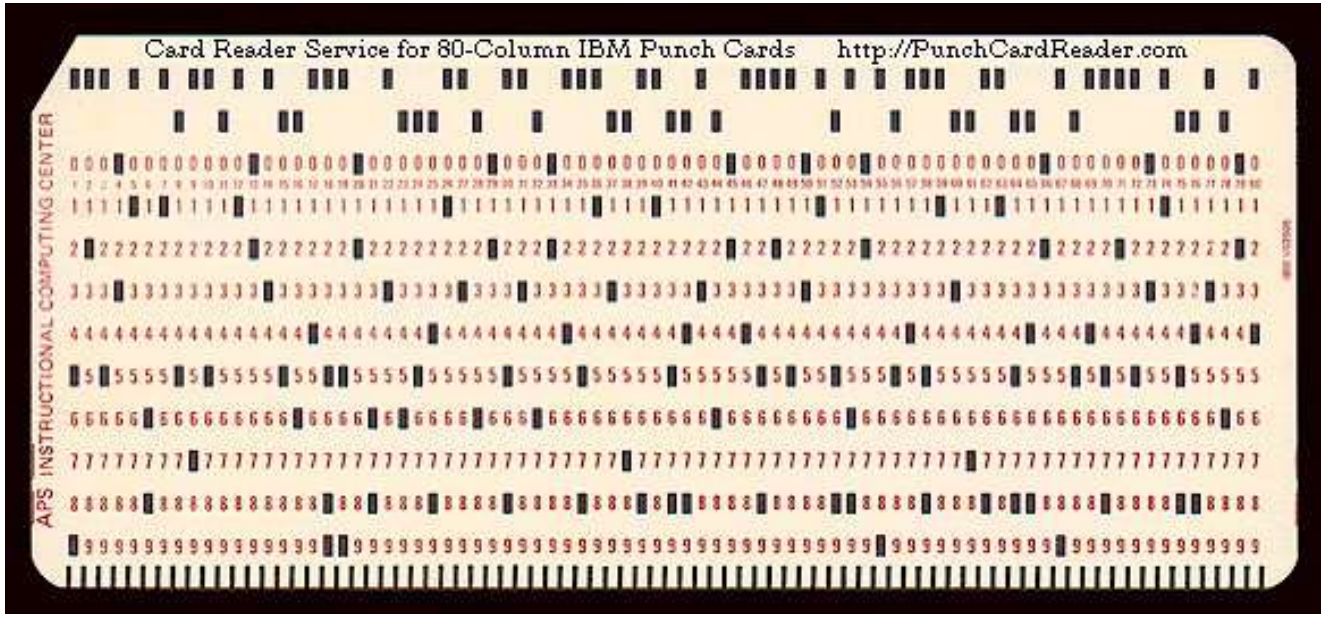
Know your audience

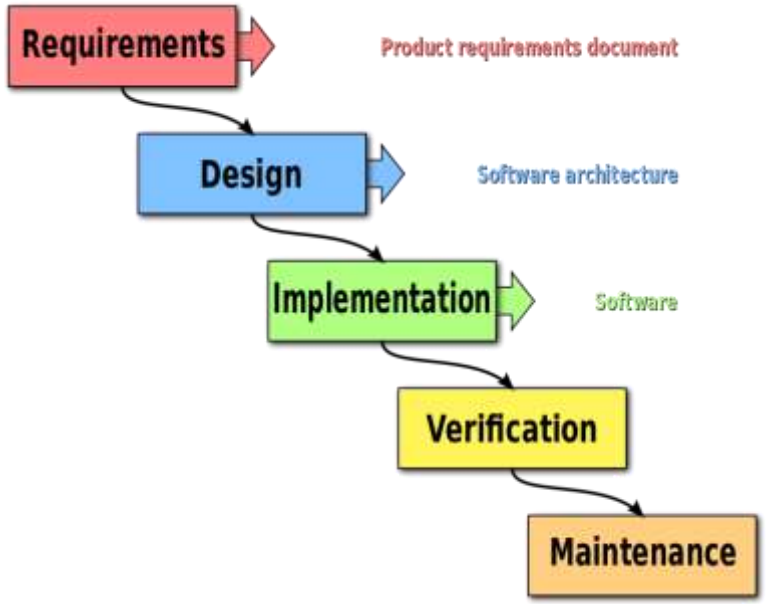
- Didn’t quite use TDD but used Unit / integration tests.
- Used TDD on a project.

What we will cover today:

- TDD concept
  - Unit test
  - Dependency Injection
  - Resource Mocking
  - Integration tests
- Best Practices
- Recent Criticism
- Open forum (Opinions, comments, experiences, questions)

Waterfall - Initial Development methodology





Agile

- Iterative & incremental
  - Small manageable units of work
- Collaboration between self-organizing, cross-functional teams
  - Develop small pieces and put it before the user first.
- Adaptive, evolutionary development & delivery
  - Built to last – rapid and flexible response to change


TDD
ALL C0D3 1S GU1LTY
UNT1L PR0V3N B G FREE

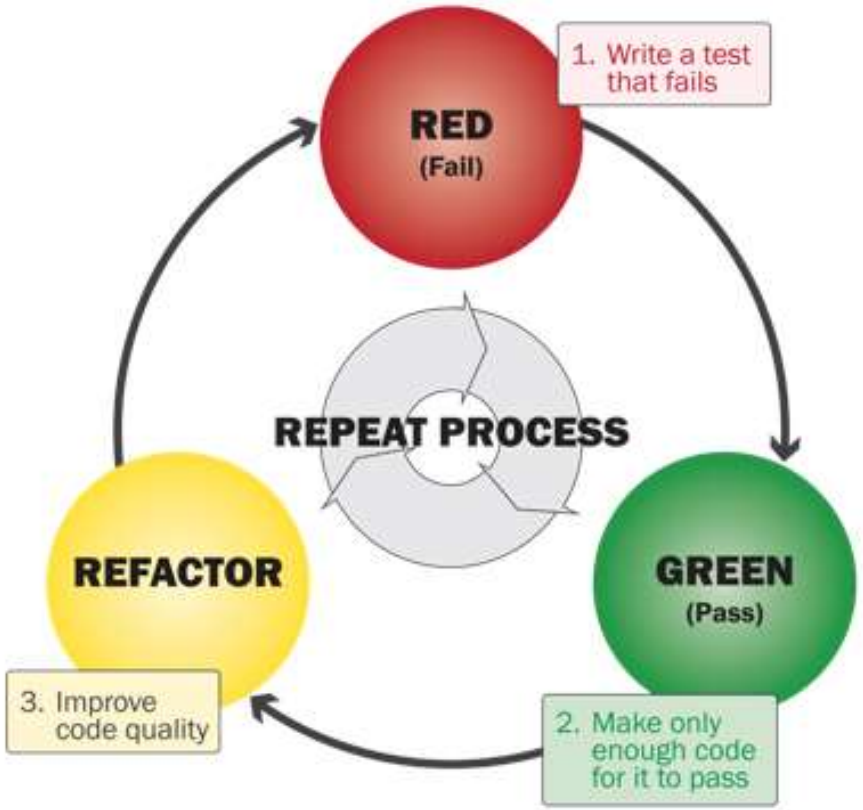
Test-driven development relies on the repetition of a very short development cycle:

First the developer writes an automated test case that defines a desired improvement or new function,

then produces the minimum amount of code to pass that test,

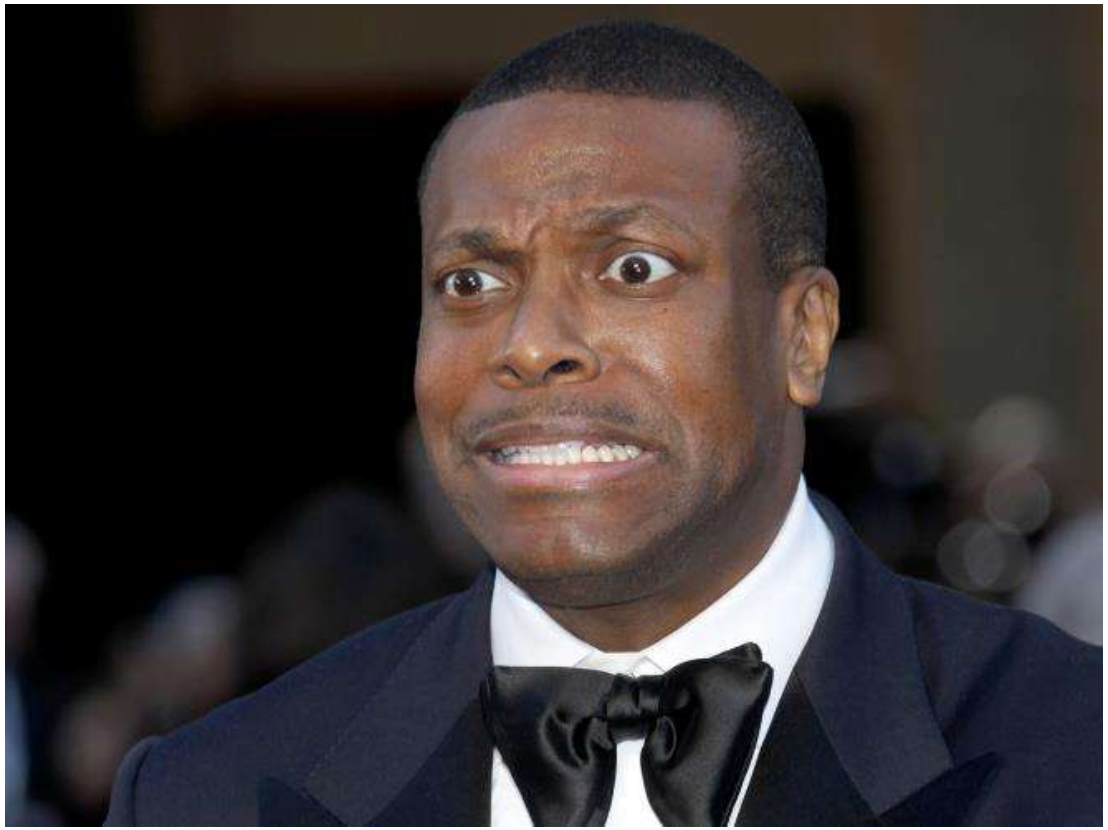
And finally refactors the new code to acceptable standards.

Red – Green - Refactor



In short…

start with testing what you haven’t written yet.



Absolutely!

If you

- Are going to maintain code base developed by other developers.
- Work with developers who do not accept their code has bugs.
- Are going to make changes to someone else's code and then someone else is not available before deployment.
- Spend good amount of time trying to identify what par of the code base is causing the problem.

You need TDD

TDD - key benefits

- Test is written against requirements.
- Provides built-in regression testing.
- Testable code, by design, follows SOLID principals.
- Ensure quality from the start.
- Puts an end to recurring bugs from Waterfall.

Unit Test

A test written to validate “One (logical) Unit” of work.

Unit
- Isolated from other code
- Isolated from other developers

Test
- Targetable
- Repeatable
- Predictable

Decoupling the Unit with Dependencies

You unit of work is dependent on external resources like:

- A persistence medium
- Logging system
- A caching mechanism
- An external service like a web or windows based service.

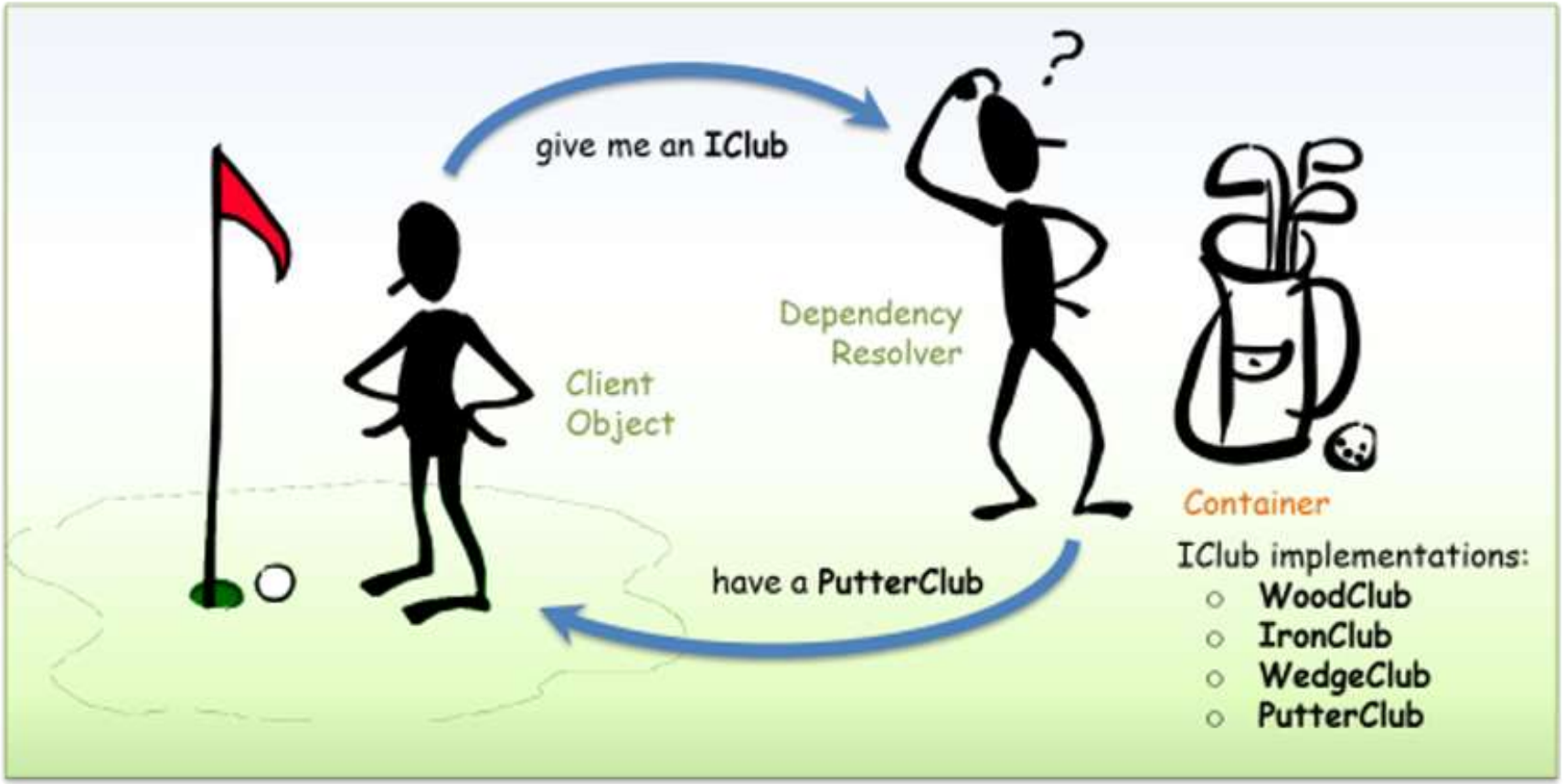
Decoupling dependencies

```
public class PaymentService
{
    private readonly string _databaseConnectionString = ConfigurationManager
                .ConnectionStrings["MyConnectionString"].ConnectionString;
    private readonly string _webServiceAddress = ConfigurationManager
            .AppSettings["WebServiceURL"];
    private readonly string _cachedFiles = ConfigurationManager
            .AppSettings["FileCacheLocation"];

    private FileCache _cache;
    private DataAccessComponent _dataAccessComponent;
    private WebServiceProxy _webServiceProxy;
    private LoggingComponent _loggingComponent;

    public PaymentService()
    {
        _cache = new FileCache(_cachedFiles);
        _loggingComponent = new LoggingComponent();
        _webServiceProxy = new WebServiceProxy(_webServiceAddress);
        _dataAccessComponent = new DataAccessComponent(_databaseConnectionString);
    }
```

Dependency Injection

```
public class PaymentService
{
    private ICache _cache;
    private IDataAccessComponent _dataAccessComponent;
    private IWebServiceProxy _webServiceProxy;
    private ILoggingComponent _loggingComponent;

    public PaymentService(ICache cache, IDataAccessComponent dataAccessComponent
                ,IWebServiceProxy webServiceProxy, ILoggingComponent loggingComponent)
    {
        _cache = cache;
        _loggingComponent = loggingComponent;
        _webServiceProxy = webServiceProxy;
        _dataAccessComponent = dataAccessComponent;
    }
```

Using IoC resolver to inject dependency



IoC container example (Using Ninject)

```
public class WebModule : NinjectModule
{
    public override void Load()
    {
        Bind<ICache>().To<HttpCacheProvider>();
        Bind<ILoggingComponent>().To<LoggingComponent>();
    }
}
```

Instantiating dependent with dependencies

```
public class DataAccessComponentProvider : Provider<IDataAccessComponent>
{
    protected override IDataAccessComponent CreateInstance(IContext context)
    {
        var databaseConnectionString = ConfigurationManager
                            .ConnectionStrings["MyConnectionString"]
                            .ConnectionString;
        return new DataAccessComponent(databaseConnectionString);
    }
}


public class WebServiceProxyComponentProvider : Provider<IWebServiceProxy>
{
    protected override IWebServiceProxy CreateInstance(IContext context)
    {
        var webServiceAddress = ConfigurationManager
                        .AppSettings["MyWebServiceAddress"];
        return new WebServiceProxy(webServiceAddress);
    }
}
```

Using providers

```
public class WebModule : NinjectModule
{
    public override void Load()
    {
        Bind<ICache>().To<HttpCacheProvider>();
        Bind<ILoggingComponent>().To<LoggingComponent>();
        Bind<IDataAccessComponent>().ToProvider(new DataAccessComponentProvider());
        Bind<IWebServiceProxy>().ToProvider(new WebServiceProxyComponentProvider());
    }
}
```

Finally

```
public class WebModule : NinjectModule
{
    public override void Load()
    {
        Bind<ICache>().To<HttpCacheProvider>();
        Bind<ILoggingComponent>().To<LoggingComponent>();
        Bind<IDataAccessComponent>().ToProvider(new DataAccessComponentProvider());
        Bind<IWebServiceProxy>().ToProvider(new WebServiceProxyComponentProvider());
        Bind<IPaymentService>().To<PaymentService>();
        //Or    Bind<PaymentService>().ToSelf();
    }
}

public class PaymentService : IPaymentService
{
    private ICache _cache;
    private IDataAccessComponent _dataAccessComponent;
    private IWebServiceProxy _webServiceProxy;
    private ILoggingComponent _loggingComponent;

    public PaymentService(ICache cache, IDataAccessComponent dataAccessComponent
                , IWebServiceProxy webServiceProxy, ILoggingComponent loggingComponent)
    {
        _cache = cache;
        _loggingComponent = loggingComponent;
        _webServiceProxy = webServiceProxy;
        _dataAccessComponent = dataAccessComponent;
    }
```

Resolving Dependency at run time

```
//repository class that enables you to request objects from the framework
var kernal = new StandardKernel(new WebModule(), new ServicesModule());


//Resolving (pulling from the IoC repository)
var paymentService = kernal.Get<IPaymentService>();
```

Mocking dependencies

Why Mocking?

Options:

- Custom rolled mock implementation of dependency
- Frameworks like Moq, RhinoMocks, Nsubstitute etc…

Mocking with Moq

```
//Account Service
public decimal GetAccountBalance(int vendorId) {

    var vendor = _vendorRepository.Get(vendorId);
    HandleNull(vendor);
    return vendor.TotalDue - vendor.TotalPaid;
}
```

```
[TestMethod]
public void GetAccountBalance_ReturnsBalanceSuccessfully() {
    //Mock
    Mock<IVendorRepository> mockVendorRepository = new Mock<IVendorRepository>();
    mockVendorRepository.Setup(vr => vr.Get(It.IsAny<int>()))
                        .Returns(new Customer {
                                        DiscountRate = 0,
                                        IsPreferredVendor = true,
                                        TotalDue = 100m,
                                        TotalPaid = 0m
                                    });
    var paymentService = new PaymentService(mockVendorRepository.Object);
    var balance = paymentService.GetAccountBalance(123);

    Assert.Equals(100, balance);
}
```

Integration test

- Any test that crosses the boundary of a Unit is Integration test.
- A full spike - end to end feature test.

Best practice / lesson learnt

- Naming test cases (Readable & Verbose)
- Exhaustive test cases – pass, fail, corner cases and conditions
- Test only the code you wrote.
- Must target all business logic / features.
- Limit mocking maintenance burden.
- Don’t cheat… ever (once or twice is enough to bring the suite down)

Recent criticism - TDD is dead. Long live Testing.

- Unit testing is very shallow and it does not has a cross layer reach.
- Too much focus on execution time of the regression test suite.
- TDD tends to Mocks heavy architecture, everything stubbed out creates refactoring difficult.
- Red-green-refactor loop – less optional and more of a mandate. Some developers do not really enjoy that development.
- TDD nudges towards over testing cost, testing code is still code that needs maintenance.
- Reduced QA people – the person who created cannot be the person who will test to break. But in reality, it eliminated the testers who only run pre written test scripts.
- Cost Vs life of the system (ignore rare cases)

6 Google hangout sessions on YouTube - http://www.youtube.com/watch?v=z9quxZsLcfo

Benefits

- Business requirement are living running test cases not document that can be missed.
- Have I broken anything while making a change? – Self testing regression suite.
- Is this code base healthy? The code base is handed over to different

Open forum

Experiences
Opinions
Questions

Slides and Code on the blog.

Audience-Driven Development.

Thank you.