Keeping Software Soft

A Practical Guide for Developers, Testers, and Managers

1st Edition

Jason Roberts

Keeping Software Soft

A Practical Guide for Developers, Testers, and Managers

Jason Roberts

This book is for sale at http://leanpub.com/KeepingSoftwareSoft

This version was published on 2015-06-10

ISBN 978-0-9924164-0-9

For my late mother who fed my imagination as a child and for my father for encouraging my love of science.

Thanks to all the early supporters of this book, including:

- *Mario Lopez*
- *Yolanda Maguire*
- *Gentry Riggen*
- *James Tweedie*

Contents

About The Author

Jason Roberts is a Journeyman Software Developer with over 12 years experience. He is a Microsoft C# MVP, a Pluralsight course author[1] and holds an honours degree in computing. He is a writer, open source contributor and has worked on numerous apps for both Windows Phone and Windows Store.

You can find him on Twitter as @robertsjason[2] and at his blog DontCodeTired.com[3].

Other Leanpub Books by Jason Roberts

Clean C

This book will help you write cleaner, more maintainable, more readable, and more pleasurable C# code.

Clean C#[4]

C# Tips

Write better C#.

This book will help you become a better C# programmer. It contains a whole host of useful tips on using C# and .Net.

C# Tips[5]

[1]http://bit.ly/psjasonroberts
[2]https://twitter.com/robertsjason
[3]http://dontcodetired.com
[4]http://cleancsharp.com/
[5]http://bit.ly/sharpbook

Pluralsight Courses

Browse Pluralsight courses by Jason Roberts[6]

[6]http://bit.ly/psjasonroberts

Introduction

Why Keep Software Soft?

Software should be easy to change.

Why should we care about keeping software soft? We don't have to care, but we accept that software that is not soft may be more aptly described as "rigidware".

Some argue that over time all software ends up as a big ball of goo: hard to understand, hard to learn, and hard to change.

We can respond to this argument in two ways: "why bother to even *try* and keep it soft then?"; or "lets try to slow the degradation as much as we can".

This still doesn't answer the question of *why* we should try and keep software soft; here are a few reasons:

- May result in lower total cost of ownership (TCO) for a longer-lived system
- May allow the business to be more agile (responding more quickly to changing marketplace conditions)
- May result in happier development teams (which in turn, may increase productivity and reduce staff turnover)

If these things aren't important to a business, if it has an unlimited supply of funds, time, and people; and the market in which it operates is slow moving and non-innovative, then this "rigidware" may well be acceptable.

Why Read This Book?

If you want to make developing software a happier, more enjoyable, and more productive experience this book is for you.

Keeping Software Soft provides practical guidance on how to keep software as soft as we can. You can read from beginning to end or just jump in at any point and use as a reference guide.

Part 1: The Code

It is amazing how much time is spent in management meetings talking about how to improve delivery speed and reduce costs. It sometimes seems that agile software development is seen as a silver-bullet that can fix all the ingrained problems an organisation has.

The strangest thing sometimes is how little energy is spent talking about improving the codebase itself.

Regardless of which software development life cycle process is being used, ultimately it is the code that must be transformed from its current state to a new state to support the goals of the organisation or users.

Part 1 of this book is focussed on the code itself, and how a codebase can be kept more pliable and easier to change.

The SOLID Principles of OO Design

The SOLID acronym represents some important principles of object-oriented programming and design.

It stands for:

- **S** - Single Responsibility Principle
- **O** - Open Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

This chapter examines each of these principles in term and relates them back to how they help to keep software soft.

The Single Responsibility Principle (SRP)

Any given class should do one well-defined thing, and **only** that one well-defined thing.

To put in another way, if a class has more than one responsibility, it will have to change whenever **any** one of its many responsibilities change.

If there is more than one reason to change a class, it is likely that the Single Responsibility Principle has been violated.

As an example, consider an `OrderManager` class that is responsible for: calculating the total order amount, debiting a credit card, and updating the order status in the database. Clearly this is a violation of SRP. Instead, a cleaner implementation could be to split out each responsibility into three separate classes: `OrderTotalsCalculator`, `CreditCardCharger`, and `OrderRepository`. Now each class only needs to change if its responsibility changes.

Why SRP?

SRP contributes to keeping software soft because:

- A class that conforms to SRP only needs to be changed when the thing it is responsible for changes

- Each class now has fewer lines of more highly-related code, so it should be easier for a developer to understand in its entirety
- More individual source code files (one for each responsible class) should result in fewer merge conflicts
- Testability is likely to be better and the related test code more highly focussed and cohesive

Without SRP there is greater fear (and potentially higher regression test costs) when changing any given part of the system.

> *Customer*: “We want to change the way order totals are calculated.”
>
> *Development Team*: “No problem, give us the new business rules, we’ll write some tests then go and change the `OrderCalculator`, we don’t have to worry about breaking the database because that’s in a separate place”.

The Open Closed Principle (OCP)

The Open Closed Principle states that a class should be open for extension, but closed for modification.

This means that it should be possible to add or change some behaviour without modifying the existing class. For example, the `OrderCalculator` class should be able to be extended to handle VIP customers who may get a discount or other preferential treatment. In this example the `CalculateOrderLine` method could be overridden in a derived `VipOrderCalculator` class. (Note: it could be argued here that to satisfy the SRP the `OrderLine` should be in it’s own class).

Why OCP?

The less that code that is in use is modified, the less chance there is of introducing defects.

If existing code is not being changed, but rather **extended** to account for new requirements, the fear of changing it is lower and the number of costly defects may be reduced.

> *Customer*: “We need to give VIP customers a `5%` discount”.
>
> *Development Team*: “No problem, we’ll add on to what we already do for existing customers, don’t worry about normal customers, we won’t be affecting those.”

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle means that calling code should be able to use an instance of a base class or an instance of a derived class without knowing it, or having to do anything special.

A square is not a rectangle. This is the canonical example used to explain the Liskov Substitution Principle. A square only has one value for the length of a side, whereas a rectangle has two. To calculate the area of a square, we only need to multiply the length of a side by itself. A rectangle requires one side length to be multiplied by the other. If we derived `Square` from `Rectangle` (or vice versa) how would a calling client know if it needed to provide a width and height (in the case of rectangle) or just a single side length in the case of a square.

In the case above, it is possible to introduce an abstraction in the form of an interface `IShape`, that declares methods such as `CalculateArea`.

LSP can also help to identify where the OO model may be incorrect. For example, if code starts to look like `if (shape is Square) ... else if (shape is Rectangle) ...`, it can be a “code smell” and the design evaluated against LSP.

Why LSP?

The LSP helps to reduce complexity in calling code so that it can work generically on any class or subclass without additional conditional code.

It helps to keep the abstractions and concepts used in the code base clean, consistent and easily understandable.

LSP also helps to keep the OO principle of polymorphism well implemented.

> *Customer*: “We need to be able to draw filled polygons…”
>
> *Development Team*: “No problem, most of the code we have will just work with the new polygons…”

The Interface Segregation Principle (ISP)

The Interface Segregation Principle is similar to the Single Responsibility Principle but applied to interfaces. It leads to the development of a greater number of more highly cohesive interfaces, rather than fewer bigger and bloated ones.

Put another way, the class implementing the interface should not have to implement lots of things that it does not care about; it should only have to implement the things it needs.

A greater number of smaller, more highly focused interfaces is like a menu from which classes can choose (and implement) only the things they need.

Why ISP?

ISP helps to reduce complexity and increase flexibility. A class can choose to implement `IPrintable` and `IRotatable` but does not have to implement anything else (such as `ISizeable`). If there was only one interface (such as `IShapeOperations`) that contained printing, rotating and sizing, a client class could not just implement printing on its own.

> *Customer*: “We need to add printing support for circles, but we don’t want them to be resizable.”
>
> *Development Team*: “No problem, we already have print support for squares [e.g. `IPrintable`] we’ll need to add that to circles so they can be printed like some of the other shapes.”

Dependency Inversion Principle (DIP)

The DIP helps to reduce tight coupling. The principle has two parts. The first that higher level code should not depend on lower level code, instead both high and low level code should depend on some abstraction. The second part is the abstractions should not depend on the details, rather the other way around, that the details should depend on the abstractions.

The higher level code here represents the “what” or “workflow” with the lower level code performing specific actions as part of this workflow.

Image a Morse code application, its job at the higher level (“workflow) is to read a letter from the keyboard and convert it to Morse code (a series of “dots” and “dashes”), then output it to the console.

Take the following simplistic implementation:

```
namespace MorseCoder
{
        class Program
        {
                static void Main(string[] args)
                {
                        MorseCodeCalculator morseCodeCalculator = new MorseCodeCalculator();
                        MorseCodeWriter morseOutput = new MorseCodeWriter();

                        while (true)
                        {
                                var keyboardLetter = Console.ReadKey().KeyChar;

                                string morse = morseCodeCalculator.FromLetter(keyboardLetter);

                                morseOutput.Output(morse);
                        }
                }
        }

        class MorseCodeCalculator
        {
                public string FromLetter(char letter)
                {
                        // implementation omitted for brevity
                        return "._.";
                }
        }

        class MorseCodeWriter
        {
                public void Output(string morse)
                {
                        Console.WriteLine(" = {0}", morse);
                }
        }
}
```

In the preceding code, the high level workflow can be generalised as:

- Get a letter from somewhere
- Convert this letter to Morse
- Output the Morse version in some way

Let's take the final part as an example. Here the high level code (`class Program`) is dependent on the lower level code (`class MorseCodeWriter`). To conform to DIP we need both `Program` and `MorseCodeWriter` to depend on an abstraction instead. The following code shows this abstraction in the form of the `IMorseCodeWriter` interface. Both `Program` (high level code) and `MorseCodeWriter` (low level code) now depend on this interface abstraction.

```
namespace MorseCoder
{
        class Program
        {
                static void Main(string[] args)
                {
                        MorseCodeCalculator morseCodeCalculator = new MorseCodeCalculator();
                        IMorseCodeWriter morseOutput = new MorseCodeWriter();

                        while (true)
                        {
                                var keyboardLetter = Console.ReadKey().KeyChar;

                                string morse = morseCodeCalculator.FromLetter(keyboardLetter);

                                morseOutput.Output(morse);
                        }
                }
        }

        class MorseCodeCalculator
        {
                public string FromLetter(char letter)
                {
                        // implementation omitted for brevity
                        return "._.";
                }
        }

        interface IMorseCodeWriter
        {
                void Output(string morse);
        }

        class MorseCodeWriter : IMorseCodeWriter
        {
                public void Output(string morse)
                {
                        Console.WriteLine(" = {0}", morse);
                }
        }
}
```

The introduction of this abstraction now means that a different implementation of `IMorseCodeWriter` can now be created, for example outputting a series of short and long beeps, rather than write to the console.

Discussions relating to DIP are often co-mingled with Inversion Of Control (IOC), Dependency Injection (DI), and DI Containers. These topics are covered elsewhere in this book. Suffice it to say that DIP is the principle that underlies these techniques.

Why DIP?

Without DIP, the high level "workflow" is harder to separate from the implementation (in the lower level code). Classes are more tightly coupled, meaning that changes can ripple throughout the system, and it may make unit testing harder because fake versions of dependencies cannot easily be supplied. This means it is harder to unit test the class in isolation because the object is creating its own (non abstract) dependencies internally.

DIP in conjunction with dependency injection also makes it possible to provide different concrete implementations of dependencies at runtime.

> *Customer*: "We need to send emails via a secure internal channel for some customers".
>
> *Development Team*: "No problems, we'll create another implementation of `IEmailGateway` that uses a secure email channel and use that instead for those customers".

Other Important Principles

In addition to SOLID, there are a number of other principles that help keep software soft.

Keep It Simple Stupid (KISS)

This principle suggests that code that is as simple as possible is better than code that is more complex.

There is a balance between using all the language features and being self-disciplined and writing what could be described as a less “elegant”, but more easily maintainable code. For example, LINQ (Language Integrated Query) to objects can greatly simplify code, but a three page LINQ-to-SQL statement works against the goal of simplicity.

Another way to think about KISS is to ask: is this code likely to be easily repairable by an average programmer working under pressure during a production defect/outage?

The full power of language and tooling should absolutely be used where appropriate, but when choosing between two possible implementations (all other things being equal) the simplest solution should usually be chosen.

There can be a natural tension in teams where the programmers are of sufficiently different skill levels or outlooks as to what constitutes simplicity. If it is purely due to less experienced programmers not wanting to use language or tooling they do not understand, then pair programming can be used as a technique to help mentor them, so that the full power of the language can be used to solve the problem.

Don’t Repeat Yourself (DRY)

DRY is sometimes referred to as the “Single Source of Truth” or “Once and Only Once”.

At its most basic level, DRY can be described as not doing copy-and-paste coding. There are plenty of legacy codebases existing today where lines of code, whole methods, classes or even whole projects are duplicated by copy and pasting.

There is a more subtle meaning to the DRY principle: “every piece of knowledge must have a single, unambiguous, authoritative representation within a system” [Hunt, A & Thomas, D. (1999) *The Pragmatic Programmer: From Journeyman to Master*]

This means that any given *concept* or *abstraction* that the code embodies should be defined only once in one place and should be clear to the reader/maintainer.

DRY is usually only applied to hand-written code. When working in a tool or at a higher abstraction level that uses code generation to write the actual source code, the generated code may not be subject to DRY conformity.

DRY can also apply to things such as configuration files; for example a 300 line config file that has 5 copies, one for each deployment environment, but that only contains a few lines difference would be considered a violation of the DRY principle. Instead there should be only one copy that gets transformed to change the relevant values for the different environments.

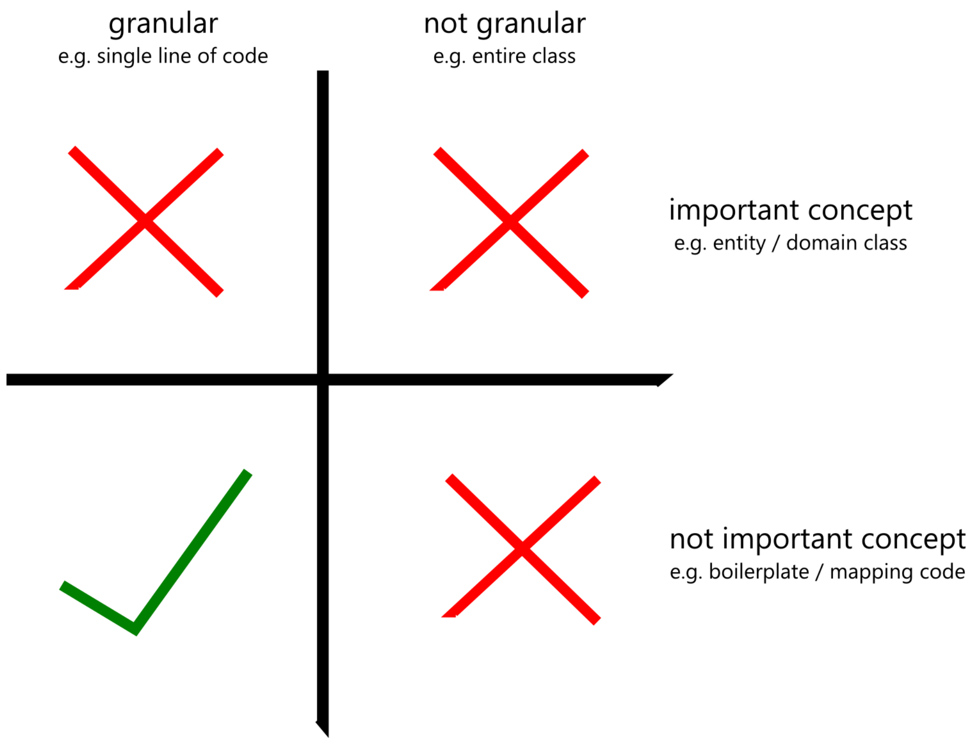
Potentially Acceptable Code Duplication

There is a concept (sometime referred to as the Rule of Three) that states that having two copies of the same thing is acceptable, but on needing a third copy all three should be refactored down to one copy. This "rule" very much depends on what level of granularity of code it is being applied to.

Having more than one identical copy of a class is not usually acceptable. Having more than one exact copy of a method or function may be more acceptable. Having more than one exact copy of a single line of code may be more acceptable still.

In addition to granularity, if the duplicated code represents an important *concept/abstraction* in the system then any duplication may be unacceptable.

The diagram below shows how these two factors may influence the decision to allow duplication.



Potentially Acceptable Code Duplication

You Aren’t Gonna Need It (YAGNI)

This simple principle advocates for the resistance of the temptation to second-guess what code *might* be needed at some future point and adding it now, rather than adding it only when it is *actually* needed.

If the code is not needed to complete the current feature or bug fix then it should not be added yet.

Violating the YAGNI principle can result in:

- Bloated code: unused/unrequired code exists now, and may never be removed/refactored
- Increased Cost: the unrequired code may still need testing and maintaining
- Slipped dates: the unrequired code takes time away from other features that may impact overall delivery dates
- Waste: by the time the unrequired code is *actually required*, it may no longer be valid if business requirements have changed in the meantime

When practicing strict YAGNI, a strict approach to refactoring is also usually needed; as Jeff Atwood[7] states: “Build what you need as you need it, aggressively refactoring as you go along; don’t spend a lot of time planning for grandiose, unknown future scenarios”.

Cohesion

Cohesion describes how closely related (or not) pieces of code are to each other.

Cohesion is usually referred to as being either **high** or **low**.

At the method level, high cohesion would mean that the lines of code within the method come together to perform one small task on a small amount of closely related data or objects.

At the class level, high cohesion would mean the members (methods, properties, events, etc.) are all closely related to each other and to the overall purpose of the class.

The concept of cohesion also applies at the namespace level and at the assembly level.

Although usually referred to as high or low, there are a series of relative qualitative measures between the two extremes. The worst kind of cohesion is that of “Coincidental Cohesion”, this is code that has been grouped together in a completely arbitrary way with no thought having been put into it. The generally accepted best kind of cohesion is “Functional Cohesion”: this is where pieces of code are highly related by the task that they perform.

The complete list of the types of cohesion (from best to worst) is:

- Functional Cohesion: related by functionality/task

[7] http://www.codinghorror.com/blog/2004/10/kiss-and-yagni.html

- Sequential Cohesion: related by input/output data flows
- Communicational Cohesion: related by operating on the same set(s) of data
- Procedural Cohesion: related by always following the same sequence of execution
- Temporal Cohesion: related by the points in time when they are used
- Logical Cohesion: related by some logical category but different by actual nature
- Coincidental Cohesion: not related

Higher cohesion helps keep software soft because individual pieces of code are:

- Easier to understand
- Need to change less often
- More easily usable, reusable, and composable
- Easier to unit test
- Easier to make well-named

Coupling

Coupling refers to the level of knowledge that one piece of code has about another piece of code that it is interacting with. Coupling is usually referred to as being either **loose/weak** or **tight/strong**.

Tight coupling means that one piece of code relies on the internal implementation details of code it collaborates with; it uses knowledge of *how* the other code does something rather than just relying on it to *do* something.

Loose coupling means that one piece of code does not rely upon, or even have knowledge of, the implementation details of the code it interacts with.

Loosely coupled code helps to keep software soft because changes in private, internal implementation details do not ripple throughout the rest of the system.

Law of Demeter (LoD)

The Law of Demeter (sometimes known as Principle of Least Knowledge) is a corollary to the concept of loose coupling. Colloquially, LoD is sometimes expressed as: “talk only to your friends, not to strangers.” This suggests that it is ok to make reference to immediate collaborators (“friends”) from a piece of code, but not the collaborators (strangers) of those collaborators.

The canonical example of a violation of LoD is the “Paper Boy, Customer, and Wallet” example. The code below shows a violation of LoD.

```
namespace KeepingSoftwareSoftSamples.OtherPrinciples.LoDViolated
{
    public class Wallet
    {
        public decimal Cash { get; set; }
    }

    public class Customer
    {
        public Wallet Wallet { get; set; }
    }

    public class PaperBoy
    {
        public Customer Customer { get; set; }
        public decimal CollectedCash { get; set; }

         public void CollectPayment()
         {
             // Arbitrary amount for demo purposes
             const decimal amountOwed = 2.45m;

             // Violation of LoD occurs here:
             //  - Customer is a "friend"
             //  - Wallet is a "stranger"
             //  - PaperBoy should not be "reaching into"
             //    a Customer's Wallet
             Customer.Wallet.Cash -= amountOwed;

             CollectedCash += amountOwed;
         }
    }
}
```

A fixed version could look like the following, note the `GetPayment` method is essentially a wrapper/”delegate” that removes the need for `PaperBoy` to know about `Wallet`:

```
namespace KeepingSoftwareSoftSamples.OtherPrinciples.LoDFixed
{
    public class Wallet
    {
        public decimal Cash { get; set; }
    }

    public class Customer
    {
        private Wallet Wallet { get; set; }

        public decimal GetPayment(decimal amountOwed)
        {
            // "is there enough in wallet" checking logic omitted
            Wallet.Cash -= amountOwed;

            return amountOwed;
        }
    }

    public class PaperBoy
    {
        public Customer Customer { get; set; }
        public decimal CollectedCash { get; set; }

         public void CollectPayment()
         {
             // Arbitrary amount for demo purposes
             const decimal amountOwed = 2.45m;

             // Fixed violation of LoD here
            CollectedCash += Customer.GetPayment(amountOwed);
         }
    }
}
```

Now the `PaperBoy` does not need to know where the `Customer` is keeping their money.

This now enables the `Customer` to change how/where it stores its cash and the `PaperBoy` is insulated from the change:

```
public class Customer
{
    private Wallet Wallet { get; set; }
    private LooseChangeJar LooseChange { get; set; }

    public decimal GetPayment(decimal amountOwed)
    {
        // Additional "business" logic can be added here
        // without needing PaperBoy to change as well

        var isEnoughLooseChange = LooseChange.Cash >= amountOwed;

        if (isEnoughLooseChange)
        {
            LooseChange.Cash -= amountOwed;
        }
        else
        {
            // "is there enough in wallet" checking logic omitted
            Wallet.Cash -= amountOwed;
        }

        return amountOwed;
    }
}
```

There are some instances where it may be perfectly valid to "violate" LoD. As Martin Fowler has stated[8]: "*I'd prefer it to be called the Occasionally Useful Suggestion of Demeter*". For example when dealing with views (ASP.NET MVC view, databound XAML view, etc.) it can be acceptable to reach into strangers' *data* in order to display it.

When deciding whether to apply LoD, it can be worthwhile pausing to consider is it **data** or **behaviour** that is being wrapped/"delegated". If it is behaviour then it can often be a better candidate for applying LoD than data. Blindly applying LoD can increase the maintenance overhead of software due to the added wrapper/"delegate" code.

[8]https://twitter.com/martinfowler/status/1649793241

Dependency Inversion Revisited

The Dependency Inversion Principle (DIP) was introduced in the chapter on the SOLID principles.

This chapter expands on the concept of DIP and introduces the related concepts of Inversion of Control (IoC) and Dependency Injection (DI).

Inversion of Control is a concept that states that objects should not create their own dependencies, but rather the *control* of dependency creation should be *inverted* and given to something else.

Dependency Injection is specific implementation that falls under the more generic umbrella term of IoC.

DI is one possible way to achieve IoC, other methods include the Service Locator design pattern, see the Useful Design Patterns chapter for more information.

When these terms are first encountered there can be a feeling of “acronym overload” or “but how do I do it?”. The remainder of this chapter discusses a practical implementation of IoC through the use of a Dependency Injection Container (DIC).

Dependency Injection Containers

A Dependency Injection Container (DIC) typically provides the following features:

- Creates objects that other objects need (i.e. their dependencies)
- Controls the lifetime of the objects that it creates
- Creates objects for dependencies all the way down an object hierarchy

If the Dependency Inversion Principle has been followed (high and low level code now depends on abstractions), a graph of objects, each with their own dependencies, can have those dependencies supplied to them (for example via a constructor parameter) by the Dependency Injection Container.

A container typically allows the lifetime of the objects that it creates to be controlled, for example supplying the same object instance (singleton) for all dependencies of a given kind.

A container supplies the dependencies not only for the first layer of dependencies, but throughout the entire object-dependency graph.

An Example of Dependency Injection with Ninject

Ninject[9] is an open source dependency injector for .NET.

A Ninject `StandardKernel` represents the container, this is the thing that creates dependencies for objects.

The example below creates a graph of dependencies, starting at the top with an instance of `AdependsOnB`.

```
// Define some interfaces that we can later map to concrete
// types in the DI container.
public interface IB {};
public interface IC {};
public interface ID {};
public interface IE {};

public class AdependsOnB
{
    public IB B { get; set; }

    public AdependsOnB(IB b)
    {
        B = b;
    }
}

public class BdependsOnC : IB
{
    public IC C { get; set; }

    public BdependsOnC(IC c)
    {
        C = c;
    }
}

public class CdependsOnDandE : IC
{
    public ID D { get; set; }
    public IE E { get; set; }

```

[9] http://www.ninject.org/

```
    public CdependsOnDandE(ID d, IE e)
    {
        D = d;
        E = e;
    }
}

public class D : ID {}

public class E : IE {}



public class ProgramWithNoDi
{
    public void CreateObjectGraphByHand()
    {
        // Manually 'wire up' the sub dependencies
        var a = new AdependsOnB(
                        new BdependsOnC(
                            new CdependsOnDandE(new D(), new E())));
    }
}


public class ProgramWithDi
{
    public void CreateObjectGraphWithDiContainer()
    {
        // Create the 'DI container'
        var kernel = new StandardKernel();

        // Tell the container which concrete types to create
        kernel.Bind<IB>().To<BdependsOnC>();
        kernel.Bind<IC>().To<CdependsOnDandE>();
        kernel.Bind<ID>().To<D>();
        kernel.Bind<IE>().To<E>();

        // Create an instance of ADependsOnB, letting Ninject
        // 'wire up' all the sub dependencies
        var a = kernel.Get<AdependsOnB>();
    }
```

```
}
```

In this trivial example, it could be argued that `ProgramWithDi` has more lines of code and is more complex. For large object graphs with many sub dependencies, wiring up dependencies by hand quickly becomes harder to maintain. Also when dependencies change, this hand-wiring also has to be changed - creating software that is less soft.

Auto-Wiring Dependencies in Ninject

In the `ProgramWithDi`, each interface is mapped to a concrete type. For example, `kernel.Bind<ID>().To<D>();` instructs Ninject to create a new `D` object whenever an `ID` dependency needs to be resolved.

For a complex system with many types, manually specifying each dependency mapping becomes error prone, creates a maintenance overhead, and prevents us from easily adding or removing dependency mappings.

An alternative to manually specifying dependency mappings is to instruct the DIC to automatically discover and map each type to a concrete dependency based on the idea of "convention over configuration". In this way, `IFoo` automatically maps to `Foo`.

The example below shows how Ninject can be instructed to search for and auto-wire up dependencies in the current assembly:

```
public class ProgramWithAutoWiredDi
{
    public void CreateObjectGraphWithDiContainer()
    {
        // Create the 'DI container'
        var kernel = new StandardKernel();

        // Tell the container to search the current assembly...
        kernel.Bind(x => x
                         .FromThisAssembly()
                         .SelectAllClasses()
                         .BindAllInterfaces());

        // Create an instance of ADependsOnB
        var a = kernel.Get<AdependsOnB>();
    }
}
```

There are other ways to tell Ninject to discover and auto-wire dependencies, for further information see the Ninject website.

Other Containers

There are other containers available such as Castle Windsor, StructureMap and Unity. Each has their own syntax and way of doing things and each has different performance characteristics that may need to be considered before choosing one. For those using a DIC for the first time, Castle Windsor or Ninject would be good starting points.

Why Use a Dependency Injection Container?

Using an auto-wired DIC means that the architecture (dependencies) of the system are not hard-coded; this creates a more fluid system. Dependencies can be added and removed (for example constructor parameters) and classes can be refactored and combined without having to spend time coding how those dependencies are provided.

Some Useful Design Patterns

A design pattern is a solution to a common problem based on a given template. Patterns are generic reusable blueprints that can be applied when a certain design outcome is required. A design pattern is implemented by writing code.

This chapter introduces some common and useful design patterns that can help keep software soft. For a more complete list of patterns see Appendix D.

Gateway

The Gateway pattern "*encapsulates access to an external system or resource*" Fowler, M[10]. A gateway class provides a simplified way of interacting with something external that perhaps has a peculiar or difficult API.

For example, suppose an application needs to use some third-party external system to validate an address. The external system provides an API: `int val(string adLn1, string adLn2, string cntry)` that returns a 1 for valid address and a 0 for an invalid address.

A gateway class can be created to facilitate interaction with this external system:

```
public interface IAddressValidationGateway
{
    bool Validate(string addressLine1,
                  string addressLine2,
                  string country);
}

public class AddressValidationGateway : IAddressValidationGateway
{
    public bool Validate(string addressLine1,
                         string addressLine2,
                         string country)
    {
        const string licenceKey = "xyz";

        var v = new AddVal(licenceKey);

```

[10] http://martinfowler.com/eaaCatalog/gateway.html

```
        var result = v.val(addressLine1, addressLine2, country);

        const int validAddress = 1;

        return result == validAddress;
    }
}
```

The application being developed can now use the gateway:

```
IAddressValidationGateway a = new AddressValidationGateway();

var isValidAddress = a.Validate("Unit 2", "First Street", "Australia");

if (isValidAddress)
{
    // etc.
}
```

The Gateway pattern helps to keep software soft by abstracting and insulating the system being developed from its external dependencies. By also defining an interface for the gateway, it makes it possible to provide fake versions for testing purposes or swap out one third-party address validation service for another without the change rippling throughout the system. The gateway also helps to deal with anything unique to the external service, in the example above this can be seen in the provision of the `licenceKey` to the third-party service.

Decorator

The Decorator pattern allows the addition of behaviour to existing classes without changing the original class. The pattern also allows the "wrapping" of multiple decorators inside each other to compose behaviours.

The decorator pattern typically starts with an abstraction of some behaviour. In the example below this takes the form of an interface called `Tweeter` that allows a message to be sent:

```
public interface ITweeter
{
    void SendTweet(string message);
}
```

This interface is implemented in a class called `Tweeter` that can be considered the basic class that will be decorated later:

```
public class Tweeter : ITweeter
{
    public void SendTweet(string message)
    {
        Console.WriteLine("TWEETING: '{0}'", message);
    }
}
```

If `SendTweet` is called with a message of “Buy Keeping Software Soft” the following is output:

> TWEETING: ‘Buy Keeping Software Soft’

The `Tweeter` can be decorated. A decorator implements the same interface as the class it is decorating. It also contains a reference to the decorated (“wrapped”) instance.

As an example, suppose that a new requirement exists to be able to add the hashtag **#KeepingSoftwareSoft** to some messages. One way to do this would be to subclass Tweeter or implement `ITweeter` a second time.

Another approach is to create a `KeepingSoftwareSoftDecorator` that automatically appends the hashtag to all messages:

```
public class KeepingSoftwareSoftDecorator : ITweeter
{
    private readonly ITweeter _decoratedTweeter;

    public KeepingSoftwareSoftDecorator(ITweeter tweeter)
    {
        _decoratedTweeter = tweeter;
    }

    public void SendTweet(string message)
    {
        _decoratedTweeter.SendTweet(message + " #KeepingSoftwareSoft");
    }
}
```

This decorator delegates to the “wrapped” instance of `ITweeter` (as supplied via the constructor) but adds the extra hashtag string.

To use this decorated version:

```
ITweeter t = new Tweeter();

ITweeter decorated = new KeepingSoftwareSoftDecorator(t);

decorated.SendTweet("Buy Keeping Software Soft");
```

This produces the following output (note the hashtag):

> TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'

The key thing here is that extra runtime behaviour has been added **without** having to modify the original `Tweeter` class.

Suppose another new requirement surfaces that requires that certain messages need to be repeated (spammed) ten times. At this point it would be possible to create another implementation of `ITweeter` or subclass `Tweeter`, but instead another decorator can be defined:

```
public class SpamDecorator : ITweeter
{
    private readonly ITweeter _decoratedTweeter;

    public SpamDecorator(ITweeter tweeter)
    {
        _decoratedTweeter = tweeter;
    }

    public void SendTweet(string message)
    {
        const int numberOfSpamMessages = 10;

        for (var i = 0; i < numberOfSpamMessages; i++)
        {
            _decoratedTweeter.SendTweet(message);
        }
    }
}
```

This decorator simply delegates to the "wrapped" `ITweeter` and calls it 10 times.

To use this decorated version:

```
ITweeter t = new Tweeter();

ITweeter decorated = new SpamDecorator(t);

decorated.SendTweet("Buy Keeping Software Soft");
```

This produces the following output:

> TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'TWEETING: 'Buy Keeping Software Soft'

At this point it is possible to send messages with a hashtag or send spam. To the calling code, everything is still an `ITweeter`, it does not care if it happens to be decorated or not.

Decorators can be composed with other decorators. For example, if some messages now need the hashtag to be added **and** sent 10 times, the existing two decorators can be combined:

```
ITweeter t = new SpamDecorator(
    new KeepingSoftwareSoftDecorator(
        new Tweeter()));

t.SendTweet("Buy Keeping Software Soft");
```

This produces the following output (note the hashtag and the 10 repeats):

TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'TWEETING: 'Buy Keeping Software Soft #KeepingSoftwareSoft'

Decorators help to keep software soft because existing behaviour can be extended and composed without having to change existing implementation code and thus risk introducing regression defects.

As with all design patterns it should be used wisely: "*Decorator works best when you have just a single I/O channel…[it] is a great way to allow to dynamically compose the way we apply processing to it.*" [Rahien, A. Design patterns in the test of time: Decorator[11]].

[11] http://ayende.com/blog/159553/design-patterns-in-the-test-of-time-decorator

Strategy

The Strategy pattern encapsulates a set of related algorithms into individual classes and makes them interchangeable with each other. This means that client code can choose which algorithm to use without needing to know the internal details of how each one works.

As an example, suppose there is a requirement to simulate the potential gain/loss for NASDAQ stock. There needs to be a quick and less-accurate simulation for getting a rough idea and a long running simulation when more accuracy is required.

An initial implementation could look something like:

```
public enum SimulationMethod
{
    Quick,
    Complex
}

public class StockSimulatorUsingSwitch
{
    public decimal SimulateNetGainOrLoss(string nasdaqCode,
                                         SimulationMethod method)
    {
        switch (method)
        {
            case SimulationMethod.Quick:
                return 0;
            case SimulationMethod.Complex:
                return -2;
            default:
                throw new ArgumentOutOfRangeException("method");
        }
    }
}
```

Client code would call this:

```
var s = new StockSimulatorUsingSwitch();
var simulatedValue = s.SimulateNetGainOrLoss("XXXX",
                                             SimulationMethod.Quick);
```

Now suppose another requirement surfaces to add another "super complex" simulation method. In the implementation above existing code will need to be changed: a new `enum` value and a new `case` line in the switch statement.

An alternative is to use the Strategy pattern.

Firstly an abstraction for the algorithm (the simulation) is created in the form of an interface:

```
public interface IStockSimulation
{
    decimal SimulateNetGainOrLoss(string nasdaqCode);
}
```

Next, two concrete classes are created to represent the two different simulation algorithms, both implementing the `IStockSimulation` interface:

```
public class QuickSimulation : IStockSimulation
{
    public decimal SimulateNetGainOrLoss(string nasdaqCode)
    {
        return 0;
    }
}

public class ComplexSimulation : IStockSimulation
{
    public decimal SimulateNetGainOrLoss(string nasdaqCode)
    {
        return -24;
    }
}
```

Now the simulator class can be refactored:

```
public class StockSimulatorUsingStrategy
{
    public decimal SimulateNetGainOrLoss(string nasdaqCode,
                                         IStockSimulation simulator)
    {
        return simulator.SimulateNetGainOrLoss(nasdaqCode);
    }
}
```

The simulator now takes both the `nasdaqCode` and the type of simulation to be applied. Client code now specifies the type of simulator to use, rather than as an enum:

```
var s = new StockSimulatorUsingStrategy();
var simulatedValue = s.SimulateNetGainOrLoss("XXXX",
                                            new QuickSimulation());
```

Now when the new "super complex" simulation method requirement surfaces, a new implementation of `IStockSimulation` can be created and passed into the `SimulateNetGainOrLoss` method as required. The `StockSimulatorUsingStrategy` class does not need to be changed and there is no enum or switch statement to modify and re-test.

The Strategy pattern helps to keep software soft by helping to adhere to the Open Closed Principle. It means that software is more easily extensible without having to modify existing code and risk regression defects.

The Strategy pattern can be combined with the Factory pattern (below) to further abstract away which algorithm gets used in what circumstances. For example, rather than client code creating a specific simulator, the factory could return the correct one based on the time of day: during trading hours the quick simulator, and outside of trading hours the complex one.

Factory

The Factory pattern allows client code to delegate creation of a particular kind of object to another class ("factory"). The type of the returned instance can depend on some data supplied by the client or can be based on the current context or state of the application.

As an example, suppose there is an `IGreeter` that is responsible for outputting a message dependant on either a time based value or a gender based value.

First of all the interface is defined:

```
public interface IGreeter
{
    void Greet();
}
```

Then the factory:

```
public static class GreeterFactory
{
    public static IGreeter FromTimeOfDay(DateTime time)
    {
        var isEvening = time.Hour > 18 || time.Hour < 5;

        if (isEvening)
            return new EveningGreeter();

        return new DaytimeGreeter();
    }

    public static IGreeter FromGender(bool isMale)
    {
        if (isMale)
            return new MaleGreeter();

        return new FemaleGreeter();
    }
}
```

Now client code can ask for an `IGreeter` to be created:

```
GreeterFactory.FromGender(true).Greet();
GreeterFactory.FromGender(false).Greet();

var midnight = new DateTime(2000, 1, 1, 0, 0, 0);
var midday = new DateTime(2000, 1, 1, 12, 0, 0);

GreeterFactory.FromTimeOfDay(midnight).Greet();
GreeterFactory.FromTimeOfDay(midday).Greet();
```

The client code does not care how the factory goes about creating an object, it just knows that it should get back an `IGreeter` of the correct kind.

The factory method does not have to have any parameters passed to it, instead it may just examine the existing state of the application to make the decision.

Chain of Responsibility

The Chain of Responsibility pattern routes a “request” through a series of “handlers”. Each “handler” determines if it can act on the “request”. If it can it does so and the chain ends, otherwise the “request” passes to the next “handler” in the chain. In this way, each handler determines if it responsible for handling the message.

It is also possible to create a variation on this pattern where the “request” passes through all “handlers” allowing multiple operations to be performed on a single “request”.

In the example below, the requirement is to have two different chains of responsibility depending on whether the type of the order is business-to-consumer (B2C) or business-to-business (B2B). Depending on the type, different people are allowed to handle the order.

For B2B the Senior Manager can handle any order up to $10,000, above this only the CEO can handle it.

For B2C the sales clerk can handle any non-VIP order up to $100. If the sales clerk cannot handle the order, then the team leader can handle it up to $1,000. Above that value, the senior manager can handle orders up to $10,000 in total value.

For this example, the following domain types are created to represent the order:

```
public enum BusinessModelType
{
    BusinessToConsumer,
    BusinessToBusiness
}

public class Order
{
    public BusinessModelType BusinessModel { get; set; }
    public decimal TotalOrderValue { get; set; }
    public bool IsVipCustomer { get; set; }
}
```

The next step is to define an abstract class that will handle the basic “chaining” functionality and allow the derived classes to override logic to determine if they can handle the order.

```
public abstract class OrderHandler
{
    private OrderHandler _nextHandlerInChain;

    public OrderHandler ChainTo(OrderHandler nextHandler)
    {
        _nextHandlerInChain = nextHandler;
        return _nextHandlerInChain;
    }

    public void ProcessOrder(Order order)
    {
        if (CanHandle(order))
        {
            HandleOrder(order);
        }
        else
        {
            _nextHandlerInChain.ProcessOrder(order);
        }
    }

    protected abstract void HandleOrder(Order order);

    protected abstract bool CanHandle(Order order);
}
```

Now this `OrderHandler` needs to be derived from to create the different types of order handlers, i.e. sales clerk, senior manager, etc. Each type of order handler needs to provide an implementation for: the `CanHandle` method that holds the business logic to determine if the current order can be handled; and the `HandleOrder` method that actually does the work of handling the order.

```
public class SalesClerkOrderHandler : OrderHandler
{
    protected override void HandleOrder(Order order)
    {
        Console.WriteLine("Handled by sales clerk.");
    }

    protected override bool CanHandle(Order order)
    {
        return order.TotalOrderValue <= 100m && !order.IsVipCustomer;
    }
}

public class TeamLeaderOrderHandler : OrderHandler
{
    protected override void HandleOrder(Order order)
    {
        Console.WriteLine("Handled by team leader.");
    }

    protected override bool CanHandle(Order order)
    {
        return order.TotalOrderValue <= 1000m;
    }
}

public class SeniorManagerOrderHandler : OrderHandler
{
    protected override void HandleOrder(Order order)
    {
        Console.WriteLine("Handled by senior manager.");
    }

    protected override bool CanHandle(Order order)
    {
        return order.TotalOrderValue <= 10000m;
    }
}

public class CeoOrderHandler : OrderHandler
{
    protected override void HandleOrder(Order order)
    {
        Console.WriteLine("Handled by CEO.");
    }

    protected override bool CanHandle(Order order)
    {
        return true;
    }
}
```

An `UnhandledOrderHandler` is also created, this will be added to the end of all chains to report any problems where an order has not been handled.

```
public class UnhandledOrderHandler : OrderHandler
{
    protected override void HandleOrder(Order order)
    {
        Console.WriteLine("Order not handled, flag error in system.");
    }

    protected override bool CanHandle(Order order)
    {
        return true;
    }
}
```

Next, an `OrderProcessingService` class is created to encapsulate processing an order. This “service” creates the appropriate chain of responsibility depending on the business type. Client code starts processing the order through the chain by calling the `Process` method.

```
public class OrderProcessingService
    {
        public void Process(Order order)
        {
            var orderHandlingChain =
                CreateChainOfResponsibility(order.BusinessModel);

            orderHandlingChain.ProcessOrder(order);
        }

        private OrderHandler CreateChainOfResponsibility(
            BusinessModelType model)
        {
            if (model == BusinessModelType.BusinessToBusiness)
            {
                return CreateB2BOrderChain();
            }

            return CreateB2COrderChain();
        }

        private OrderHandler CreateB2COrderChain()
        {
            var startOfChain = new SalesClerkOrderHandler();

            startOfChain.ChainTo(new TeamLeaderOrderHandler())
                        .ChainTo(new SeniorManagerOrderHandler())
                        .ChainTo(new UnhandledOrderHandler());

            return startOfChain;
        }

        private OrderHandler CreateB2BOrderChain()
        {
            var startOfChain = new SeniorManagerOrderHandler();

            startOfChain.ChainTo(new CeoOrderHandler())
                        .ChainTo(new UnhandledOrderHandler());

            return startOfChain;
        }
    }
```

The Chain of Responsibility pattern helps to keep software soft because it allows the Open Closed Principle to be respected as a new order handler can be created and inserted into the pipeline without significant modification of existing code. Only the `CreateC2COrderChain` or `CreateB2BOrderChain` methods need to be modified to inject the new handler.

Chain of Responsibility is useful when a simpler Strategy+Factory gets too complex and/or when the business logic is complex, or when business rules change frequently. In the example above it is easy to create new chains, for example if the business wanted different chains for B2B-online and B2B-postal orders. Another example would be the CEO not wanting to be involved in any order and letting their Senior Manager handle all orders.

Readability - Writing Code for Humans

One of the more subtle and important realisations is that code is written for people, not compilers.

This may sound strange at first, of course the code *gets compiled* so it can do something useful by actually being executed, but the compiler does not have to maintain the code over time - humans do.

If software is to be as soft as possible, then code needs to be as easily understood as possible. Part of this ease of understanding comes from how **readable** it is.

Readable code (also referred to as *clean code*) can be considered to have a number of qualities:

- Makes crystal clear the programmer's intent
- Makes it easy for bugs to stand out
- Can be easily understood and changed by other programmers
- Creates fewer surprises and annoyances when being read

One of the simplest and most elegant definitions comes from Grady Booch who among other things says: "*clean code reads like well-written prose*". This implies that when reading well-written code, the syntax falls away and the brain instead sees the "*picture of intent*" of the programmer who wrote it. Another way to think about it, is that when reading well-written code there is no sense of *friction*, but rather the eyes and mind effortlessly glide through it.

More readable code helps keep software soft because it reduces the mental effort required to understand it and therefore be able to change it safely.

The remainder of this chapter gives some practical examples on how the readability of source code can be improved.

Comments

Comments should always **add value** to the person reading the code. Comments that do not add value should be removed because they give the eye and brain more to process, only to be discarded or mistrusted by the reader.

Imagine reading a favourite novel, but with every sentence preceded by comments on why the author wrote it.

Don’t Explain The Language

Comments that explain how the language operates do not add value. It should be assumed that the reader knows the language. The exception to this may be if some code is utilising some highly esoteric, little known or undocumented construct or syntax.

XML Comments

Remove XML comments that add no value. For example, the following comment just adds noise that the reader must ignore:

```
/// <summary>
/// Adds 2 integers
/// </summary>
/// <param name="a"></param>
/// <param name="b"></param>
/// <returns></returns>
public int Add(int a, int b)
{
        return a + b;
}
```

Notice that the summary offers nothing more over the value of the method name and the parameters do not give any information at all. This is a comment that adds no value over what the reader can get from the code itself.

Good XML comments can provide value and provide additional IntelliSense information to consumers of the code.

Excessive XML comments can help identify areas where the naming of things can be improved. For example the following method does not have a very descriptive name nor a very well-named parameter:

```
/// <summary>
/// Process a customers order
/// </summary>
/// <param name="id">The id of the customer</param>
public void Process(int id)
{
        // etc.
}
```

If the method and parameter names are improved, the comments could be removed completely:

```
public void ProcessCustomerOrder(int customerId)
{
        // etc.
}
```

If the code is going to be provided as a library to people to use as an API, or there is a need to auto-generate documentation, then XML comments may still need to be provided.

Do Not Comment Every Line

There is no need to have a comment above every line of code:

```
public void ContrivedFindNameExample(string[] names)
{
        // need an integer so I can use it as a counter, set it to
        // zero because we want to start searching at the
        // first name
        int counter = 0;

        // need a Boolean to indicate if we've found "Jason" yet,
        // initially this is obviously going to be false
        bool isJasonFound = false;

        // loop through all the names until "Jason" is found
        while (!isJasonFound)
        {
                // set the current name by looking in the array
                string currentName = names[counter];

                // Check if we have found Jason yet
                if (currentName == "Jason")
                {
                        // We have found Jason
                        isJasonFound = true;
                }

                // add one to counter so when the loop is finished we
                // know how many loops we did and what index we are
                // looking at in the array
                counter++;
        }
}
```

Naming Booleans

The readability of logical constructs such as an `if` statement can often be improved by the introduction of a well-named, intermediary variable. Consider the following code fragments:

```
if (ageInYears == 40 && isEyeColorBlue && firstName == "Julia")
{
        // etc.
}


bool isValid = ageInYears == 40 && isEyeColorBlue && firstName == "Julia";

if (isValid)
{
        // etc.
}
```

Whilst these two fragments are functionally equivalent, examine the readability. The later version introduces an intermediary variable `isValid` which simplifies the reading of the if statement. The benefit of this can sometimes be a subtle thing to understand: a reader can now do an initial scan-read of the code to get the higher-level mental picture, *then* drill down into the detail of the logical condition if needed.

If the code is performing some action when something is *invalid*, it can make sense to rename the `isXXX` to `isNotXXX` so that the reader doesn't have to think as much:

```
if (isNotValidPerson)
{
        // etc.
}
```

Rather than:

```
if (!isValidPerson)
{
        // etc.
}
```

Again this is a subtle thing, but it can help contribute to the idea that code can be like "well-written prose".

Method Length

The number of statements in any given method have an impact on readability.

Methods with lots of lines of code tend to be harder to understand than methods with fewer lines of code.

Some people like to assign arbitrary values, e.g. "all methods should have no more that 10 lines of code", while other people have a less rigid view, e.g. "the whole method should fit on the screen". A good compromise between these two viewpoints is to set a number of lines but treat it as a guide, with the full knowledge that as the number of lines increases, readability decreases.

It is the readability of the method overall that is important, not the exact number of lines of code. If an existing method had the maximum arbitrary 10 lines, but overall readability could be improved my adding an intermediary `isXXX` boolean, then this rigid limit would not allow this to happen.

Remove Unused Code

Every line of code that needs to be read has a mental cost associated with it. One of the ways to reduce this cost is to remove any code that is no longer used or needed. This sounds like an obvious thing to do but there can sometimes be the temptation to "leave it there in case we need it in the future". Sometimes this may be a valid decision but more often it is not.

- Remove unused `using` statements
- Removed unused variables and fields
- Removed unused methods and properties
- Remove unused classes
- Remove unused project, .dll and NuGet references
- Removed unused content files (e.g. images)

If a source control system is being used (and it should be) then these things can be retrieved and examined in the future.

Productivity tools such as ReSharper[12] can help to locate and delete "dead" code.

[12]http://www.jetbrains.com/resharper/

Nesting

Multiple levels of nested code (ifs, loops, etc.) can reduce readability.

The following code has nested `if` statements:

```
if (x)
{
        return 1;
}
else
{
        if (y)
        {
                return 2;
        }
        else
        {
                return 3;
        }
}
```

The nesting can be reduced by eliminating the `else` because the return statement will effectively "end" the method at that point:

```
if (x)
{
        return 1;
}

if (y)
{
        return 2;
}

return 3;
```

Once code gets to more than one or two levels of nesting, it should probably be examined to see if it can be refactored to improve overall readability.

Naming Variables and Parameters

In general, the smaller the scope of the variable, the more terse the name can be.

This is evident in the standard practice of using `i` as a loop counter:

```
for (var i = 0; i < 10; i++)
{
        Console.WriteLine(i);
}
```

The variable `i` has a very limited scope inside the `for` loop.

The name `i` would not however be a very readable name if it had a huge scope such as: `public static int I = 42;`.

The length of method parameter names can be related to the generality of the parameter and the method itself. For example the following `Add` method uses more verbose parameter names:

```
public int Add(int firstNumberToAdd, int secondNumberToAdd)
{
    return firstNumberToAdd + secondNumberToAdd;
}
```

This could also be written:

```
public int Add(int a, int b)
{
    return a + b;
}
```

It can be argued that in the second version, the shorter parameter names (and the method body) are more readable.

As parameters become less generalised, the names tend to need to be more verbose. For example, the parameters of the following method are poorly named:

```
public string CalculatePersonsHoroscope(int y, DateTime t, string n)
```

The parameters are *less generalised*, so terse single letter names harm readability. Contrast this with the method below that has more verbose parameter names:

```
public string CalculatePersonsHoroscope(int ageInYears,
                                DateTime today,
                                string firstName)
```

Another situation where more verbose parameter names can be important is when the name of the method itself does not make it clear what the method does.

These are just guidelines, overall readability should never be sacrificed for the simple sake of terseness.

Whitespace

Whitespace can have a huge impact on readability. The following class definition has no whitespace between members:

```
internal class NoWhiteSpaceExample
{
    private int f1;
    private int f2;
    private int f3;
    public NoWhiteSpaceExample()
    {
        // etc.
    }
    public string P1 { get; set; }
    public string P2 { get; set; }
    public void Method1()
    {
        // etc.
    }
    private void Method2()
    {
        // etc.
    }
}
```

By adding some whitespace between members and groups of members the readability is increased:

```
internal class WhiteSpaceExample
{
    private int f1;
    private int f2;
    private int f3;


    public WhiteSpaceExample()
    {
        // etc.
    }


    public string P1 { get; set; }
    public string P2 { get; set; }


    public void Method1()
    {
        // etc.
    }

    private void Method2()
    {
        // etc.
    }
}
```

Notice the single-line members (the fields and properties) do not have whitespace between them, but they do have two lines of white space between them and the next “group” of members.

The multi-line method members have a space separating them, but still have two spaces separating them from the next “group” of members.

This kind of grouping (using whitespace) helps the brain to more quickly organise the class and form a mental model. This kind of grouping and separating is based on the Gestalt design principles of Similarity and Proximity.

People have varying stylistic preferences to exactly how many whitespace lines to leave between things. The actual number is not really important as long as it helps the reader to quickly navigate between “groups” of members and is reasonably consistent throughout the codebase.

It should be noted that the greater the number of whitespace lines between groups, the greater amount of vertical scrolling may be needed in the editor/IDE.

There is also the preference that some people have to arrange members in a more usage/functionally related way, for example putting a property backing field next to the property that uses it. This is also a valid organisation structure, but if your classes are small enough (for example fit on a screen entirely) the member-grouping method may be sufficient.

Regions

Generally speaking, regions harm rather than enhance readability. If the classes are small and well coded then there should be no need to use regions to organise and hide sections of code.

With bad code (e.g. classes with thousands of lines of code) regions my be required - in this case they can offer some temporary relief and make navigating hundreds of class members a bit less intimidating.

There are coding standards in organisations that say things like "*you shall put each kind of member in their own regions*", this kind of blanket coding standard can end up with regions that contain only a single property or field: clearly this is nonsense and harms readability.

Magic Strings and Constants

There is an argument that you should only introduce a constant to represent a magic string/number if you use it more than once. Another argument is that readability can be improved by using a well named constant in all cases.

Consider the following code that does some kind of hypothetical validation:

```
public void WithoutConstants(int ageInYears, string firstName)
{
    if (ageInYears >= 40 && firstName != "Sarah")
    {
        // etc.
    }
}
```

Whilst the conditions can be inferred by reading the code, readability can be improved by stating what `40` and `"Sarah"` are:

```
public void WithConstants(int ageInYears, string firstName)
{
    const int minimumValidAge = 40;
    const string invalidFirstName = "Sarah";

    if (ageInYears >= minimumValidAge &&
                                firstName != invalidFirstName)
    {
        // etc.
    }
}
```

Now the `if` statement reads a little better. Understanding what data items are involved in determining validity is also now easier; the reader does not need to look at the actual constant values if they do not need to. This enables quicker scan-reading before drilling into the details if required.

The readability of this example can further be improved by introducing an intermediate variable:

```
public void WithConstantsAndVar(int ageInYears, string firstName)
{
    const int minimumValidAge = 40;
    const string invalidFirstName = "Sarah";

    var isValid = ageInYears >= minimumValidAge &&
                        firstName != invalidFirstName;

    if (isValid)
    {
        // etc.
    }
}
```

This code now enables an initial scan-read to be performed and the reader can quickly just focus initially on either: what happens when it is valid, what data items are used to constitute validity, or what specific values are used in the validation.

An Introduction to Architectural Styles

Architectural styles do not solve a specific domain problem, rather they provide the following:

- Describe methods of code organisation/structure, deployment, and/or communication in the system
- Provide a set of principals and a common vocabulary
- Technology and vendor agnostic view

A given system may also be composed of a number of different architectural styles.

In the remainder of this chapter a number of common architectural styles will be introduced.

Client-Server

The Client-Server style describes network-based applications. The client (for example a desktop GUI) communicates with a server (or number of servers) over a network connection.

The client initiates a request to the server to perform some work. The server responds to the client request, performs some action, and returns the result to the client. In this way the client is an active part of the system, whereas the server occupies a more passive role. The server also generally provides services to more than one instance of the client application.

The Client-Server style attempts to achieve a separation of concerns: the client is concerned with the user interface functionality; the server with business logic/data storage.

Assuming the contract stays the same between client and server, each can potentially evolve and change independently, this of course assumes that no additional features are being added, etc.

The Client-Server style can be considered a 2-layer, 2 tier style. There are 2 logical **layers** that are also deployed in two separate places or **tiers**.

The terms **tier** and **layer** and often used interchangeably. In this book the term “layer” refers to a logical separation/implementation and “tier” a physical separation/implementation.

Layered

The Layered style builds on the idea of the Client-Server style. It separates the application into a number of logical "layers". Each of these layers implements a set of related responsibilities or roles and is arranged in a vertical fashion. For example, a 3-layer application may consist of a user interface layer at the top, a business logic layer in the middle, and a data access layer at the bottom. Each layer depends upon the layer beneath it ("downwards dependencies") rather than the layer above.

The layers should be loosely coupled to each other through the use of abstractions such as interfaces.

Each layer can be implemented in a separate place, for example a 3-layer, 3-tier system, or in the same place: 3-layer, 1-tier.

Message Bus

The Message Bus style allows the different parts of the system to exist separately and be connected by a common communication channel. This communication channel is referred to as a "bus".

The different parts of the system communicate with each other by passing messages over the bus. A part of the system can send a message onto the bus, where other interested parts of the system can pick up the message and perform some action.

Because each part of the system is isolated, some common shared "understanding" is required between them. Messages must be understandable by those parts of the system that are interested in them, in this way the sender and receiver may share this common understanding by way of some message schema definition or other common contract.

While the Message Bus style can implemented on a single tier, it is more usual for different parts of the system to reside on different tiers.

Service Oriented Architecture

The Service Oriented Architecture (SOA) style provides system functionality by way of a set of services that other parts of the system (e.g. client applications) can use. Services in the SOA style encapsulate business processes or features that can be "called" via standard communication methods from other parts of the system.

Each of the services that make up an SOA are deployed and maintained independently of other services, for example the "stock service" and "billing service" are autonomous of each other and can be updated independently (assuming no change in interface/contract takes place). Services can be distributed across the network and/or deployed locally.

Consumers of services should not know or depend on the internal implementation details of the service, instead the service and consumer share a common contract/schema/understanding that represents the actions that the service can perform.

Onion Architecture

The Onion Architectural style as coined by Jeffrey Palermo[13] is a response to some of the potential drawbacks of the Layered style. Rather than layers building vertically on each other, the Onion style represents the architecture as a number of concentric rings (imagine the rings in a cut onion). Dependencies are formed from the outer rings to the inner rings, but outer rings can call into *any* of the inner rings, not just their immediate neighbours.

At the core of the Onion style, is the domain object model that represents the business domain. Wrapped around this core is the "Object Services" layer that provides services such as `IFooRepository`. Next, the "Application Services" layer wraps around the Object Services layer to provide services to the user interface and also to infrastructure such as a database or file system. Contrast this to the Layered style where the data access (infrastructure) layer is the bedrock on which the rest of the application sits. In the Onion style, this data layer exists on the periphery and the core domain model has no dependency on it. This means that outer layers such as the user interface and database (infrastructure) can change without needing to change the application core, e.g. the domain model.

Event Sourcing

The Event Sourcing style models the changing state of the system as a series of stored events. The current state of the system is that of each of the events applied in the sequence in which they occurred.

For example, the current balance of a bank account is the result of all the transactions (events) that have occurred on the account:

- +$100 : Initial Account opening deposit
- -$10 : ATM cash withdrawal
- -$40 : Online purchase
- +$20: Branch cheque deposit

When the above events are played back in chronological order, the current balance of the account (current state) can be determined: 100-10-40+20 = $70. The balance at a given point in time can also be determined by "playing back" the events up to a specific date.

The current state does not have to be calculated every time the system is queried. The current state can be stored (for example in a relational database, or in-memory cache) and can be updated as often as is deemed necessary.

When the change in system state is stored as a series of events, the "event stream" can be "projected" into a number of other different representations and perhaps stored in systems such as a relational

[13]http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

database, file, or document database. Each of these representations can make use of differing parts of event data and/or different time periods to provide insights into the business. Because each change in state is captured, Event Sourcing has the potential to be able to provide business insight for reports that are thought up in the future, without having to architect the system to support them now.

Part 2: Testing

Part 2 of this book explores topics relating to the effective testing of software.

It covers topics (such as TDD) that are not wholly concerned with testing but that feature testing as a part of the overall activity.

The main focus is on tests that can be automated, whether at the code level or by automating the actual user interface.

A good suite of automated tests helps keep software soft by providing rapid feedback when things break. The longer a software defect ("bug") exists for, the more costly it becomes.

Tests can become a costly maintenance overhead, which is why it is important to take a holistic view of testing, a subject which is also covered in these chapters.

An Introduction to Unit Testing with xUnit.net

Unit testing is the execution of discrete pieces of code and then checking (asserting) to see if it behaved as expected.

The purpose of having a suite of unit tests (or any type of test) is to increase the level of confidence that the system is behaving as expected.

The distinction of “**unit**” is important: a unit test should only test a small discreet piece of code (e.g. a class/method) and not rely on concrete dependencies such as other non-primitive objects or external things such as databases and files.

Qualities of Good Unit Tests

In addition to only testing a discreet “unit”, good unit tests posses a number of other qualities as outlined below.

Isolated and Independent

A given unit test should be able to be run at any time and in any order relative to other tests. When run, it should not rely on the result/state of a previous test having been executed. Unit tests should also not rely on being run on a specific server or network.

Repeatable

A given test should either pass or fail regardless of when it is run. The test should be able to run an indefinite number of times, each time producing the same result (given that that no changes are made to the codebase).

Valuable

Any given unit test should add some value, i.e. if the test fails then the knowledge that “*something is broken over there*” should be of value to the development team. An example of a test that would add no value would be testing automatic property getters and setters; these are language features and should not be unit tested. Unit tests should only test custom code, not 3rd party or platform framework code.

Fast

Unit tests should execute quickly. They should be able to be run at any time by the developer and not hold them up for too long before they can continue coding. Unit tests that take too long to execute are likely to not be run as often as they should. This can result in defects existing for longer than they should.

As a general rule, an individual unit test should execute in less than about half a second, though the specifics of the codebase and problem domain should be taken into account.

Trustworthy

The result of a test should be 100% reliable and the development team should not feel the need to doubt the results.

Readable

As with production code, test code should be readable and of as high a quality as production code.

Testing Frameworks

It is possible to create a suite of unit tests without using a testing framework. However, testing frameworks provide a set of tools and language constructs that make writing and executing tests easier. IDE and plugin support also make it easier to run tests that utilise these testing frameworks.

Popular unit testing frameworks for .NET include NUnit, MSTest and xUnit.net.

About xUnit.net

NUnit and MSTest are possibly the current "de facto" unit testing frameworks in use today. xUnit.net was created to address certain concerns the authors had with "some very clear patterns of success (and failure) with the tools we were using" (Brad Wilson[14]).

For a comparison of testing framework attributes see Appendix C.

The design of xUnit.net attempts to align itself more closely with the .NET framework and language features (such as Generics and Anonymous Delegates) and attempts to codify good testing practices in its design.

Another design goal of xUnit.net was to create a testing framework that aligns nicely with Test Driven Development but can also be used in more general testing scenarios.

[14]http://xunit.codeplex.com/wikipage?title=WhyDidWeBuildXunit&referringTitle=Home

Using xUnit.net

Installation

Create a new Visual Studio Visual C# Class Library project called "XUnitTestingDemo".

The easiest way to install xUnit.net is to use NuGet:

- Right click on the "XUnitTestingDemo" project and choose "Manage NuGet Packages"
- Click "Online" and use the search box to search for "xUnit.net"
- Click the Install button

The xUnit assemblies are now referenced in the project.

Creating Something to Test

Add a new class called "Calculator" that has the following content:

```
namespace XUnitTestingDemo
{
    class Calculator
    {
        public int Add(int a, int b)
        {
            return a + b;
        }

        public int Divide(int number, int by)
        {
            return number / by;
        }
    }
}
```

This is the code that will be unit tested by xUnit.net.

Create the First Test

The next step is to create a class that will contain the test code. This test class will end up containing a number of methods, each method representing a single unit test for one small part of the `Calculator`.

xUnit.net uses a `[Fact]` attribute to designate a method as a test. Whenever a test runner (see below) encounters a `[Fact]` it will run that code and report whether or not it passed.

Create a new public class called "CalculatorTests", and add an initial test:

```
using Xunit;

namespace XUnitTestingDemo
{
    public class CalculatorTests
    {
        [Fact]
        public void ShouldAdd()
        {
            var sut = new Calculator();

            var result = sut.Add(1, 1);

            Assert.Equal(2, result);
        }
    }
}
```

`sut` is a naming convention that stands for System Under Test and helps to quickly identify the thing that is being tested.

Apart from the `[Fact]` attribute and the `using Xunit;` statement, the only other line of code related to xUnit.net itself is the `Assert.Equal(2, result);`. This line instructs xUnit.net to compare the two values (`2` and `result`) and check if they are equal. If this assert fails (for example if the add method was accidentally subtracting numbers), the test will fail when it is run and the test runner will notify the programmer of this failure.

Running xUnit.net Tests

There are a number of ways that xUnit.net tests can be run.

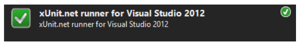
Running Tests in Visual Studio 2012

In Visual Studio 2012 an extensible test runner was introduced that allows the developer to choose from a number of testing frameworks.

To install the xUnit.net test runner:

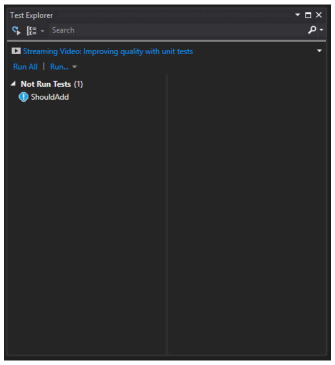
- In Visual Studio 2012 go to the Tools menu and choose Extensions and Updates
- Click Online
- In the search box, search for xUnit

- Install the “xUnit.net runner for Visual Studio 2012”


xUnit.net runner for Visual Studio 2012
xUnit.net runner for Visual Studio 2012

Installing the Visual Studio 2012 xUnit.net Test Runner

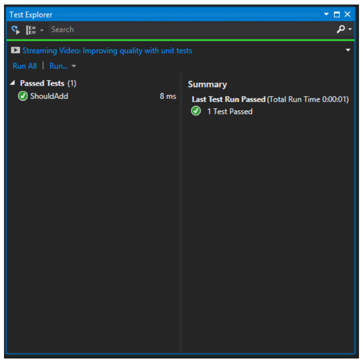
To run the `ShouldAdd` test created above: in Visual Studio 2012 go to the Test menu, Windows, Test Explorer, this will open the window as shown below:



The Visual Studio 2012 Test Explorer window

Ensure that the project has been built or the test may not appear.

To run the `ShouldAdd` test, click the Run All button. The xUnit.net test will now execute and the test will pass:


Test Explorer
Search
Streaming Video: Improving quality with unit tests
Run All | Run...
Passed Tests (1)
ShouldAdd 8 ms
Summary
Last Test Run Passed (Total Run Time 0:00:01)
1 Test Passed

Test Results in the Test Explorer window

Failing tests are shown with a red icon.

The Test Explorer window can be used to drill down into failing tests to get more information by clicking on the test name from the left hand list.

Other Test Runners

xUnit.net tests can be run in a variety of other ways.

ReSharper

At the time of writing ReSharper (version 7) does not support xUnit.net out of the box. ReSharper runner support is provided as part of the xUnit.net Contrib project[15] where ReSharper test runner installation instructions can be found.

TestDriven.Net

According to the xUnit project site: "The latest version of TestDriven.net automatically supports xUnit.net. No installation is required."

Command Line

Tests can be run from the command line. To run all the tests the following command line would be used:

```
xunit.console XUnitTestingDemo.dll
```

This will run all the tests inside the .dll.

The command line runner (xunit.console.exe) can be downloaded from the project site[16].

DevExpress CodeRush

The current version of DevExpress CodeRush supports the xUnit.net testing framework.

Checking the Results of Tests (Asserting)

xUnit.net provides a number of ways for checking (asserting) results are as expected.

The following tests shown the different types of assertions that xUnit.net supports:

```
using System;
using System.Collections.Generic;
using Xunit;

namespace KeepingSoftwareSoftSamples.XUnitTestingDemo
{
    public class XUnitAssertExamples
    {
        [Fact]
```

[15] http://xunitcontrib.codeplex.com/

[16] http://xunit.codeplex.com/

```
        public void SimpleAssertsThatOneValueEqualsAnother()
        {
            Assert.Equal(1, 2); // fail
            Assert.Equal("hello", "hello"); // pass

            Assert.NotEqual(1, 2); // pass
            Assert.NotEqual("hello", "hello"); // fail
        }

        [Fact]
        public void BooleanAsserts()
        {
            Assert.True(true); // pass
            Assert.True(false); // fail

            Assert.False(false); // pass
            Assert.False(true); // fail

            // Don't do this
            Assert.True(1 == 1); // pass
        }

        [Fact]
        public void Ranges()
        {
            const int value = 22;

            Assert.InRange(value, 21, 100); // pass
            Assert.InRange(value, 22, 100); // pass

            Assert.NotInRange(value, 999, 99999); // pass

            Assert.InRange(value, 23, 100); // fail
        }

        [Fact]
        public void Nulls()
        {
            Assert.Null(null); // pass

            Assert.NotNull("hello"); // pass

            Assert.NotNull(null); // fail
        }

        [Fact]
        public void ReferenceEquality()
        {
            var objectA = new Object();
            var objectB = new Object();

            Assert.Same(objectA, objectB); // fail

            Assert.NotSame(objectA, objectB); // pass
        }

        [Fact]
        public void AnIEnumberableContainsASpecificItem()
        {
            var days = new List<string>
                                 {
                                     "Monday",
                                     "Tuesday"
                                 };

            Assert.Contains("Monday", days); // pass
            Assert.Contains("Friday", days); // fail

            Assert.DoesNotContain("Friday", days); // pass
        }

        [Fact]
        public void IEnumerableEmptiness()
        {
            var aCollection = new List<string>();

            Assert.Empty(aCollection); // pass

            aCollection.Add("now no longer empty");

            Assert.NotEmpty(aCollection); // pass

            Assert.Empty(aCollection); // fail
        }

        [Fact]
        public void IsASpecificType()
        {
            Assert.IsType<string>("hello"); // pass

            Assert.IsNotType<int>("hello"); // pass

            Assert.IsType<int>("hello"); // fail
        }

        [Fact]
        public void IsAssignableFrom()
        {
            const string stringVariable = "42";

            Assert.IsAssignableFrom<string>(stringVariable); // pass

            Assert.IsAssignableFrom<int>(stringVariable); // fail
        }
    }
}
```

How Many Asserts in a Test?

There is a school of thought that suggests each test should have one, and only one assert. While this can *sometimes* make the purpose of the test clearer, it can also result in a greater number of tests that need to be maintained.

If the test code is testing behaviours not methods and the multiple asserts are all related to the specific behaviour, and the test is well-named, then multiple asserts are usually acceptable.

In general, the code (e.g. method) being tested should only be executed once in the body of a given test method, unless the test is specifically testing multiple invocations. For example, the following test contains multiple assertions and multiple uses of the `Calculator` object, if this test fails it is not immediately obvious what behaviour has caused the failure.

```
[Fact]
public void ShouldTestCalculator()
{
    var sut = new Calculator();

    var result = sut.Add(1, 1);

    Assert.Equal(2, result);

    result = sut.Divide(10, 2);

    Assert.Equal(5, result);
}
```

Checking if Exceptions have been Thrown

In addition to asserting the result/state, sometimes a test needs to check that a given exception was thrown.

To check that a specific kind of exception was thrown, the `Assert.Throws` method is used. The easiest overload of this method is to provide the specific type of `Exception` that is expected, then a lambda expression that represents the actual code that is expected to throw the exception.

The example below asserts that a `DivideByZeroException` is thrown when the `Calculator.Divide` method is called and the divisor is zero:

```
[Fact]
public void ShouldErrorWhenDivideByZero()
{
    var sut = new Calculator();

    Assert.Throws<DivideByZeroException>(() => sut.Divide(1, 0));
}
```

The following test will fail because the `Console.WriteLine` method does **not** throw an exception:

```
[Fact]
public void ExampleOfAFailingTestWhereExceptionNotThrown()
{
    Assert.Throws<Exception>(() =>
        Console.WriteLine("This Test Will Fail"));
}
```

It is also possible to get hold of the actual exception object that was thrown to make further assertions:

```
[Fact]
public void AssertingOnSpecificsOfAnException()
{
    var e = Assert.Throws<Exception>(() =>
                {
                    throw new Exception("This is a message");
                });

    Assert.Equal("This is a message", e.Message);
}
```

Explicitly asserting that *no* exceptions are thrown is unnecessary. If the code throws an unexpected exception then the test will automatically fail.

Running Setup Code Before Each Test

Sometimes there will be some code that needs to run before each and every test is executed. This kind of "per-test setup" code usually involves setting up any dependencies/fake objects that the code being tested relies upon.

In the xUnit.net documentation[17], the team state that this kind of per-test setup creates "difficult-to-follow and debug testing code, often causing unnecessary code to run before every single test is run". The main objection is that there can be code that runs before each and every test, code that may only be required by some of the tests. Also when reading tests it may be necessary to keep having to navigate to the setup method to understand what is being initialised.

When using xUnit.net, this per-test setup code can simply be placed in the constructor of the test class:

[17]http://xunit.codeplex.com/wikipage?title=Comparisons&referringTitle=Home#note2

```
public class XUnitAssertExamples
{
    public XUnitAssertExamples()
    {
        Debug.WriteLine("This will be run before " +
                        "each and every test gets executed");
    }
```

As a general rule-of-thumb, if the per-test setup code applies to each and every test, and the constructor is placed at the top of the file (so it is obvious when a reader first opens the file), then per-test setup can help to reduce code duplication.

Temporarily Stopping a Test From Running

Sometimes it is necessary to temporarily take a test out of action. Rather than commenting out the whole test, the `Skip` property of the `[Fact]` attribute can be set to a string that states why the test is being skipped.

For example, the following test will not be run until the `Skip` is removed:

```
[Fact(Skip="Need to fix x first")]
public void SkippedTest()
{
    // etc.
}
```

Tests should only be skipped for short periods of time, and un-skipped as soon as possible. In legacy codebases it is common to find a lot of skipped (ignored) tests that have not been run for ages.

Group Tests into Categories

Sometimes it is useful to categorise tests so that discrete categories can be run, rather than running the whole test suite. For example, if there are a lot of tests and they take a while to run, tests can be categorized so that the programmer can choose to run a subset of them.

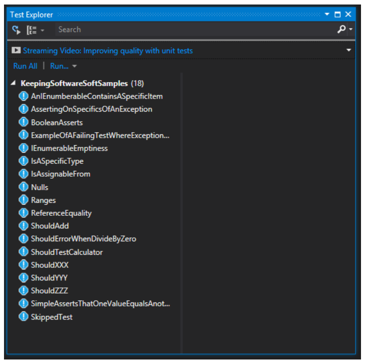
In xUnit.net, the `[Trait]` attribute is used to introduce “categories”. The `Trait` attribute allows arbitrary name/value metadata to be added to tests.

The following code shows tests split into two categories: “smoke” and “business logic”:

```
[Fact]
[Trait("Category", "smoke")]
public void ShouldXXX()
{
    // etc.
}

[Fact]
[Trait("Category", "smoke")]
public void ShouldYYY()
{
    // etc.
}

[Fact]
[Trait("Category", "business logic")]
public void ShouldZZZ()
{
    // etc.
}
```

To view the tests grouped by category:

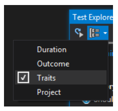
Build the project, then open the Visual Studio 2012 Test Explorer (Test menu –> Windows –> Test Explorer), the default view is to group by project:



```
KeepingSoftwareSoftSamples (18)
  AnIEnumberableContainsASpecificItem
  AssertingOnSpecificsOfAnException
  BooleanAsserts
  ExampleOfAFailingTestWhereException...
  IEnumerableEmptiness
  IsASpecificType
  IsAssignableFrom
  Nulls
  Ranges
  ReferenceEquality
  ShouldAdd
  ShouldErrorWhenDivideByZero
  ShouldTestCalculator
  ShouldXXX
  ShouldYYY
  ShouldZZZ
  SimpleAssertsThatOneValueEqualsAnot...
  SkippedTest
```

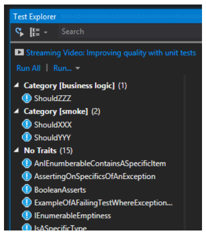
Default Test Explorer grouping tests by project

Click the grouping button (to the left of the search box) and choose “Traits”:



Grouping tests by traits

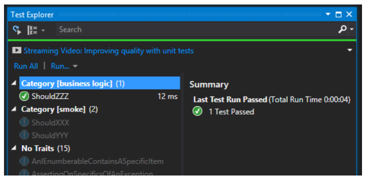
Now the Test Explorer groups the tests by categories:



```
Category [business logic] (1)
  ShouldZZZ
Category [smoke] (2)
  ShouldXXX
  ShouldYYY
No Traits (15)
  AnIEnumberableContainsASpecificItem
  AssertingOnSpecificsOfAnException
  BooleanAsserts
  ExampleOfAFailingTestWhereException...
  IEnumerableEmptiness
  IsASpecificType
```

Test Explorer grouped by traits

To run only those tests within a specific category, right-click the category group and choose "Run Selected Tests".



```
Category [business logic] (1)
  ShouldZZZ                        12 ms
Category [smoke] (2)
  ShouldXXX
  ShouldYYY
No Traits (15)
  AnIEnumberableContainsASpecificItem
  AssertingOnSpecificsOfAnException
```

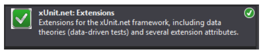
Last Test Run Passed (Total Run Time 0:00:04)
1 Test Passed

Running tests in a given category

Data Driven Tests

There are a set of extensions to xUnit.net that exist in a separate xunit.extensions.dll.

The extensions can be installed via NuGet by searching for "xunit extensions":



xUnit.net: Extensions
Extensions for the xUnit.net framework, including data theories (data-driven tests) and several extension attributes.

xUnit.net extensions in NuGet

The extensions add a number of additional features, including the ability to create data driven tests.

Data driven tests allow the same test to be executed multiple times but with different test data.

Data driven tests follow a similar approach to non data driven tests but rather than the test data being hard-coded in the test, it is provided by an "external" source.

Rather than using the `Fact` attribute, data driven tests instead use the `Theory` attribute in combination with other attributes that designate where the test data will be supplied from.

Inline Data Driven Tests

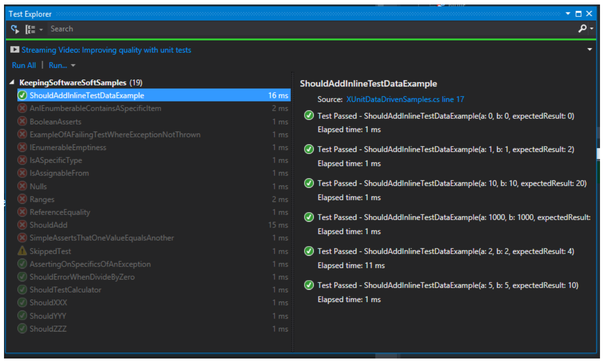
The simplest data driven tests specify the test data by using multiple `InlineData` attributes.

The `InlineData` attribute allows a list of test data items to be specified, each of these data items are passed to the parameters of the test method during test execution.

The following example shows how the `Calculator.Add` method is tested 6 times with 6 different sets of test data:

```
[Theory]
[InlineData(0, 0, 0)]
[InlineData(1, 1, 2)]
[InlineData(2, 2, 4)]
[InlineData(5, 5, 10)]
[InlineData(10, 10, 20)]
[InlineData(1000, 1000, 2000)]
public void ShouldAddInlineTestDataExample(int a,
    int b,
    int expectedResult)
{
    var sut = new Calculator();

    var result = sut.Add(a, b);

    Assert.Equal(expectedResult, result);
}
```

If this test is run in the Visual Studio 2012 Test Explorer, the follow output is produced:



```
KeepingSoftwareSoftSamples (19)
ShouldAddInlineTestDataExample                  16 ms
AnIEnumberableContainsASpecificItem              2 ms
BooleanAsserts                                   1 ms
ExampleOfAFailingTestWhereExceptionNotThrown     1 ms
IEnumerableEmptiness                             1 ms
IsASpecificType                                  1 ms
IsAssignableFrom                                 1 ms
Nulls                                            1 ms
Ranges                                           2 ms
ReferenceEquality                                1 ms
ShouldAdd                                       15 ms
SimpleAssertsThatOneValueEqualsAnother           1 ms
SkippedTest                                      1 ms
AssertingOnSpecificsOfAnException                1 ms
ShouldErrorWhenDivideByZero                      1 ms
ShouldTestCalculator                             1 ms
ShouldXXX                                        1 ms
ShouldYYY                                        1 ms
ShouldZZZ                                        1 ms
```

```
ShouldAddInlineTestDataExample
    Source: XUnitDataDrivenSamples.cs line 17
  Test Passed - ShouldAddInlineTestDataExample(a: 0, b: 0, expectedResult: 0)
    Elapsed time: 1 ms
  Test Passed - ShouldAddInlineTestDataExample(a: 1, b: 1, expectedResult: 2)
    Elapsed time: 1 ms
  Test Passed - ShouldAddInlineTestDataExample(a: 10, b: 10, expectedResult: 20)
    Elapsed time: 1 ms
  Test Passed - ShouldAddInlineTestDataExample(a: 1000, b: 1000, expectedResult:
    Elapsed time: 1 ms
  Test Passed - ShouldAddInlineTestDataExample(a: 2, b: 2, expectedResult: 4)
    Elapsed time: 11 ms
  Test Passed - ShouldAddInlineTestDataExample(a: 5, b: 5, expectedResult: 10)
    Elapsed time: 1 ms
```

Test Explorer results with data driven tests

Note that the one test has been executed six times, with different test data.

The `InlineData` attribute maps the test data by position into the parameter(s) of the test method.

Property Data Driven Tests

If the same set of test data needs to be used across multiple tests, or the test data is somehow being generated dynamically, the `[PropertyData]` attribute can be used.

Instead of getting test data from multiple `InlineData` attributes, `PropertyData` gets data from a static property that is defined in the test class.

The following example does the same as the previous `InlineData` example, but gets test data from a property called “SomeTestData”.

```
public static  IEnumerable<object[]> SomeTestData
{
    get
    {
        yield return new object[] {0, 0, 0};
        yield return new object[] { 1, 1, 2 };
        yield return new object[] { 2, 2, 4 };
        yield return new object[] { 5, 5, 10 };
        yield return new object[] { 10, 10, 20 };
        yield return new object[] { 1000, 1000, 2000 };
    }
}

[Theory]
[PropertyData("SomeTestData")]
public void ShouldAddPropertyTestDataExample(int a,
    int b,
    int expectedResult)
{
    var sut = new Calculator();

    var result = sut.Add(a, b);

    Assert.Equal(expectedResult, result);
}
```

Other Data Driven Sources

At the time of writing, the following additional data sources are also supported:

- Excel: `[ExcelData]`
- OleDb: `[OleDbData]`
- Sql Server: `[SqlServerData]`

Whilst detailed example usage of these is beyond the scope of this chapter, it should be noted that these attributes make it possible for test data to come from places *outside* the code, which potentially means that business analysts or other non-coding members of the team can help define and review test case data.

Test Driven Development

Test Driven Development (TDD) turns the previous idea of writing code, then writing tests, on its head.

The focus of TDD is not on tests *per se*, but rather on getting a **well designed** codebase. It is more about design than testing; tests are simply a (valuable) by-product of the process. For this reason, the term Test Driven Design is sometimes used instead.

Overview

With TDD, before you write any implementation code you first write a test.

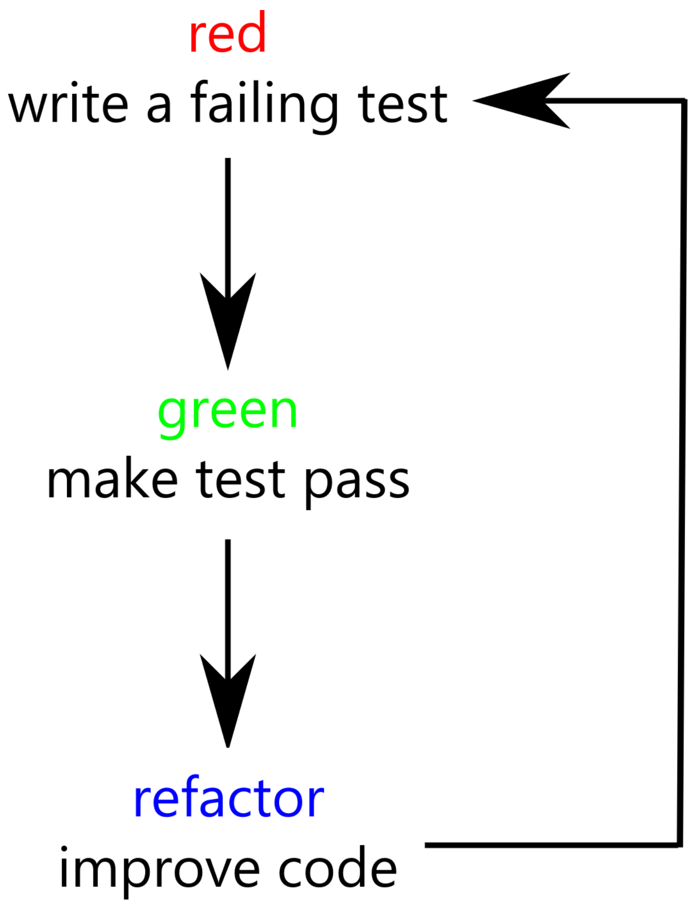
One of the mantras of TDD is:

> **“Red, Green, Refactor”**

The “**red**” part is writing a test, running it and watching it fail (it should still compile however).

The “**green**” part is writing some implementation code, running the test and watching it pass. Usually the aim is to write the least amount of code to make the test pass.

The “**refactor**” part is looking at the code that was written and looking for opportunities to improve it, and potentially other code that interacts with it.



Red, Green, Refactor

Example

The example below shows the stages some code could go through when practicing TDD. The example shows the development of a simple `Calculator` class, using the popular open source .NET testing framework called NUnit. Other testing frameworks such as MSTest and xUnit.net are also available.

Write the First Test

For some developers, getting started with the first test can be the hardest part about TDD. It is good not to spend too long thinking or talking about it and just get the first test written, the others will then start to flow.

Red

For the `Calculator` a good first test is to add two numbers together:

```
[Test]
public void ShouldAddTwoNumbers()
{
    var sut = new Calculator();

    Assert.That(sut.Add(1, 1), Is.EqualTo(2));
}
```

The variable name `sut` is a convention that stands for System Under Test. Following this convention makes it really obvious to the reader which class is being tested. It is not a requirement for doing TDD however.

At this point the code will not compile as there is no such class as `Calculator`. To get the code to compile, `Calculator` is created with the `Add` method.

```
public class Calculator
{
    public int Add(int a, int b)
    {
        throw new NotImplementedException();
    }
}
```

The code now compiles so we can run our first test. This is the "**red**" stage.

It may be tempting to skip the "**red**" (test fails) phase and just write the implementation to try and get the test to pass. The reason why it is important not to do this is that it is a sense-check of the assumptions that were made when writing the test.

Green

The aim of this phase is to write as little implementation code as possible to get the test to pass.

Writing the least amount of code sometimes feels weird as it is possible to just return whatever value the test is expecting without even implementing any meaningful logic. This is fine because future tests will evolve the implementation and design.

To get the `ShouldAddTwoNumbers` test to pass, the following implementation can be written:

```
public int Add(int a, int b)
{
    return 2;
}
```

By writing the least amount of code required to get the test to pass, it stops the tendency to think too far ahead of what *might* be needed in the future.

The test is now run and the result should be “**green**”, i.e. the test should pass.

Refactor

The “**refactor**” phase is just as important as the other two phases - but it is often forgotten in the excitement to write the next test.

In this phase, the implementing code is examined as well as the test code itself, to see if it can be improved to remove duplication, etc.

In this example, there is currently very little code to be refactored.

Write the Second Test

Now the process goes back to the “**red**” phase where the next failing test is written.

For the sake of brevity, the example will continue with the `Add` method as opposed to testing other methods such as `Subtract`, etc.

The next test could be to add two negative numbers together:

```
[Test]
public void ShouldAddNegativeNumbers()
{
    var sut = new Calculator();

    Assert.That(sut.Add(-1, -1), Is.EqualTo(-2));
}
```

It runs and fails (**red**).

The minimal amount of code is written to make the test pass (and hopefully not break the other tests):

```
public int Add(int a, int b)
{
    return a+b;
}
```

The test now passes.

It is important that all the tests are run each time, not just current test that is being worked on. This ensures that other parts of the system have not been unintentionally broken.

Before the next test is written, the implementation and test code are examined to see if anything can be refactored. There is some minor duplication in the tests in that the `Calculator` is being created in each test. To fix this, a private field is introduced to the test class to hold the instance of the `Calculator` that is being tested.

Continue Around the Loop

The process of red, green, refactor continues until there are no more failing tests that can be written.

The continuation of the TDD creation of the `Calculator` class is left as an exercise for the reader.

The current state of the code at this point is shown below.

```
using NUnit.Framework;

namespace KeepingSoftwareSoftSamples.TDD
{
    [TestFixture]
    class CalculatorTests
    {
        private readonly Calculator _sut = new Calculator();

        [Test]
        public void ShouldAddTwoNumbers()
        {
            Assert.That(_sut.Add(1, 1), Is.EqualTo(2));
        }

        [Test]
        public void ShouldAddNegativeNumbers()
        {
            Assert.That(_sut.Add(-1, -1), Is.EqualTo(-2));
        }
    }

    public class Calculator
    {
        public int Add(int a, int b)
        {
            return a+b;
        }
    }
}
```

Practicing TDD

TDD is a **practice**. It is something that becomes easier and quicker over time. It can take a while to fully “get it” and realise the benefits and the nice rhythm that can be achieved.

As a practice, it is something that should be approached for the first time with an open attitude and the mindset of: “I’ll keep getting better at it”. It can take a while to “click in” to TDD but the benefits are worthwhile.

The tips below will help to get up to speed with TDD more quickly.

Test Behaviour Not Methods

When writing the test, it is good to think from a behavioural point of view rather than a method/function point of view. This is a subtle point but it helps to write better test names. For example, instead of naming the test `ShouldAddTwoNumbers()` it could have been `Add_Method_-Should_Return_2_When_2_Plus_2()`. While this method name is verbose, the main problem with this “method focus” is that it has created a brittle dependency between the test name and the implementation code. If the `Add` method is ever renamed or refactored then the test name also needs to be changed.

Focusing on naming tests by behaviour also tends to make it easier for someone new to the code to understand what the thing being tested is supposed to do.

Starting the test name with “Should” can also help to think about testing behaviour rather than implementation details - though it is not a requirement to practice TDD.

High Quality Test Code

As with “test-last” development, test code should be of equally high quality as implementation code. The aim is to have tests that are as easy to change as the implementation code.

TDD with Pair Programming

TDD works extremely well with pair programming, where two developers program the same code, together, at the same machine. One excellent technique is to have one developer write the test, then the other developer writes the implementation to make it pass - this is sometimes described as “ping-pong programming or “ping-pong TDD”.

Why TDD?

There are a number of reasons TDD helps to keep software soft:

Loosely coupled design: Because the codebase is evolved (rather than being designed up-front), lessons learned along the way feed into future tests and hence improve the overall design. Because the focus is on unit testability, code is more easily composable, reusable, and easier to call by clients.

Inherent testability: Because you write the tests first, the implementation code by definition must be testable.

Regression Test Suite: A by-product of TDD is a set of well written, behaviourally-focused tests, with excellent test coverage. A good regression test suite is essential to help keep software soft: if there is a sense of fear about changing things because it is unknown what might break, refactoring is less likely to happen and the codebase will continue to degenerate into more of a mess.

Rhythm and Productivity: It is possible to get into a highly focused, highly productive state of being when practicing TDD. The “red, green, refactor” loop becomes second nature and there is zero friction. Over time a kind of “mental muscle memory” builds up and it is no longer necessary to think about which TDD phase is currently being executed.

Faking It - Testing Code in Isolation

It is good to be able to test code (e.g. a class) in isolation.

Testing in isolation makes test code less brittle and allows the focus to be on testing a single, well-defined thing.

Testing in isolation is Unit Testing.

To be able to test in isolation, it must be possible to provide fake versions of dependencies.

Consider the code below:

```
public class SpamBot
{
    public void SendSpam(string emailAddress)
    {
        const string message = "Buy Keeping Software Soft...";

        var g = new EmailGateway();

        g.SendEmail(emailAddress, message);
    }
}
```

This class cannot be unit tested as it has a dependency on the concrete type `EmailGateway`, not only that but it is creating the dependency itself with the `new` keyword.

The first thing to do to make `SpamBot` unit testable is to perform some refactoring to improve the code and try to adhere to some of the SOLID principles such as the Dependency Inversion Principle (see the chapter on SOLID for more information).

After refactoring the code looks like this:

```
public interface IEmailGateway
{
    void SendEmail(string toAddress, string messageBody);
}


public class SpamBot
{
    private readonly IEmailGateway _gateway;

    public SpamBot(IEmailGateway gateway)
    {
        _gateway = gateway;
    }


    public void SendSpam(string emailAddress)
    {
        const string message = "Buy Keeping Software Soft...";

        _gateway.SendEmail(emailAddress, message);
    }
}
```

An interface called `IEmailGateway` has been introduced to create an abstraction for something that knows how to send an email message.

Rather than the `SendSpam` method creating an instance of a concrete type, it now uses whatever `IEmailGateway` was given to it via the `SpamBot` constuctor.

The result of this refactor is that test code can now tell `SpamBot` to use anything that implements `IEmailGateway`. This means it can be tested in isolation by passing in some kind of fake version of `IEmailGateway`.

It is likely that if `SpamBot` were created using Test Driven Development then this refactoring would not be required as it would have better coded (more testable) from the start.

Mock, Stub, Fake, Dummy, Test Double

Over time, a lot of terminology has built up to describe what are essentially "pretend objects", i.e. the dependencies/collaborators that need to be supplied to the class we are testing.

Typically these differences are described in terms of how the “pretend object” is used.

While different people use different vocabularies for talking about these things, generally speaking the following can said to be true:

- **Stub**: A pretend object collaborator that fulfils some dependency and provides pre-defined answers/data (state) to the thing being tested. Does not usually provide for interaction/behavioural expectations/asserts.
- **Mock**: A pretend object collaborator that fulfils some dependency and provides for interaction/behavioural expectations/asserts such as “assert that my code called the mock SendEmail method with the subject “Keeping Software Soft”.

The differentiation between mocks and stubs can make it difficult for those new to using pretend objects in their tests to get started. For this reason, the more generic term “*fake*” will be used for the rest of this chapter.

An Example of Faking

Returning to the `SpamBot` example above, a test can now be written, supplying a fake `IEmailGateway`.

One way of supplying a fake would be a hand-coded fake class that implements `IEmailGateway`, but creating hand-coded fakes for all dependencies in all test suite is a big overhead. It also does the opposite of keeping software soft because the hand-coded fakes now also need to be manually changed when implementation code is refactored.

An alternative to hand-coding fakes is to use a *mocking framework* to automatically generate them.

A mocking framework will dynamically generate an instance of a type that can then be supplied as a dependency to the code that is being tested.

In the .NET space, popular free open source frameworks include Rhino Mocks and Moq (pronounced “*mock-you*” or sometimes just “*mock*”).

The example below shows how Moq is used to create a fake `IEmailGateway` which is then passed to a new `SpamBot`.

The method that is being tested (`SendSpam`) is called and then `Verify` that `SpamBot` called the `SendEmail` method on the fake exactly one time.

```
[Test]
public void ShouldSendASpamEmail()
{
    var fake = new Mock<IEmailGateway>();

    var sut = new SpamBot(fake.Object);

    sut.SendSpam("x@dmwadma.com");

    fake.Verify(x => x.SendEmail(It.IsAny<string>(), It.IsAny<string>()),
                      Times.Once());
}
```

This kind of test is sometimes called *behaviour verification* testing or *interaction* testing, i.e. ensuring that the thing being tested is interacting correctly with its dependencies/collaborators.

It is also possible to test that the thing we are testing correctly changes the state of the fake. As a (somewhat contrived) example, `IEmailGateway` is changed to add a `UseHighPriority` property that `SpamBot` should set to true when `SendEmail` is called.

```
public interface IEmailGateway
{
    void SendEmail(string toAddress, string messageBody);
    bool UseHighPriority { get; set; }
}
```

The new test looks like the following:

```
[Test]
public void ShouldSendEmailUsingHighPriority()
{
    var fake = new Mock<IEmailGateway>();

    // Tell Moq to have standard property get/set behaviour
    fake.SetupAllProperties();

    var sut = new SpamBot(fake.Object);

    sut.SendSpam("x@dmwadma.com");

    Assert.True(fake.Object.UseHighPriority);
}
```

Note that rather than making a verification on the fake, this test asserts that the final state of the fake is as expected.

Auto Mocking

In the chapter Faking It - Testing Code in Isolation the concept of faking (mocking) was introduced. Once the concept of faking is understood, it can be built on by introducing what is generally referred to as “auto mocking” or using an “auto mocking container”.

Although in the chapter on Faking It the term “fake” was used as a general term, in this chapter the term mock will be used to make the examples clearer.

Introducing Auto Mocking

An auto-mocking container is like a glue that brings together a Dependency Injection Container and a mocking framework.

Auto-mocking allows the thing being tested (the System Under Test or SUT) to be created and any of its **dependencies** will automatically be provided as mocks.

As an example, suppose the following code requires testing:

```
public interface IStringJoiner
{
    string JoinStrings(string s1, string s2);
}

public interface IDependencyB { }


public class Thing
{
    private readonly IStringJoiner _stringJoiner;
    private readonly IDependencyB _b;

    public Thing(IStringJoiner stringJoiner, IDependencyB b)
    {
        _stringJoiner = stringJoiner;
        _b = b;
    }

    public string Concat(string a, string b)
    {
        return _stringJoiner.JoinStrings(a, b);
    }
}
```

Using the xUnit.net testing framework and the Moq[18] mocking framework the following test can be written to check that: when the `Thing.Concat` method (the SUT) is called, that `Thing` delegates to the `IStringJoiner` that was passed in to the constructor of `Thing`.

```
[Fact]
public void ShouldUseStringJoiner_WithoutAutomocking()
{
    var mockJoiner = new Mock<IStringJoiner>();
    var mockB = new Mock<IDependencyB>();

    var sut = new Thing(mockJoiner.Object, mockB.Object);

    sut.Concat("a", "b");

    // Check that mockJoiner's JoinStrings method was called with
    // the correct parameters
    mockJoiner.Verify(x => x.JoinStrings("a", "b"), Times.Once());
}
```

This is a standard looking test that uses explicitly defined mocks that are manually passed to the SUT as dependencies via the SUT's constructor.

The following auto-mocking examples use the "AutoFixture with Auto Mocking using Moq" NuGet package that can be installed using the package manager in Visual Studio. When this package is installed it will also automatically install the base AutoFixture package and the Moq package.

With the relevant NuGet packages installed into the testing project, the above test can be rewritten as follows:

[18]https://github.com/Moq/moq

```
[Fact]
public void CreatingASutWithAutomocking()
{
    // Create the automocking fixture, and tell it to set itself up
    // ready to be used with the Moq mocking framework.
    var fixture = new Fixture().Customize(new AutoMoqCustomization());

    // Use the automocking fixture to create the thing to be tested
    // and automatically create it's dependencies as fakes/mocks
    var sut = fixture.Create<Thing>();

    // asserts/verify would go here
}
```

Here, at no point are explicit mocks created, they are automatically provided by AutoFixture with the line `fixture.Create<Thing>()`. This is a simplified example to demonstrate the basic concept; to make this test functionally equivalent to the non-auto-mocking test, the verification still needs to happen. The code below shows how to accomplish this using AutoFixture:

```
[Fact]
public void ShouldUseStringJoiner_WithAutomocking()
{
    var fixture = new Fixture().Customize(new AutoMoqCustomization());

    // Get a reference to the mock that will be auto supplied,
    // this enables Verifications to be made on it
    var mockJoiner = fixture.Freeze<Mock<IStringJoiner>>();

    // Use AutoFixture to create the thing to be tested, the
    // mockJoiner instance will be supplied.
    var sut = fixture.Create<Thing>();

    sut.Concat("a", "b");

    mockJoiner.Verify(x => x.JoinStrings("a", "b"), Times.Once());
}
```

Note in this example that AutoFixture's `Freeze` method is used to ensure which exact instance of a mock is provided when the SUT is created. Now when `fixture.Create<Thing>()` is called, the mock (as referenced by the `mockJoiner` variable) is passed to the SUT's constructor to satisfy the `IStringJoiner` dependency. Because the test code now has a direct reference to the mock `IStringJoiner`, it can have the same verification(s) made.

At first glance this auto-mocking test code does not look a great deal simpler or easier to understand than the non-auto-mocking test at the start of this chapter. The benefit of auto-mocking becomes evident when a SUT's dependencies are changed.

Imagine that a test suite contains hundreds of tests, if an additional dependency is added to the SUT's constructor, each of these tests has to be changed to add the new dependency wherever the SUT is being created.

As an example, suppose two new dependencies are added to the `Thing` constructor:

```
public Thing(IStringJoiner stringJoiner, IDependencyB b,
             IDependencyC c, IDependencyD d)
{
    _stringJoiner = stringJoiner;
    _b = b;
}
```

The initial test (without auto-mocking) now needs to be changed to create two new mocks and then provide them to the SUT:

```
[Fact]
public void ShouldUseStringJoiner_WithoutAutomocking()
{
    var mockJoiner = new Mock<IStringJoiner>();
    var mockB = new Mock<IDependencyB>();
    var mockC = new Mock<IDependencyC>();
    var mockD = new Mock<IDependencyD>();

    var sut = new Thing(mockJoiner.Object, mockB.Object,
                        mockC.Object, mockD.Object);

    sut.Concat("a", "b");

    // Check that mockJoiner's JoinStrings method was called with
    // the correct parameters
    mockJoiner.Verify(x => x.JoinStrings("a", "b"), Times.Once());
}
```

It is this brittleness in test code that auto-mocking seeks to remove and thus help to keep software soft.

Notice that the two auto-mocking tests **need no changes** and continue to work correctly.

For more information on AutoFixture see the project home page[19].

[19]https://github.com/AutoFixture

More About AutoFixture

In addition to enabling auto-mocking, AutoFixture also makes it easier to create "test data" when the exact values are not important: "*AutoFixture is designed to make Test-Driven Development more productive and unit tests more refactoring-safe. It does so by removing the need for hand-coding anonymous variables as part of a test's Fixture Setup phase.*" AutoFixture Project Site[20].

In the test examples above, the two strings "a" and "b" are concatenated together, however the exact strings are unimportant to the meaning of the test. Making use of AutoFixture's ability to generate "anonymous values", the test can be rewritten as follows:

```
[Fact]
public void ShouldUseStringJoiner_WithAutomocking_AnonValues()
{
    var fixture = new Fixture().Customize(new AutoMoqCustomization());

    var mockJoiner = fixture.Freeze<Mock<IStringJoiner>>();

    // create some strings to test with, the content is unimportant
    var string1 = fixture.Create<string>();
    var string2 = fixture.Create<string>();

    var sut = fixture.Create<Thing>();

    sut.Concat(string1, string2);

    mockJoiner.Verify(x => x.JoinStrings(string1,string2), Times.Once());
}
```

AutoFixture can also create anonymous numbers, seeded strings, complex types, sequences of objects, etc. For an overview see the Cheat Sheet[21].

AutoFixture and xUnit.net Custom Attributes

A more advanced refactoring when using xUnit.net and AutoFixture is to create a custom xUnit.net attribute that provides the SUT and any specific mocks automatically to the test code.

[20]https://github.com/AutoFixture/AutoFixture
[21]https://github.com/AutoFixture/AutoFixture/wiki/Cheat-Sheet

The first step in this process is to install the "AutoFixture with xUnit.net data theories" NuGet package, this provide a number of new xUnit.net attributes including a base attribute called `AutoDataAttribute`. This attribute can be derived from to create one that creates a new AutoFixture `Fixture` and uses that to provide objects to test code via test method parameters.

Once the NuGet package is installed, create a class that derives from `AutoDataAttribute`:

```
public class AutoFixtureMoqAutoDataAttribute : AutoDataAttribute
{
    public AutoFixtureMoqAutoDataAttribute() :
        base(new Fixture().Customize(new AutoMoqCustomization()))
    {
    }
}
```

This custom attribute can now be used with the xUnit.net `Theory` attribute (see xUnit chapter):

```
[Theory, AutoFixtureMoqAutoData]
public void ShouldUseStringJoiner_WithAutomockingAttribute(
    [Frozen] Mock<IStringJoiner> mockJoiner,
    Thing sut)
{
    sut.Concat("a", "b");

    mockJoiner.Verify(x => x.JoinStrings("a", "b"), Times.Once());
}
```

This greatly simplifies the "setup" or "arrange" code required in the test and improves readability in the test body.

Introduction to JavaScript Unit Testing

JavaScript code has moved beyond the client-side browser and is seeing increasing use on the server-side such as Node.js and Windows Azure Mobile Services. Just as (non-JavaScript) server-side code can be unit tested, so too can JavaScript code.

Introducing QUnit

The official site[22] describes QUnit as: "*a powerful, easy-to-use JavaScript unit testing framework. It's used by the jQuery, jQuery UI and jQuery Mobile projects and is capable of testing any generic JavaScript code, including itself!*".

QUnit essentially consists of a single JavaScript file (qunit.js) and a single CSS file (qunit.css). QUnit tests can be written by creating a simple HTML file and referencing these files and the .js file(s) containing the unit tests. An alternative is to run these tests inside Visual Studio 2012.

Creating and Running QUnit Tests in Visual Studio 2012

The example below shows how a new web project (ASP.NET Web Application) can be configured to allow unit testing with QUnit.

Creating the Project

In Visual Studio 2012, create a new ASP.NET Empty Web Application called "UnitTestingJavaScript". The "Empty Web Application" template creates a basic shell without pages.

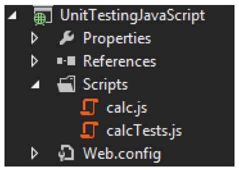
Create a new "Scripts" directory and add a new JavaScript file called "calc.js" with the following contents:

[22] http://qunitjs.com/

```
var add = function (a, b) {
    return a + b;
};

var subtract = function (a, b) {
    return a - b;
};
```

In this directory, create another JavaScript file called “calcTests.js”, this file will eventually hold the test code.

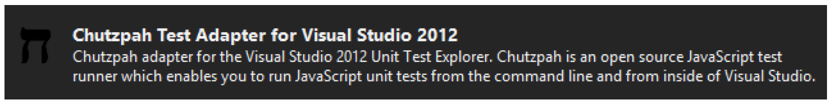
The folder structure will now look like this:



Initial Project Structure

Installing the Visual Studio 2012 Test Runner

Visual Studio 2012 allows third-parties to write “test adapters” that hook into Visual Studio. To enable QUint tests to be run, and results displayed, in Visual Studio 2012 the “Chutzpah Test Adapter for Visual Studio 2012” needs to be installed from the Tools, Extensions and Updates dialog:


Chutzpah Test Adapter for Visual Studio 2012
Chutzpah adapter for the Visual Studio 2012 Unit Test Explorer. Chutzpah is an open source JavaScript test runner which enables you to run JavaScript unit tests from the command line and from inside of Visual Studio.

Installing the Chutzpah Test Adapter

Write the First Test

In the “calcTests.js”, the first test can now be written.

The first step is to link the test .js file to the code that will be tested. At the start of the file add the following line:

```
/// <reference path="calc.js" />
```

The first test will check that the add function adds two positive integers. QUnit provides the `test` function that takes a description of the test (that is displayed in the in test runner) and a function that encapsulates the test code and asserts:

```
test("add 2 integers", function () {

    // test code

    var result = add(1, 1);

    // "assert" the the result is as expected
    strictEqual(result, 2);
});
```

The `strictEqual` assert function checks not only the value but also that the type of the expected and actual values are the same, the `equal` function can also be used to test non-strict equality. There are also the equivalent `notStrictEqual` and `notEqual` assert functions.

To run this first test, open the Test Explorer window from the Test, Windows menu. Right-click on the "add 2 integers" test and choose "Run Selected Tests".

The assert functions can also take an optional message, when the test fails this message is displayed in the test runner:

```
test("subtract 2 integers", function () {

    // test code

    var result = subtract(1, 1);

    // "assert" the the result is as expected
    strictEqual(result, 0, "It should be zero");
});
```

Rather than asserting that two things are equal, the more generic `ok` assert function can be used that takes a boolean condition plus an optional message. The add test could be written as:

```
test("add 2 integers (using ok)", function () {

    // test code

    var result = add(1, 1);

    // "assert" the the result is as expected
    ok(result === 2);
});
```

Creating and Running QUnit Tests in HTML

Running tests in Visual Studio is fine for testing JavaScript that does not manipulate the Document Object Model (DOM) in the browser. To test code that interacts with the DOM, for example writing a value into a DIV, a test HTML page can be created.

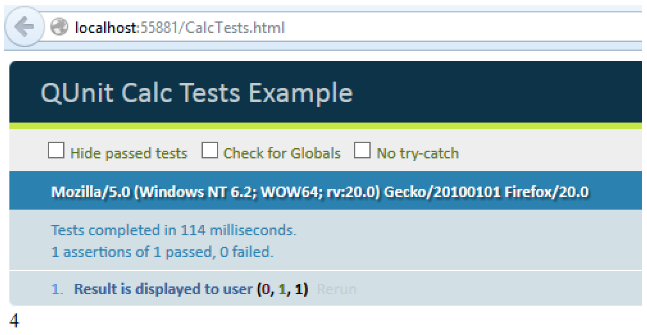
Add the following function to “calc.js”:

```javascript
var displayResultOf2Plus2 = function (element) {
    element.textContent = add(2, 2);
}
```

Add a new HTML file called “CalcTests.html”, with the following contents:

```html
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>QUnit Calc Tests Example</title>
    <link rel="stylesheet"
        href="http://code.jquery.com/qunit/qunit-1.11.0.css">
    <script src="http://code.jquery.com/qunit/qunit-1.11.0.js"></script>
    <script src="Scripts/calc.js"></script>
    <script>

        // The tests here are inline, but could be included in
        // an external js file instead
        test("Result is displayed to user", function () {

            var resultElement =
                document.getElementById("calcResultOutput");

            displayResultOf2Plus2(resultElement);

            equal("4", resultElement.textContent);
        });
    </script>
</head>
<body>
    <!-- the qunit div is required to display test results -->
    <div id="qunit"></div>

    <div id="calcResultOutput"></div>
</body>
</html>
```

Save these files and then open "CalcTests.html" by right-clicking on it in Solution Explorer and choosing View In Browser. You should see the result as shown below:


localhost:55881/CalcTests.html

QUnit Calc Tests Example

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch

Mozilla/5.0 (Windows NT 6.2; WOW64; rv:20.0) Gecko/20100101 Firefox/20.0

Tests completed in 114 milliseconds.
1 assertions of 1 passed, 0 failed.

1. Result is displayed to user (0, 1, 1) Rerun

4

QUnit HTML Output

Note that the "4" has been output to the test DIV at the bottom of the page.

An Introduction to Automated Functional UI Testing for Web Applications

Automated Functional User Interface Tests (AFUITs) are an important part of keeping software soft. They allow regression defects to be found more quickly than a human performing manual UI regression testing. They are not however a complete replacement for manual human testing or unit/integration tests.

AFUITs are about testing the **functional** behaviour of the system via the UI, they are not about verifying the look and feel of UI elements such as fonts, colours, and alignment.

Testing through the UI is a lot slower than unit testing (the UI needs time to navigate and redraw) so typically only a subset of functional scenarios can be economically tested. For example, testing some kind of check digit validation for a membership number that has (for arguments sake) a hundred different test cases, is best handled at the unit test level.

The exact amount of functionality covered by AFUITs will be influenced by:

- How much effort the team is willing to spend
- Technical ability of testers
- Complexity/size of the system under test (SUT)
- Number/complexity of the SUT's external dependencies (mainframes, services, databases, etc.)
- Risk

Without due care, AFUITs can easily become brittle and create a higher maintenance overhead in the long term.

The point of AFUITs is to catch defects at a **higher abstraction level**, namely that of a user actually interacting with the system, just as they would do in the "real world", and treating the application as a black-box.

Record-Playback Versus Coded Automation

There are many vendors offering silver bullet solutions to creating AFUITs that do not require the "testers" to have any programming (or even testing) knowledge. They usually take the form of a tester clicking a "record" button, interacting with the application and hitting the "stop" button. During this time, the software records the interactions into a script (button presses, typing, etc.) and allows the test script to be "played back".

While these tools allow anyone to create a test, they can create long term maintenance problems. For example, a change to a button's name/id may require whole sets of test scripts to be re-recorded as there is no common "model" of the pages/screens/application. Record-Playback tools can also record a lot of irrelevant details into the test script, making them hard to manually edit.

With these tools, the test scripts tend to live in their own private repositories rather than sitting alongside the source code in the version control system. They are thus not able to be easily branched and merged with the different versions of the application.

It should be noted that some vendors are now extending these tools to enable mocking out external dependencies such as databases or mainframes. This can be useful, but a well architected system should be able to be configured with fakes anyway, which can make these aspects of vendor tools less relevant. For badly architected, tightly coupled legacy applications these vendor tools may be the only option available if there is no appetite to improve the software itself. The problem is that rather that fixing the problem itself (i.e. the software), another problem is created: now there is a hard to change system **and** a hard to change set of tests.

An alternative to record-playback style tests are coded automation tests. These are created by programmers or automation test specialists and are defined in code. There is no real-time recording of steps as a user interacts with the system.

Coded UI tests can:

- Use the full power of the programming language and IDE tooling
- Re-use code within the SUT to do test data/system setup (e.g. data access code)
- Sit alongside SUT source code and be branched and merged like any other code
- Have lower maintenance costs long-term: test code can be refactored and kept soft like any other code

An Example of Coded AFUITs Using WatiN

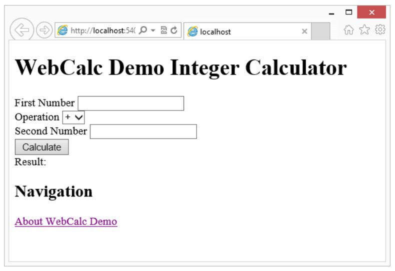
WatiN[23] (pronounced "what-in") is an open source library for .NET and stands for "Web Application Testing in dotNet". It allows tests written in .NET languages to manipulate Internet Explorer and simulate a user interacting with web pages such as typing text and clicking buttons.

WatiN is usually used alongside a testing framework (xUnit.net, NUnit, MSTest, etc.) so that existing test runners can be used to run them.

[23]http://watin.org/

The Demo Application

The demo application "WebCalc Demo Integer Calculator" used in the examples below allows simple addition and division of integer numbers. It recognises a divide-by-zero situation and displays an error message. The main page also has a link to an about page.



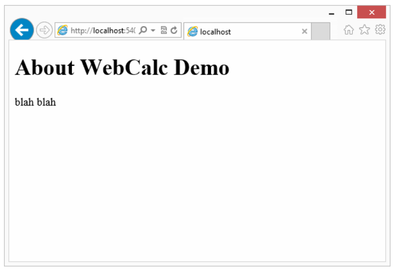
WebCalc Demo Integer Calculator

First Number
Operation +
Second Number
Calculate
Result:

Navigation

About WebCalc Demo

Main Page



About WebCalc Demo

blah blah

About Page

Creating the First Tests

To create the functional UI test suite:

- Create a new class library project in Visual Studio
- Install xUnit.net via NuGet (you could also use NUnit, MSTest, etc.)
- Install WatiN via Nuget
- Open References in Solution Explorer, locate "Interop.SHDocVw", change its "Embed Interop Types" property to "False"
- Create a new class called `CalculatorTests` to house the UI tests

The initial tests will be to check:

- Adding two numbers produces the correct result
- Dividing two numbers produces the correct result
- Dividing by zero produces an error message

- Navigating to the “about page” works

The `IE` object is provided by WatiN and gives the test code access to the HTML elements/DOM within Internet Explorer. A complete discussion about the WatiN API is outside the scope of this book, for more information see the WatiN website[24].

The tests assume that the website being tested is running at “http://localhost:54006/”.

These initial test scenarios are show below.

```
using System;
using WatiN.Core;
using Xunit;

namespace WebCalcFunctionalUITests.V1
{
    public class CalculatorTests
    {
        [Fact]
        [STAThread]
        public void ShouldAdd()
        {
            using (var browser =
                new IE("http://localhost:54006/Default.aspx"))
            {
                // Leave the browser open after the test completes
                // to see what's happened for demo purposes
                browser.AutoClose = false;

                browser.TextField(Find.ById("FirstNumber")).TypeText("1");

                browser.TextField(Find.ById("SecondNumber")).TypeText("1");

                browser.Button(Find.ById("Calculate")).Click();

                var result = browser.Span(Find.ById("Result")).Text;

                Assert.Equal("2", result);
            }
```

[24] http://watin.org/

```
        }

        [Fact]
        [STAThread]
        public void ShouldDivide()
        {
            using (var browser =
                new IE("http://localhost:54006/Default.aspx"))
            {
                browser.AutoClose = false;

                browser.TextField(Find.ById("FirstNumber")).TypeText("10");

                browser.SelectList(
                    Find.ById("Operation")).Option("/").Select();

                browser.TextField(Find.ById("SecondNumber")).TypeText("2");

                browser.Button(Find.ById("Calculate")).Click();

                var result = browser.Span(Find.ById("Result")).Text;

                Assert.Equal("5", result);
            }
        }

        [Fact]
        [STAThread]
        public void ShouldShowDivideByZeroError()
        {
            using (var browser =
                new IE("http://localhost:54006/Default.aspx"))
            {
                browser.AutoClose = false;

                browser.TextField(Find.ById("FirstNumber")).TypeText("10");

                browser.SelectList(
                    Find.ById("Operation")).Option("/").Select();

                browser.TextField(Find.ById("SecondNumber")).TypeText("0");

                browser.Button(Find.ById("Calculate")).Click();

                var result = browser.Span(Find.ById("Result")).Text;
                var error = browser.Span(Find.ById("Error")).Text;

                Assert.Equal("n/a", result);
                Assert.Equal("Can't divide by zero", error);
            }
        }

        [Fact]
        [STAThread]
        public void ShouldNavigateToAboutPage()
        {
            using (var browser =
                new IE("http://localhost:54006/Default.aspx"))
            {
                browser.AutoClose = false;

                browser.Link(Find.ById("aboutNav")).Click();

                Assert.True(browser.ContainsText("About WebCalc Demo"));
            }
        }
    }
}
```

Refactoring the Tests

The initial test code above contains a lot of duplication and is too brittle; if the HTML `id` of the Calculate button was changed then the top three tests would all break because WatiN will not be able to find that element on the page. To fix this all three tests would need to be changed, if there were a hundreds or thousands of tests all with this hardcoded id then the change would create a significant maintenance overhead.

One fix would be simply to replace all the hardcoded strings with constants. While this is an improvement, a better approach is to **model** each page so they can be treated in a strongly typed way from the test code.

WatiN provides a `Page` class to derive from when creating a strongly-typed representation of a page. It allows fields to be mapped to HTML controls by Id or other means. Further abstraction can be introduced by creating methods such as the `Calculate` method (below) that represent a piece of functionality that the user can perform.

```
using WatiN.Core;

namespace WebCalcFunctionalUITests.V2
{
    internal class MainPage : Page
    {
        [FindBy(IdRegex = "FirstNumber")]
        public TextField FirstNumber;

        [FindBy(IdRegex = "SecondNumber")]
        public TextField SecondNumber;

        [FindBy(IdRegex = "Result")]
        public Span Result;

        [FindBy(IdRegex = "Calculate")]
        private Button CalculateButton;

        [FindBy(IdRegex = "aboutNav")]
        private Link AboutLink;

        public void Calculate()
        {
            CalculateButton.Click();
        }

        public void NavigateToAbout()
        {
            AboutLink.Click();
        }
    }
}
```

The test code can now be simplified to:

```
using System;
using WatiN.Core;
using Xunit;

namespace WebCalcFunctionalUITests.V2
{
    public class CalculatorTests
    {
        private const string DefaultPageUrl =
            "http://localhost:54006/Default.aspx";

        [Fact]
        [STAThread]
        public void ShouldAdd()
        {
            using (var browser = new IE(DefaultPageUrl))
            {
                browser.AutoClose = false;

                var p = browser.Page<MainPage>();

                p.FirstNumber.TypeText("1");
                p.SecondNumber.TypeText("1");
                p.Calculate();

                Assert.Equal("2", p.Result.Text);
            }
        }

        // Other tests omitted for brevity
    }
}
```

There are still improvements that can be made. The abstraction level can be raised, for example by the introduction of an `Application` class that represents the web application being tested and provides common features to the test code.

```
using System;
using WatiN.Core;

namespace WebCalcFunctionalUITests.V3
{
    internal class Application : IDisposable
    {
        // In a real implementation, this would likely come from a
        // configuration file so it can be set for different
        // test environments
        private const string BaseUrl = "http://localhost:54006/";

        private readonly IE _browser;

        public Application()
        {
            _browser = new IE();

            // for demo purposes
            _browser.AutoClose = false;
        }

        public void Dispose()
        {
            _browser.Dispose();
        }

        /// <summary>
        /// Ensure page is currently being displayed in browser
        /// </summary>
        public T OnPage<T>() where T : WebCalcPage, new()
        {
            var p = new T();

            _browser.GoTo(BaseUrl + p.Url());

            return _browser.Page<T>();
        }
    }
}
```

A `WebCalcPage` class is also introduced to provide common functionality that the `Application` class can use, such as the `Url`.

```
using WatiN.Core;

namespace WebCalcFunctionalUITests.V3
{
    internal abstract class WebCalcPage : Page
    {
        public abstract string Url();
    }
}
```

The `MainPage` class can be further enhanced by creating a method to represent the *addition* feature of the page so that test code can be read at an even higher abstraction layer:

```
public void AddNumbers(int a, int b)
{
    FirstNumberBox.TypeText(a.ToString());
    SecondNumberBox.TypeText(b.ToString());
    // assumes default operation is set to addition
    CalculateButton.Click();
}
```

This now makes it possible for the test code to be simplified to:

```
using System;
using Xunit;

namespace WebCalcFunctionalUITests.V3
{
    public class CalculatorTests
    {
        [Fact]
        [STAThread]
        public void ShouldAdd()
        {
            using (var a = new Application())
            {
                var p = a.OnPage<MainPage>();

                p.AddNumbers(1, 1);

                Assert.Equal("2", p.Result);
            }
        }

        // Other tests omitted for brevity
    }
}
```

The exact level of abstract that is required will be depend on how much time the team wants to invest and the complexity of the system under test. It is also possible to increase the level of abstraction further by creating fluent interfaces to make test code readable by non-programmers; this allows business people and business analysts to be more involved in validating functional tests. For example, a fluent interface for an online shopping application could look something like:

```
Application
        .WithLoggedInCustomer(Customer.Default)
        .AddItemToCart(Item.SlrCamera)
        .Checkout()
        .OrderSummary
        .Contains(Item.SlrCamera);
```

Another feature that a a coded UI testing framework is likely to evolve is a navigation system to get to a given page via a number of other steps - such as a set of wizard style pages.

Some Other Tools

There are a number of other tools, frameworks, etc. that can be used. The list below is by no means exhaustive but represents some commonly used tools in both the record-playback and coded styles:

- Selenium[25]
- Microsoft Coded UI Tests[26]
- HP QuickTest Professional[27]
- CA LISA[28]

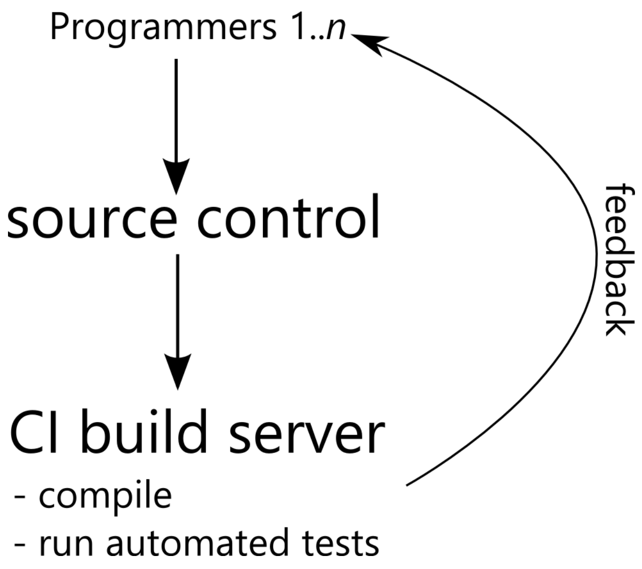
[25]http://docs.seleniumhq.org/
[26]http://msdn.microsoft.com/en-us/library/dd286726.aspx
[27]http://en.wikipedia.org/wiki/HP_QuickTest_Professional
[28]http://www.ca.com/us/default.aspx

An Introduction to Continuous Integration with TeamCity

Continuous Integration (CI) is the process of taking code from multiple programmers (on the same project) and putting their changes together often, so as find problems. CI typically involves two overall steps: ensuring all the individual changes compile when put together; then once the software compiles successfully, running automated tests against it.

CI is typically used when you have more than one developer checking in code to a common place, though it is also of use for single developers where the tests take a long time to run because code integrates with many external "services". Executing these long running tests before each commit (to source control) would reduce the progress the programmer is able to make. CI also reduces risk by allowing the deployment packages to be built in a central location rather than on the developer's machine.

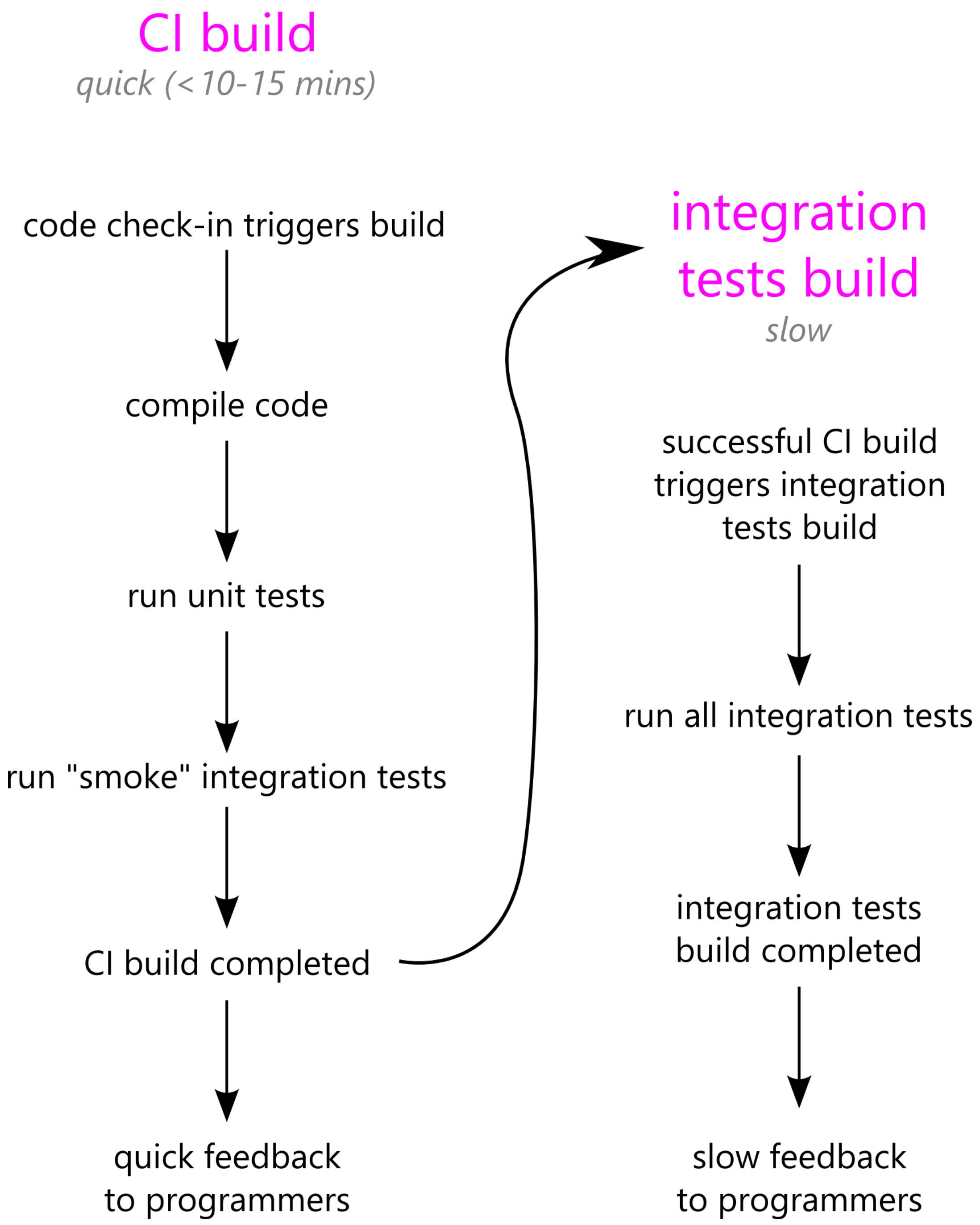
Multiple programmers should make regular commits when logical points in code writing are reached. Ideally programmers should be checking code into the "mainline" code branch at least once per day, if not multiple times per day. The core purpose of CI is to find problems sooner, so regular smaller commits help with this, though commit frequency may be less when performing a large refactoring or working on a larger feature.



CI Overview

Build Pipelines

A (CI) build pipeline is a term used to describe a series of steps that a build server performs.


CI build
quick (< 10-15 mins)
code check-in triggers build
compile code
run unit tests
run "smoke" integration tests
CI build completed
quick feedback to programmers
integration tests build
slow
successful CI build triggers integration tests build
run all integration tests
integration tests build completed
slow feedback to programmers

A Sample Build Pipeline

The diagram above outlines a sample pipeline that performs a CI phase (rapid feedback) followed by slower integration tests.

The reason for separating the integration tests into a separate build that is that a large suite of integration tests typically takes a longer amount of time to run. If the CI build included these then the feedback time would be a lot longer than just executing the (fast) unit tests.

The diagram also shows that a subset of integration tests are still run as part of the CI build. These "smoke" tests are a very limited subset of the integration tests that test high-level, key integration features that the team need to know about more quickly.

Implementing a Build Pipeline in TeamCity

TeamCity by JetBrains[29] is a "*user-friendly continuous integration (CI) server for professional developers and build engineers, like ourselves. It is trivial to setup, manage and use. It scales perfectly for a company or a team of any size*" TeamCity product page[30]. It is free for small teams with additional licensing terms available for larger teams.

TeamCity can be downloaded from the JetBrains website[31].

The remainder of this chapter demonstrates configuring TeamCity to replicate the above sample build pipeline. It assumes TeamCity has been installed locally at port 81 (the default is port 80).

The source code that will be used as a sample build solution is hosted on GitHub[32].

Getting Started - VCS Root

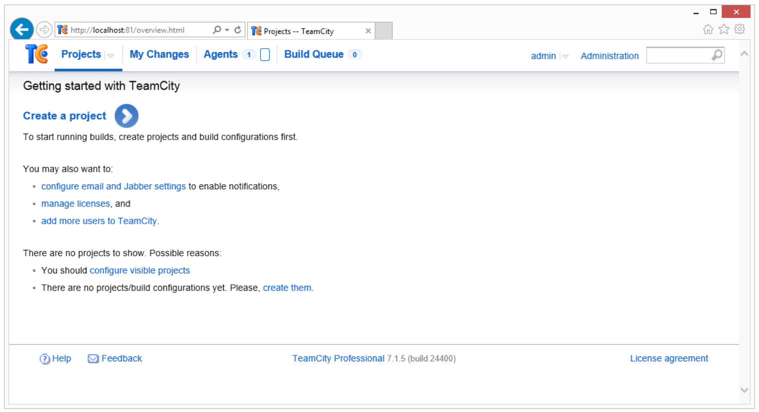
A VCS (version control system) root allows TeamCity to connect to a source code repository and download the latest changes. To add a new VCS root, open TeamCity in a browser, e.g. http://localhost:81/overview.html.

[29]http://www.jetbrains.com/

[30]http://www.jetbrains.com/teamcity/

[31]http://www.jetbrains.com/

[32]https://github.com/jason-roberts/KeepingSoftwareSoft.com_TeamCityCISample



Getting started with TeamCity

Create a project

To start running builds, create projects and build configurations first.

You may also want to:

- configure email and Jabber settings to enable notifications,
- manage licenses, and
- add more users to TeamCity.

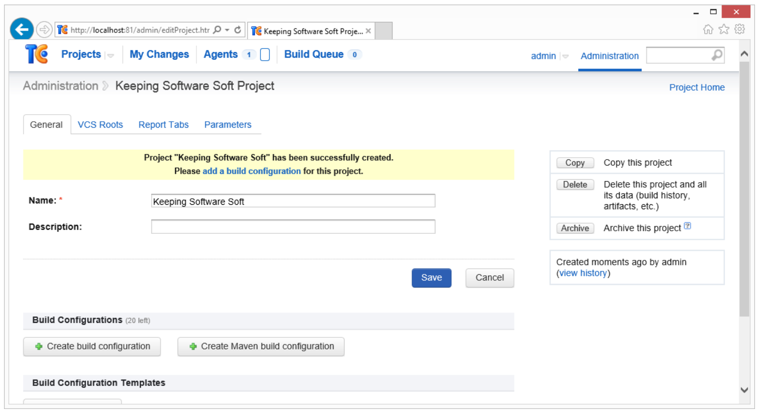
There are no projects to show. Possible reasons:

- You should configure visible projects
- There are no projects/build configurations yet. Please, create them.

Help Feedback TeamCity Professional 7.1.5 (build 24400) License agreement

A New TeamCity Install

Click “Create a project”, give it a name of “Keeping Software Soft” and click “Create”. A project in TeamCity is a related group of build configurations.



Administration > Keeping Software Soft Project

Project "Keeping Software Soft" has been successfully created.
Please add a build configuration for this project.

Name: * Keeping Software Soft

Description:

Copy this project

Delete this project and all its data (build history, artifacts, etc.)

Archive this project

Created moments ago by admin (view history)

Build Configurations (20 left)

Build Configuration Templates

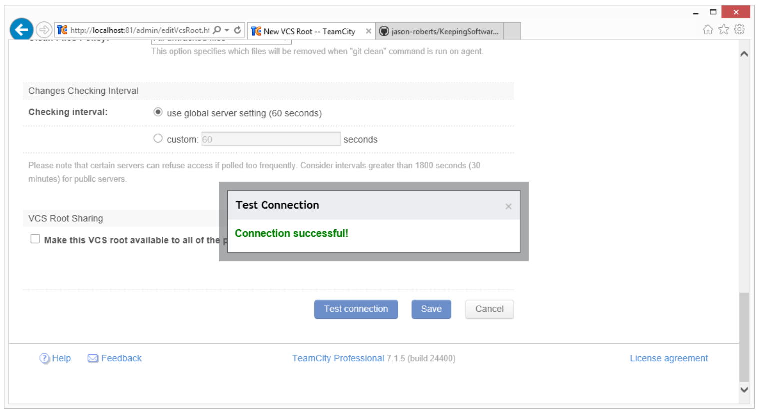
Creating a New Project

Click the “VCS Roots” tab and choose “Create VCS root”.

Choose Git as the type of root and enter the following values:

- **VCS Root Name**: Keeping Software Soft CI Sample
- **Fetch URL** : https://github.com/jason-roberts/KeepingSoftwareSoft.com_TeamCityCISample.git
- **User Name Style**: UsedId (jsmith)
- **Authentication Method**: Anonymous

Scroll down to the bottom and click Test Connection, you should see a success message:


http://localhost:81/admin/editVcsRoot.ht
New VCS Root -- TeamCity
jason-roberts/KeepingSoftwar...
Changes Checking Interval
Checking interval:
use global server setting (60 seconds)
custom: 60 seconds
VCS Root Sharing
Make this VCS root available to all of the p
Test Connection
Test connection
Save
Cancel
Help
Feedback
TeamCity Professional 7.1.5 (build 24400)
License agreement

This option specifies which files will be removed when "git clean" command is run on agent.

Please note that certain servers can refuse access if polled too frequently. Consider intervals greater than 1800 seconds (30 minutes) for public servers.

Connection successful!

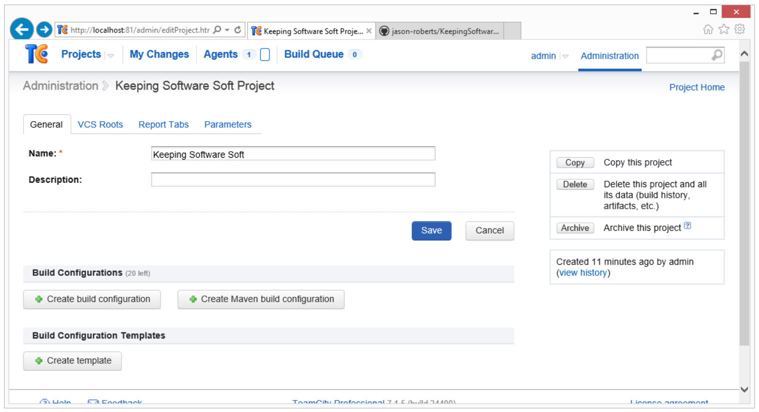
Creating a New Project

Close the success box and click save.

Creating the Initial Build Configuration

A TeamCity build configuration is a list of build steps, triggers and other settings that define an individual build.

Click the General tab:


http://localhost:81/admin/editProject.htr
Keeping Software Soft Proje...
jason-roberts/KeepingSoftwar...
Projects
My Changes
Agents 1
Build Queue 0
admin
Administration
Administration
Keeping Software Soft Project
Project Home
General
VCS Roots
Report Tabs
Parameters
Save
Cancel
Copy
Delete
Archive
Build Configurations (20 left)
Create build configuration
Create Maven build configuration
Build Configuration Templates
Create template

Name: * Keeping Software Soft

Description:

Copy this project

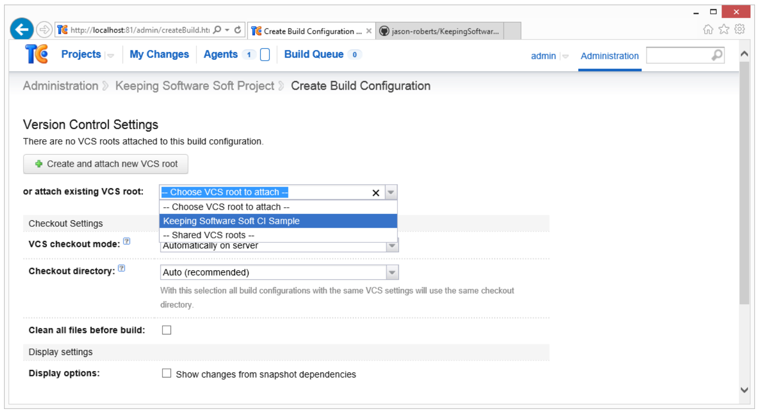
Delete this project and all its data (build history, artifacts, etc.)

Archive this project

Created 11 minutes ago by admin (view history)

The General Tab

Choose "Create build configuration" and give it a name of "CI". Click the VCS Settings button and choose the existing VCS root that was created above:



Administration › Keeping Software Soft Project › Create Build Configuration

Version Control Settings

There are no VCS roots attached to this build configuration.

Create and attach new VCS root

or attach existing VCS root: -- Choose VCS root to attach --
-- Choose VCS root to attach --
Keeping Software Soft CI Sample
-- Shared VCS roots --

Checkout Settings

VCS checkout mode: Automatically on server

Checkout directory: Auto (recommended)

With this selection all build configurations with the same VCS settings will use the same checkout directory.

Clean all files before build: ☐

Display settings

Display options: ☐ Show changes from snapshot dependencies

Setting the Build VCS Root

Click "Add build step".

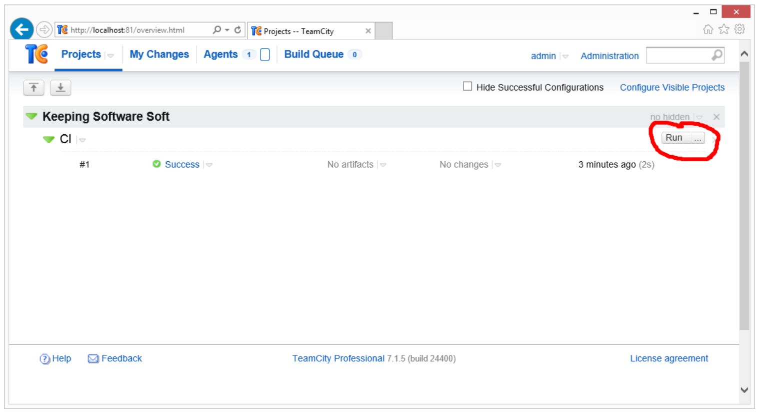
The first build step will be to compile (build) the Visual Studio solution. For the "runner type" choose "Visual Studio (sln)".

Enter the following settings:

- **Step Name**: Main sln build
- **Solution file path**: src/TeamCityCISample/TeamCityCISample.sln

Click "save", then click the "Projects" menu option at the top left.

Click the "Run button" associated with the CI build configuration, the build will now run:



☐ Hide Successful Configurations Configure Visible Projects

Keeping Software Soft no hidden

CI

#1 Success No artifacts No changes 3 minutes ago (2s)

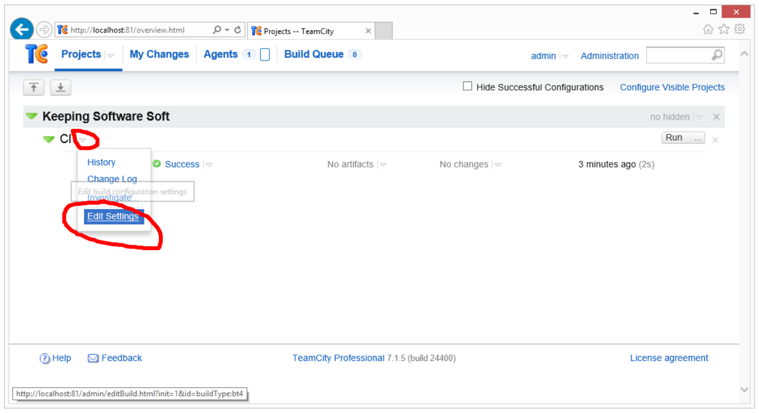
TeamCity Professional 7.1.5 (build 24400)

The First Run

Adding Unit Testing to the Build Configuration

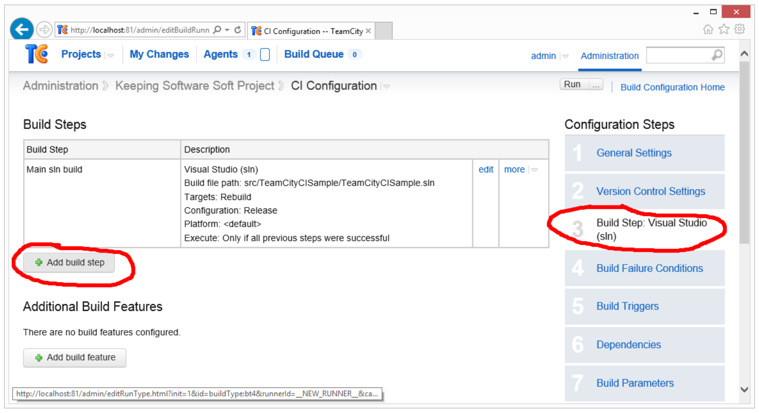
The Visual Studio solution that is being built has two (NUnit) test projects: Calculator.IntegrationTests and Calculator.UnitTests, the CI build created above will now be given a second build step to run the unit tests.

Click the drop down menu next to the CI build and choose “Edit Settings”:



Editing the CI Build Configuration

Click the “Build Step: Visual Studio (sln)” build step and click “Add build step”:


http://localhost:81/admin/editBuildRunn
CI Configuration -- TeamCity
Projects
My Changes
Agents 1
Build Queue 0
admin
Administration
Administration › Keeping Software Soft Project › CI Configuration
Run
Build Configuration Home
Configuration Steps
1 General Settings
2 Version Control Settings
3 Build Step: Visual Studio (sln)
4 Build Failure Conditions
5 Build Triggers
6 Dependencies
7 Build Parameters
Add build step
Add build feature
http://localhost:81/admin/editRunType.html?init=1&id=buildType:bt4&runnerId=__NEW_RUNNER__&ca...

Build Steps

Build Step	Description		
Main sln build	Visual Studio (sln) Build file path: src/TeamCityCISample/TeamCityCISample.sln Targets: Rebuild Configuration: Release Platform: <default> Execute: Only if all previous steps were successful	edit	more

Additional Build Features

There are no build features configured.

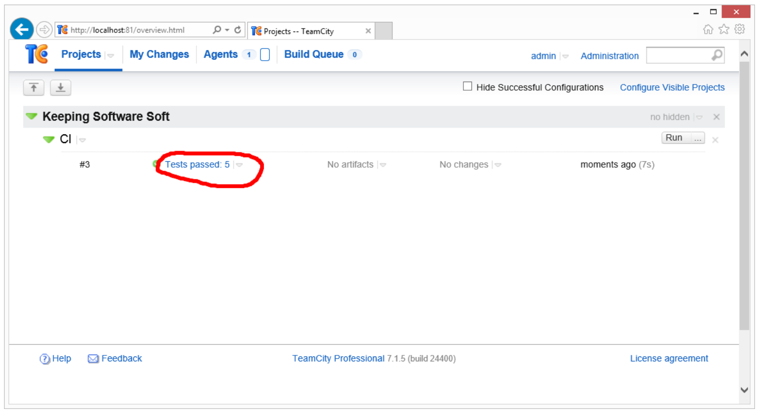
Adding a Build Step

Enter the following settings:

- **Runner type**: NUnit
- **Step Name**: Run unit tests
- **Run tests from**: **\bin**Calculator.UnitTests.dll
- **.NET Runtime**: v4.0

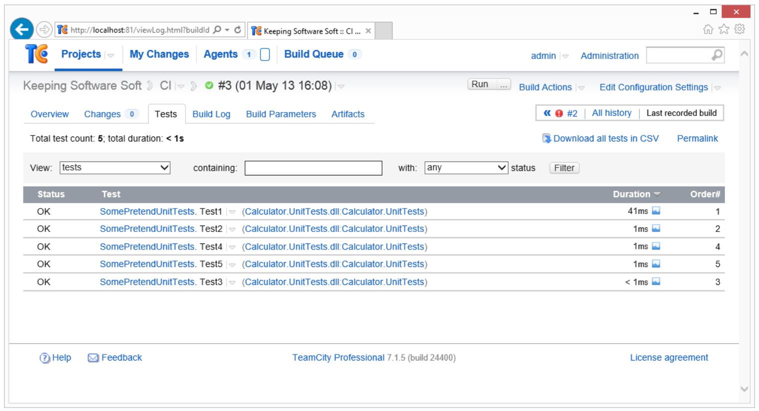
Click “Save”.

Run the build again, the results summary will now show 5 tests passed:



5 Passing Tests

Clicking the "Tests passed: 5" link will go to the build details, which enables the drilling-down into tests:



Total test count: 5; total duration: < 1s

Status	Test	Duration	Order#
OK	SomePretendUnitTests. Test1 (Calculator.UnitTests.dll:Calculator.UnitTests)	41ms	1
OK	SomePretendUnitTests. Test2 (Calculator.UnitTests.dll:Calculator.UnitTests)	1ms	2
OK	SomePretendUnitTests. Test4 (Calculator.UnitTests.dll:Calculator.UnitTests)	1ms	4
OK	SomePretendUnitTests. Test5 (Calculator.UnitTests.dll:Calculator.UnitTests)	1ms	5
OK	SomePretendUnitTests. Test3 (Calculator.UnitTests.dll:Calculator.UnitTests)	< 1ms	3

Build Test Details

Triggering an Integration Test build

When the CI build completes successfully, the next step in the build pipeline is to run the slow integration tests build.

Create a new build and call it "Run Integration tests", choose the same VCS root. When it comes to adding the build step, do not create a "Visual Studio (sln) step. The only build step required is an NUnit runner step. Configure the NUnit step with the following values:

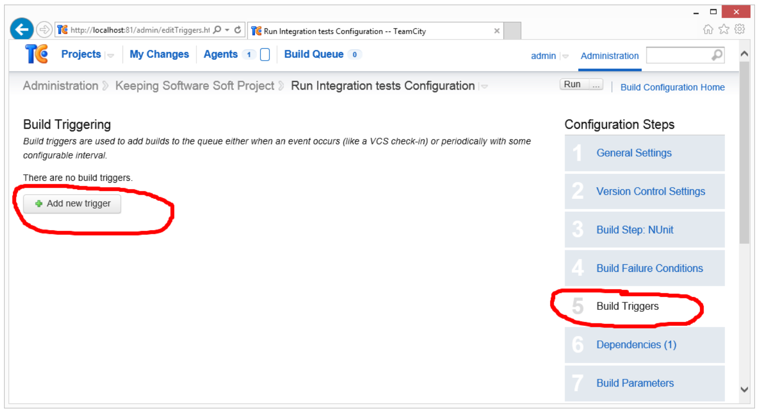
Enter the following settings:

- **Runner type**: NUnit
- **Step Name**: Run unit tests
- **Run tests from**: **\bin**Calculator.IntegrationTests.dll
- **.NET Runtime**: v4.0

The next step is to add a Build Trigger.

In addition to starting a build manually, TeamCity can be configured to start a build when certain conditions are met; among the available triggers is a “Finish Build Trigger”.

In the “Run Integration tests” build configuration, choose “Build Triggers” and click “Add new trigger”:



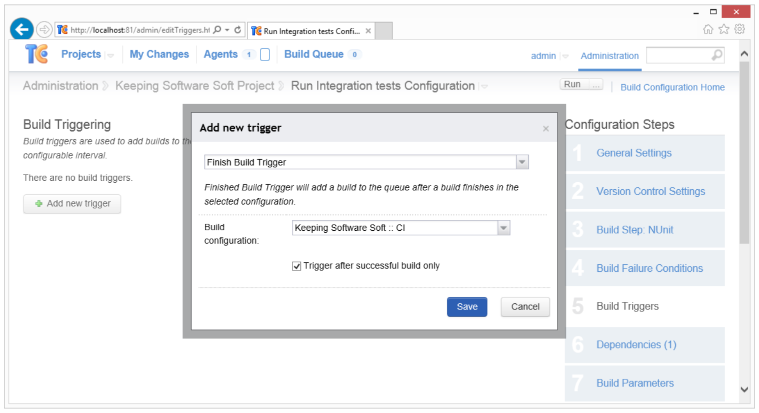
Build Triggering

Build triggers are used to add builds to the queue either when an event occurs (like a VCS check-in) or periodically with some configurable interval.

There are no build triggers.

Creating a Build Trigger

Choose “Finish Build Trigger”, set “Build configuration” to “Keeping Software Soft :: CI” and check the “Trigger after successful build only” box:



Add new trigger

Finish Build Trigger

Finished Build Trigger will add a build to the queue after a build finishes in the selected configuration.

Build configuration: Keeping Software Soft :: CI

☑ Trigger after successful build only

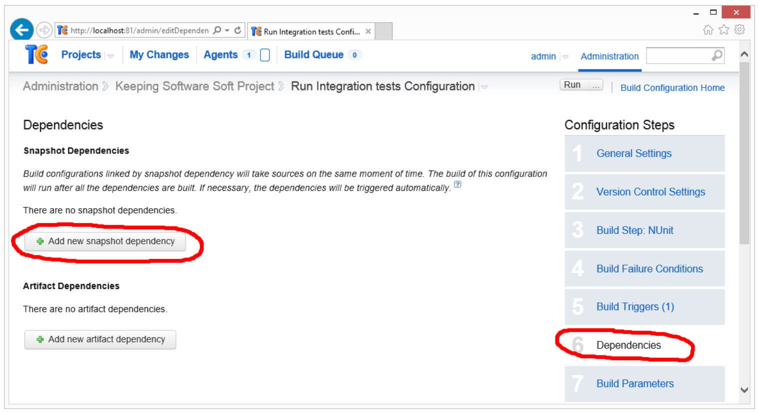
Configuring the Trigger

Click “Save”.

The next step is to tell TeamCity to use the output of the previous build. The integration tests should be run against the same set of source code and build binaries as the CI build. This is why there is no “Visual Studio (sln)” build step in the “Run Integration tests” build. If the source was downloaded and built again it is possible that between the CI build starting and the integration test build starting, some code changes could have been committed. This would result in the integration tests running against different code than the CI build, which is clearly not an ideal situation.

TeamCity has the concept of Snapshot Dependencies that allow triggered builds to use the same set of sources from a previous build.

To create a Snapshot Dependency choose "Dependencies" and click "Add new snapshot dependency":


http://localhost:81/admin/editDependen
Run Integration tests Confi...

Projects | My Changes | Agents 1 | Build Queue 0 | admin | Administration

Administration › Keeping Software Soft Project › Run Integration tests Configuration | Run ... | Build Configuration Home

Dependencies

Snapshot Dependencies

Build configurations linked by snapshot dependency will take sources on the same moment of time. The build of this configuration will run after all the dependencies are built. If necessary, the dependencies will be triggered automatically.

There are no snapshot dependencies.

Add new snapshot dependency

Artifact Dependencies

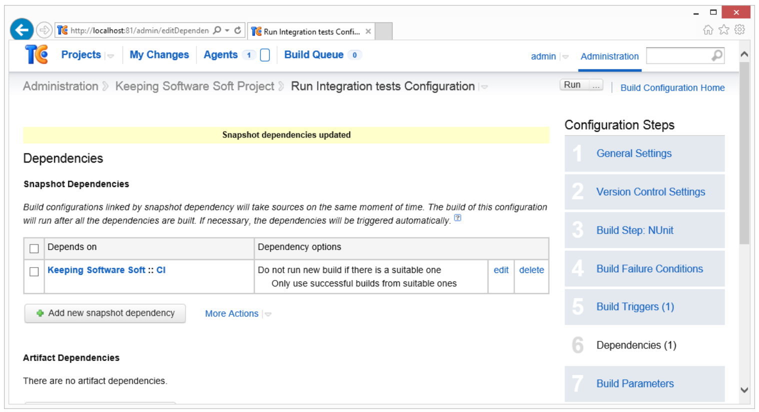
There are no artifact dependencies.

Add new artifact dependency

Configuration Steps
1 General Settings
2 Version Control Settings
3 Build Step: NUnit
4 Build Failure Conditions
5 Build Triggers (1)
6 Dependencies
7 Build Parameters

Creating a Snapshot Dependency

Choose "CI" for the "Depend on" item and click "Save":


http://localhost:81/admin/editDependen
Run Integration tests Confi...

Projects | My Changes | Agents 1 | Build Queue 0 | admin | Administration

Administration › Keeping Software Soft Project › Run Integration tests Configuration | Run ... | Build Configuration Home

Snapshot dependencies updated

Dependencies

Snapshot Dependencies

Build configurations linked by snapshot dependency will take sources on the same moment of time. The build of this configuration will run after all the dependencies are built. If necessary, the dependencies will be triggered automatically.

	Depends on	Dependency options		
☐	Keeping Software Soft :: CI	Do not run new build if there is a suitable one Only use successful builds from suitable ones	edit	delete

Add new snapshot dependency | More Actions

Artifact Dependencies

There are no artifact dependencies.

Configuration Steps
1 General Settings
2 Version Control Settings
3 Build Step: NUnit
4 Build Failure Conditions
5 Build Triggers (1)
6 Dependencies (1)
7 Build Parameters

Snapshot Dependency Created

The result of the above changes is that when the CI build runs and completes successfully, the integration test build will be automatically triggered and passed the same set of build "artefacts" that the CI build used.

Add Integration Smoke Tests to the CI Build

Referring back to the sample build pipeline diagram, during the CI build a subset of the integration tests need to be run. These "smoke" tests execute basic integration tests that confirm things are "holding together" as expected and will not fail in a disastrous way during further testing.

The integration test code is as follows:

```
using System.Threading;
using NUnit.Framework;

namespace Calculator.IntegrationTests
{
    [TestFixture]
    public class SomePretendIntegrationTests
    {
        private void SimulateLongTest()
        {
            Thread.Sleep(5000);
        }

        [Test]
        public void Test1()
        {
            SimulateLongTest();
            Assert.Pass();
        }

        [Test]
        public void Test2()
        {
            SimulateLongTest();
            Assert.Pass();
        }

        [Test]
        public void Test3()
        {
            SimulateLongTest();
            Assert.Pass();
        }

        [Test]
        [Category("smoke")]
        public void Test4()
        {
            SimulateLongTest();
            Assert.Pass();
        }

        [Test]
        [Category("smoke")]
        public void Test5()
        {
            SimulateLongTest();
            Assert.Pass();
        }
    }
}
```

`Test4` and `Test5` have an additional `Category` attribute. This is part of the NUnit testing framework that allows arbitrary categorisation of tests, in this case defining them as "smoke".

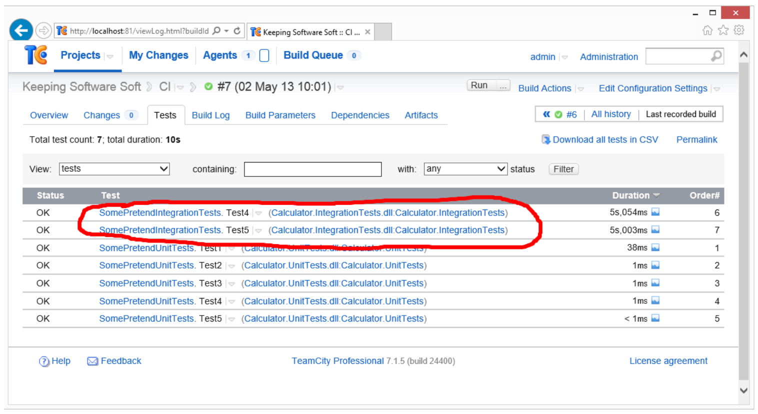
TeamCity can be instructed to run only a subset of tests based on category, rather than automatically running all tests in the dll(s).

To add the smoke tests to the CI build, edit its configuration and add a third NUnit build step.

Enter the following settings:

- **Runner type**: NUnit
- **Step Name**: Run smoke tests
- **Run tests from**: **\bin**Calculator.IntegrationTests.dll
- **.NET Runtime**: v4.0
- **NUnit categories include**: smoke

Save the changes and run the CI build, when it completes the two smoke tests have now been executed in addition to the unit tests:



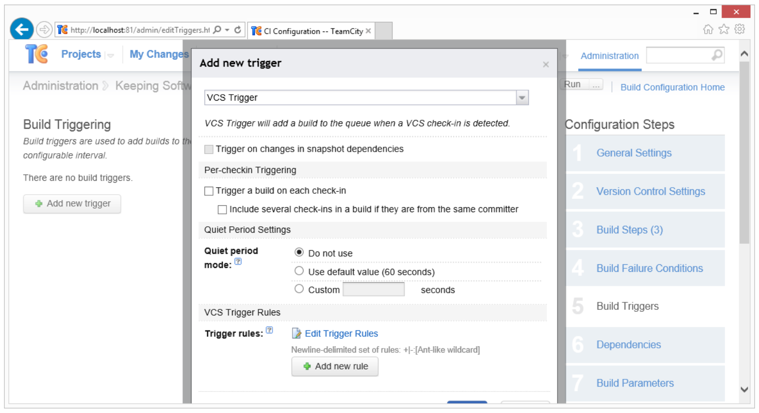
Total test count: 7; total duration: 10s

Status	Test	Duration	Order#
OK	SomePretendIntegrationTests. Test4 (Calculator.IntegrationTests.dll:Calculator.IntegrationTests)	5s,054ms	6
OK	SomePretendIntegrationTests. Test5 (Calculator.IntegrationTests.dll:Calculator.IntegrationTests)	5s,003ms	7
OK	SomePretendUnitTests. Test1 (Calculator.UnitTests.dll:Calculator.UnitTests)	38ms	1
OK	SomePretendUnitTests. Test2 (Calculator.UnitTests.dll:Calculator.UnitTests)	1ms	2
OK	SomePretendUnitTests. Test3 (Calculator.UnitTests.dll:Calculator.UnitTests)	1ms	3
OK	SomePretendUnitTests. Test4 (Calculator.UnitTests.dll:Calculator.UnitTests)	1ms	4
OK	SomePretendUnitTests. Test5 (Calculator.UnitTests.dll:Calculator.UnitTests)	< 1ms	5

CI Smoke Tests Running

Triggering the Whole Pipeline

The final step in configuring the build pipeline is to trigger it whenever new code is committed to source control. This is done by adding a “VCS Trigger” to the CI build:


Add new trigger

VCS Trigger

VCS Trigger will add a build to the queue when a VCS check-in is detected.

Trigger on changes in snapshot dependencies

Per-checkin Triggering

Trigger a build on each check-in

Include several check-ins in a build if they are from the same committer

Quiet Period Settings

Quiet period mode: [✓ Do not use] Use default value (60 seconds) Custom seconds

VCS Trigger Rules

Trigger rules: Edit Trigger Rules

Newline-delimited set of rules: +|-:[Ant-like wildcard]

http://localhost:81/admin/editTriggers.ht
CI Configuration -- TeamCity
Projects
My Changes
Administration
Administration > Keeping Softw
Run
Build Configuration Home
Build Triggering
Build triggers are used to add builds to th configurable interval.
There are no build triggers.
Add new trigger
Add new rule
Configuration Steps
1 General Settings
2 Version Control Settings
3 Build Steps (3)
4 Build Failure Conditions
5 Build Triggers
6 Dependencies
7 Build Parameters

Triggering the Pipeline on Code Commits

There are a number of advanced options that can be configured to fine tune the trigger.

Alternative CI/Build Servers

In addition to TeamCity there are other build servers in common use including:

- CruiseControl[33]
- Team Foundation Build[34] and cloud version[35]
- Jenkins[36]

[33]http://cruisecontrol.sourceforge.net/
[34]http://msdn.microsoft.com/library/ms181715.aspx
[35]http://tfs.visualstudio.com/
[36]http://jenkins-ci.org/

A Holistic View of Testing

The quality of software is increased by testing and fixing the defects that are found. This chapter is aimed at functional testing as opposed to performance or security testing.

Types of Testing

Testing can be roughly divided into to two types: human, and automated.

Human Testing

Human testing is the exercising of software by humans (theoretically this includes the end-user) to find defects. Human testing may involve manually executed pre-written test cases (that reduce the human component to robot-like action) or Exploratory Testing where the tester uses their experience, creativity and prior learnings to generate new test cases.

Human testing may not be 100% repeatable, errors can be made that mean a defect is missed or false defects are raised. By the same token, good human testers can think of strange and extraordinary ways that end users may operate the system that may not have been accounted for in the design and development of the software.

Human testing is usually limited to exercising the user interface, though ad-hoc human testing may occur by developers.

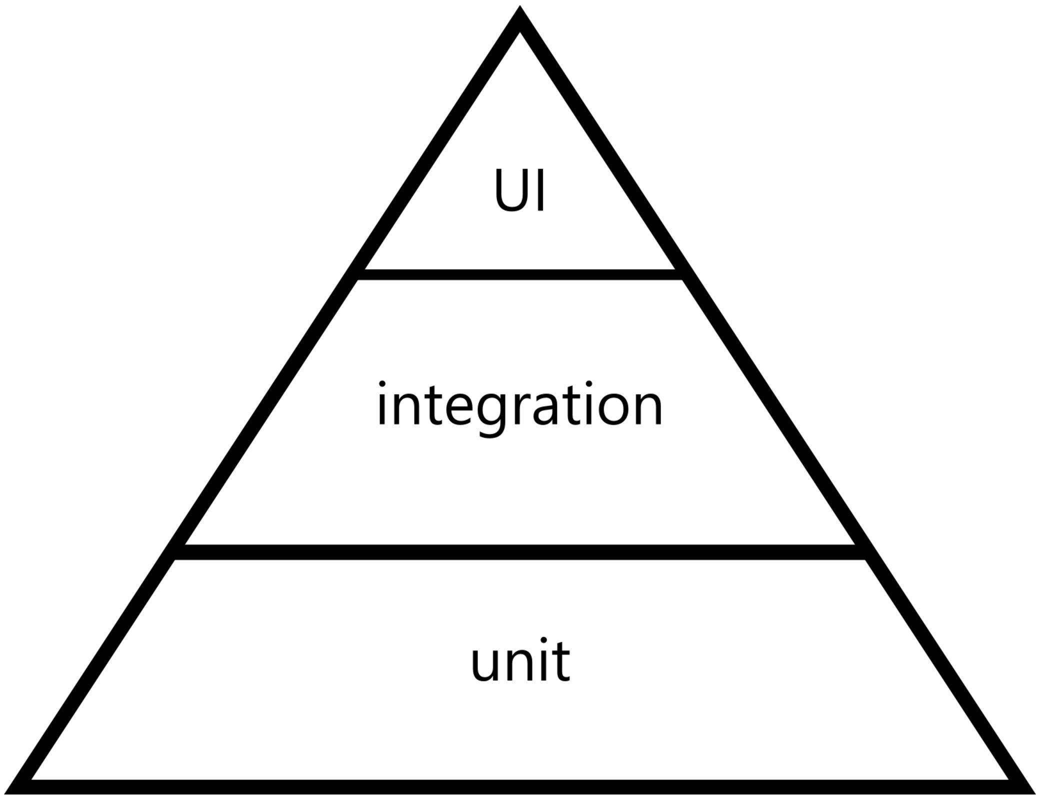
Automated Testing

Automated tests are executed by the computer and the results verified by the computer. Good automated tests are 100% repeatable and reliable and can be executed more quickly that human testers. Once they are created, other than maintenance costs, they are "cost-free" and can be run as often as required.

Automated tests are usually created and maintained by developers or "automation testers".

The Testing Pyramid

Testing can be performed at different levels of granularity. The testing pyramid diagram is typically used to illustrate the number of tests that occur at the different levels. There are many variations on this theme, the typical pyramid is shown below.



Basic Testing Pyramid

Unit Testing

Unit testing involves the execution of small "units" of code (for example a class) in isolation and verifying that it behaves as expected. They are the fastest types of test to execute and can more easily cover many permutations of test data inputs.

Unit tests are typically created by developers either after implementation code is written, or when practicing Test Driven Development before the code is written.

To isolate the class/code being tested, any dependencies are faked, usually using a faking/mocking framework.

Integration Testing

This level of testing can be further subdivided into component integration, external system integration, and service layer testing.

Component Integration Testing

Component integration testing is concerned with taking "components" such as classes (or sets of classes) and using them together (as opposed to using them in isolation with fakes) to ensure they interact correctly.

External System Integration Testing

These tests deal with verifying that the written code communicates correctly with external dependencies such as databases, third party APIs/components/services, etc.

Service Layer Testing

If the architecture of the system being developed exposes services, these services can be exercised as part of automated testing. Service layer testing is usually the last level before moving up to the UI testing level.

User Interface (UI) Testing

UI testing operates at the level of the user and interacts with the user interface of the application. UI testing typically has the slowest execution speed of all testing because it has to wait for the UI to refresh before performing each action. See the chapter on Automated Functional UI Tests for Web Applications for further detail on UI testing.

Automated UI testing usually requires some library or framework to allow test code to "drive" the user interface and "read" information from it.

What and Where to Test

The testing pyramid is a good guide to how many tests should ideally be at each level of granularity.

Generally speaking, the higher up the pyramid, the more able the test is to exercise a complete vertical slice of the application. For example a UI test to create a new customer may end up going via a service layer, some business logic, data access layer/ORM, and database/stored procedures. However, tests at the UI level will not typically exercise all the possible code branches.

At the opposite end of the scale, to test an algorithm, unit tests are more appropriate. At this level the unit tests can supply many different variations of input and assert the outputs. To perform this kind of testing through the UI would be extremely slow to execute.

An Example

As an example, imagine a typical Internet banking application. The table below lists some features/functions and the most likely types of testing that may best fit them (P = Primary testing, S = Secondary Testing).

Feature/Function	**Unit**	**Integration**	**UI**
Check digit validation	P		S
Date validation	P		S
Funds transfer		P	S
Statment PDF			P
Login		S	P
Order a replacement card		P	S
View account balances		P	S
Pay bill		P	S
Change password		S	P

Whilst it is impossible to generalise as every project team, codebase, and business direction is different; the above table shows that the same feature/function can be tested at multiple levels. Though some functions such as validation may be more appropriate at lower levels, they may also have a UI test to ensure that an invalid condition produces an error message and does not allow the user to proceed.

Testing Legacy Applications

When working with legacy applications it may be very difficult to get close to the typical testing pyramid. If the business/management do not want to spend effort on improving the codebase then this can limit the amount of testing that can be performed at the lower levels. If the code is essentially not soft, with tightly coupled code and low cohesion then unit testing may be difficult or impossible. In situations such as this there is little option* but to perform more testing at the higher levels. In doing this there are trade-offs, tests will generally be slower to execute and more brittle (increasing maintenance overheads).

* There are commercial tools such as Telerik's JustMock[37] that allow things to be tested in isolation even with poorly designed code.

[37]http://www.telerik.com/

Part 3: People and Process

People create code.

People follow a process when writing code - even if it is one of chaos.

Part three of this book discusses aspects relating to people, such as Software Craftsmanship and what makes programmers happy. Part three also looks at some processes related to software creation such as agile software development and pair programming.

At first glace some of these topics may not appear to be related to keeping software soft, but (for example) unhappy programmers may be more likely to write bad code and poor project management or time-management can create undue schedule pressure which leads to a temptation to write bad code.

An Introduction to Agile Software Development

Agile software development is a process that suggests a more lightweight approach to delivering software in a more incremental way. It places a high emphasis on people and collaboration.

Overview

The “agile software development methodology” has been around for at least ten years. The *Manifesto for Agile Software Development* (see Appendix A) was created when a group of seventeen software developers met in February of 2001 as a reaction to so-called heavyweight software development processes. Even before the Manifesto, some of the ideas of agile were already in existence in places such as the Unified Process that advocated iterative software development.

The Manifesto outlines a series of *values*:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

While the emphasis is on the items in bold, the other items (on the right) are also important.

There are also a number of principles behind the Agile Manifesto (see Appendix B).

Agile Software Development in Practice

While the agile software development process (agile from this point forward) is not a prescriptive process, a number of common *tools* tend to be used to help uphold the Manifesto values. While these tools are useful, if an agile team gets no value from them then they do not have to be continued, though some things like iterative development are usually seen as key.

Co-Location and Role-Independence

The people in an agile team are preferably physically co-located in a common area, room, table, etc. The team is usually made up from a group of different roles (a cross-functional team) all sitting and working together. For example a team could consist of programmers, business analysts, testers, project managers, and business representatives. While specialist roles still exist, the best agile teams are ones where the role distinctions fall into the background. This can manifest itself by testers sitting with programmers to review unit tests, business representatives sitting with developers to tweak user interfaces in real-time, testers and business analysts creating acceptance criteria together, and the business analysts executing manual testing.

Work Backlog

The backlog is a list of work that needs to be done. It can include any type of items that the team decides upon, but often includes "user stories" or features that the business wants to implement. It can also include bugs, organisational, and technical tasks.

The team can have as many backlogs as required, and there are typically at least two: the main backlog and the iteration backlog.

The main backlog (product backlog, feature backlog) is where some or all of the possible things the team needs to do are listed.

The iteration backlog is a list of the things the team is working on in the current iteration.

Backlogs are usually kept in a prioritised state so that the most important things are always at the top and are well understood by the team.

Iterative Development

An "iteration" is a recurring period of time where the team performs work. An iteration is generally somewhere between one week to one month, but can be whatever length the team deems to best suit the work they are doing.

At the start of every iteration, the team (including the business representative) choose the things they think they can complete during the iteration taking into consideration holidays, sickness, and other constraints.

Items are usually prioritised by highest importance/value first and these are started at the beginning of the iteration.

Daily Stand-Up

The daily stand-up is a short meeting (usually no more than fifteen minutes) where the whole team usually stands around in a circle and talks about the things they are working on. The idea being, by standing up, people will quickly become uncomfortable and naturally keep the meeting short.

The simplest form of update from a team member is to tell the rest of the team:

- What they did yesterday
- What they plan to do today
- Any “blockers” preventing them from proceeding

This is by no means a prescriptive format, teams usually tailor their stand-ups over time.

During a stand-up, things are generally not discussed in great depth and team members will usually congregate afterwards if there are things that need further detailed discussion.

Experienced agile team members know not to wait for the the daily stand-up to discuss important things. At any point in time a person can call a stand-up when the issue affects the whole team.

Another common theme for the stand-up is for the team to “focus on the backlog” or to use it as a “daily planning” meeting.

Definition of Done

The team decides what it means for an item of work to be called done. For some teams this can mean that it is in production and being used, for other teams this can mean it has been tested in a local environment but not yet in a production-like environment.

Team Retrospective

A team retrospective usually happens at the and of an iteration. It is a meeting where the team talks about what went well in the last iteration. It is also a time to discuss what went badly. The key output from a retrospective (“retro”) is a few items that the team agrees to improve on in the next iteration.

Retrospectives can also occur at other times, they could be held at the end of a project, release, quarter, year, etc.

Usually someone from the team will facilitate/guide the retrospective for the team, but sometimes an external facilitator can help to uncover difficult items that the team might tend to shy away from. An external facilitator can also bring a welcome change of energy.

Cardwall

A cardwall is a visual representation of the items of work that the team is working on. It can be created on a wall, whiteboard, window or any other suitable surface. The wall contains items of work represented by sticky notes, index cards, or pieces of paper.

Work typically flows through a series of columns or rows that represent different stages of the software development process. The simplest would be three stages: **to do**, **doing**, **done**. Usually teams evolve from this basic idea to include more stages as required. The stages are not set in stone, they usually evolve over time as the team itself evolves.

There are electronic tools to track items of work and create digital cardwalls, but physical ones have a number of benefits:

- Visible all the time
- A physical sense of progress (as items are physically moved)
- Viewable by anyone (great for starting conversations)
- Easily reconfigurable

An electronic tool can be used (in conjunction with a physical wall) to capture the whole backlog, but it is only when items are “in play” as part of an iteration that a physical card is created. The electronic tool can be a dedicated “agile” tool or a simple spreadsheet.

Electronic tools have a number of advantages:

- Reporting/statistics
- Require no physical space
- Security
- Audit
- Usable by physically separated team members

Physical cardwalls however help create a focus point around which the team can hold stand-ups and other conversions.

Some teams create “avatars” for the people in the team, these avatars can be simple stickers with the person’s name, photos of the person they represent, or even cartoon figures. These avatars are pinned/stuck to cards to represent what people are currently working on.

Items of work can also be decorated with coloured stickers or sticky notes to represent additional process metadata where a separate row/column might be overkill. This metadata can be anything: testing criteria reviewed, legal approval received, design proofs reviewed, automated functional UI tests created, etc.

A cardwall should be as simple and usable as possible and should not create a huge process overhead.

Different colour cards or sticky notes can be used to represent different types of work; for example, defects could be written on red cards, business features on yellow cards, and technical tasks on blue cards.

For inspiration on agile cardwalls, http://agileboardhacks.com/[38] is an interesting resource.

[38]http://agileboardhacks.com/

Automated Testing

The sooner defects are found the better. Automated tests (unit, integration, and functional UI) that run as often as required provide the team with quick feedback. They allow things to be changed with a greater degree of confidence that nothing else has broken as a side effect.

User Stories

A user story represents some feature that is required to be implemented. It usually represents some value to the business representative or end user. A user story can be as vague or as detailed as the team deems necessary.

User Stories can contain anything that is useful to the team. Some common things that are sometimes included are:

- Brief description
- Long description
- Flow chart
- Screen/design mock-ups
- Testing notes
- Acceptance Criteria (what it must *do* to be considered done)

It is usually better to aim for the least amount of detail as possible and have good team collaboration to “fill in the blanks”, though there may be audit or other compliance requirements that the team may have to abide by.

When creating user stories it is usually desirable to have a greater number of smaller stories than a few massive ones. This allows pieces of work to be continually flowing and making it to “done”.

Product Showcase/Demo

At the end of each iteration, the team demonstrates the working software to people outside the team. This can include stakeholders, other agile teams, end users, or whoever may be able to offer feedback about the software that the team is building.

Sustainable Pace

The team should be able to operate at a continuous pace over an indefinite period of time. Agile teams where people are feeling constantly stressed, tired, and unhappy are failing. While there may be short periods of overly intense activity, this should not be the norm.

Sustainable pace means that once a team is up to speed they should be getting (roughly) the same amount accomplished each iteration and be able to sustain that level of output for as long as is required.

Overarching Principles

Agile software development could be summarised as:

- Get the right people working and sitting together
- Include the customer/business representative as part of the team
- Deliver the software in small iterative chunks
- Shorten feedback loops as much as possible
- Focus on the software being built, rather than the process
- Teams continuously improve and refine over time
- Be open and able to deal with changes in requirements
- Sustainability over time

It is worth spending some time reading through the *Manifesto for Agile Software Development* (see Appendix A) and the Principles behind the Agile Manifesto (see Appendix B).

Pair Programming and Code Reviews

Having another person (usually a fellow programmer) look at code helps to find defects. While unit, integration, and functional testing helps to find defects, so too can human code reviews. Having multiple people look at code can help find defects that may not otherwise have been found through other methods.

The reviewing of code can find non-logical code related defects such as:

- Problems with incorrect or missing requirements
- Design and architectural mistakes
- Violations of coding/stylistic standards
- Overly complex code
- Code duplication of a function that appears elsewhere

There can also be a number of other benefits over and above finding defects:

- Domain knowledge transfer
- Language knowledge transfer
- IDE shortcut and tip sharing
- Team/project resilience to unexpected absence
- Increased awareness of system architecture/functions
- Cross-pollination of ideas

The reviewing of another developer's code should be an open and honest conversation with the sole purpose of improving the code that will eventually go into production. When reviewing code, there are a number of human factors that are required in order to achieve the best results:

- Honesty
- Clear communication
- Open-mindedness
- Courage
- Inquisitiveness
- Low ego
- Patience
- Respect

When conducting a review of another person's code, the main focus should be on finding logical errors, with less of an emphasis on code formatting/style. The reviewing of code can be used with any style of software development process, be it agile or otherwise.

Generally speaking there are two ways to review code: a "formal" **code review**, and **pair programming**.

Formal Code Reviews

A formal code review occurs when a developer reviews another programmer's code after it is deemed complete by the originating programmer. Once it has been reviewed it is either "signed off" as suitable for production use or it is deemed unacceptable and requires some amendments.

In a healthy team/organisation, informal code "reviews" happen continuously.

Just as with pair programming (see below), code reviews can help find defects that other forms of testing may not. Code reviews should not be done just to satisfy some management/audit requirement. This just makes them superficial and helps to give code reviews a bad reputation. The outcomes of code reviews should also not be used for "witch hunts" or as a management performance-related metric.

In terms of actually performing a review, there are some companies that print code and attach a coversheet with tens of tick boxes representing code quality aspects. At the other end of the scale, some companies have no formal logging or review criteria. If code reviews are to be seen as useful and embraced by programmers, the level of process and friction should be as minimal as possible to gain the benefits of a review, while at the same time satisfying any audit requirements.

Even with the inherent benefits of reviewing code, formal code reviews have a number of drawbacks. Because the code has already been written (perhaps a day or weeks worth of effort), if there are any problems that need fixing, the programmer has to go back and fix them before another review happens. Depending how often reviews happen, the programmer may have already started a new piece of work, this means having to context-switch back to the previous task which incurs a cost overhead. Also because the programmer has to spend time explaining to the reviewer how/what the code does, the review needs to spend additional mental energy understanding both the requirement and the implementation.

Even with these drawbacks, introducing code reviews to a team can still increase the quality of the code.

Pair Programming

Pair programming ("pairing") is where two programmers share a single computer and write code collaboratively. It is described by some as "real-time code reviewing".

Pairing has all the benefits of formal code reviews, but the "review" happens constantly and at the point the code is written, thus the feedback cycle is much shorter.

Typically one person is the "driver" and the other person the "navigator" (sometimes called the "observer"). The driver is the the person actively writing the code, while the navigator is on the look-out for mistakes and is considering where the code is going at a higher level.

The two programmers switch roles often so that both stay engaged and both have a chance to write code. If Test Driven Development is being practiced while pairing, then the switch can happen after each test is written. This way programmer A gets to write a failing test, programmer B makes the test pass (and refactors), then programmer B writes the next failing test and programmer A makes it pass. Pairing can also work when combined with fixed time-boxes and with techniques such as the Pomodoro Technique.

In addition to the regular switching of roles, the regular switching of the people in the pairs can also take place. This helps to spread knowledge throughout the team and also helps to ensure a given pair does not become "stale".

The interval that both role and pair switching occur can be whatever best suits the programmers and team. Sometimes a timer is used, whenever it rings then the swap happens. Pairs could swap every hour, half day, or at the end of a logical (small) piece of work.

In terms of how much pairing to undertake, some suggest 100% of code should be paired on. The right level will depend on the project, the team, and the management. Pairing can be tiring so both participants should be mindful of their own and their partner's energy levels, especially if sustainable pace is a goal.

Benefits of Pair Programming

The conclusions made by researchers on the benefits of pairing are varied.

One study Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise[39] [Arisholm, E, Gallis, H, Dyba, T & Sjoberg, D. (2007) *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 2,*] shows an overall reduction in **duration** of 8%, an overall increase in **correctness** of 7%, with an overall increase in **effort** of 84% when compared to individual programmers working alone. However, when the results are broken down by complexity and programmer experience, a greater range of results emerge. For a complex system the findings are: 6% increase in **duration**, 112% increase in **effort**, and and increase in **correctness** of 48%. For a complex system with junior programmers pairing the results for **correctness** are much larger, with a 149% increase. The study concludes: "the results show that the effects of PP [pair programming] depend on a combination of system complexity and the expertise of the subjects".

In *The Costs and Benefits of Pair Programming*[40] [Cockburn, A & Williams, L] the authors make reference to a previous study [Williams, L., et al., *Strengthening the Case for Pair-Programming, in*

[39]http://simula.no/research/se/publications/Arisholm.2006.2/simula_pdf_file

[40]http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF

IEEE Software. submitted to IEEE Software] that found pairing increased the time spent by about 15% and the resulting code had about 15% fewer defects. The authors point out that “the initial 15% increase in code development expense is recovered in the reduction in defects”. The paper goes on to list some other important factors that result from pairing, including: increased programmer satisfaction (“most of the programmers enjoyed programming collaboratively”); better design as evidenced by fewer lines of code; increase in problem solving speed; team building; and knowledge sharing.

As Martin Fowler points out, productivity is hard to measure: “this is somewhere I think we have to admit to our ignorance” - CannotMeasureProductivity[41]. Some generalisations can still be made as to when pair programming may be of most benefit; namely when:

- The task is not well-understood at the start
- The task is complex/difficult (especially when apprentice programmers are undertaking it)
- The task carries a high level of risk
- The task requires a high level of creativity
- The task is architecturally significant
- Management requires resilience to programmer absence
- Management requires cross-skilling

Anecdotally, even when programmers are not sure if they *like* pairing, they usually agree that the end solution is better than if it was created alone.

Pairing and People

People pair. Every programmer has different skill levels, experience, and outlooks. Initially some programmers find pairing difficult - programmers typically tend to the more introverted end of the introvert-extrovert scale. One commentator on an article[42] from Coding Horror sums this up: “I picked an office job in programming in part because it lets me spend most of my day in silence and concentration. Having to work that closely with someone else all day would certainly put a dent in job attractivity [sic] for me”. This is by no means a blanket statement about all programmers, rather it highlights that pairing for some people may be an unpleasant experience at first.

Pairing Tips

When getting started with pairing it can be worth printing out a list of good practices and sticking it to the monitor(s) that will be used for paring. The following list outlines some things that can help pairing be more effective and enjoyable:

[41]http://martinfowler.com/bliki/CannotMeasureProductivity.html
[42]http://www.codinghorror.com/blog/2007/11/pair-programming-vs-code-reviews.html

- Writing of code should be shared equally (try using a timer)
- Use two keyboards and mice, rather than sharing a single one
- Use a desk with space for two people to sit comfortably
- Take regular breaks to keep energy levels higher
- Rotate paring partners regularly
- Be open to learning new things
- Communicate: explain, listen, and ask questions
- Wait ten seconds before calling out a typo
- Use Test Driven Development in conjunction with pairing
- Chew gum, use breath mints
- Write some code (even if it is really bad) to start a conversation
- Be respectful
- Have fun

Time Management and Motivation

It is more likely that when people are tired and lacking in energy and/or motivation that they will create bad code. Programming is a purely mental activity, the physical part (banging on a keyboard) is merely the mechanism where these mental constructs are made manifest in source code.

This chapter explores some topics that can aid in maintaining energy and motivation and thus result in higher quality, softer code.

Time Management

There is a finite amount of time available if sustainable pace is to be maintained. This means that some tasks will not be completed. This implies that the most important and valuable tasks should be worked on first.

There are a number of techniques to help with deciding what are the most important things, two of these techniques are MoSCoW and the Covey Quadrants.

Prioritisation With MoSCoW

The MoSCoW method (developed by Dai Clegg of Oracle UK Consulting) is a technique where things are placed into one of four categories (the o's do not have a meaning):

- **M**ust have
- **S**hould have
- **C**ould have
- **W**on't have

Before jumping into MoSCoW, the current objectives should be well understood by everyone participating in the conversation. It can be worth stating the objects explicitly before starting. If using MoSCoW to prioritise personal tasks (as opposed to team tasks) it can sometimes be worth writing down what the current personal objectives are. Having the correct context before starting helps when prioritising individual items.

Must Have

Items that are placed in this category are the highest priority. They constitute features or other work that is of highest value to the customer, team, business, etc.

It is usually the case that if even a single must have item is not delivered then the entire endeavour is regarded as an overall failure.

Should Have

Items in this category are usually considered of equally high importance as must have items. The reason they are considered lower priority may be because:

- They are not as time critical (e.g. can go into a future release)
- There is a workaround that can be used in the short-term (e.g. a bug than can be bypassed)
- There are external dependencies/systems that are not currently available, so work cannot start yet

Could Have

Items in the could have category can be described as “nice to have”. They are not critical to success and are not as high value as the categories above. Could have items (e.g. “right align the labels so the UI looks better”) can still offer improvements to user/customer satisfaction and can be included if there is enough time to work on them.

Won’t Have

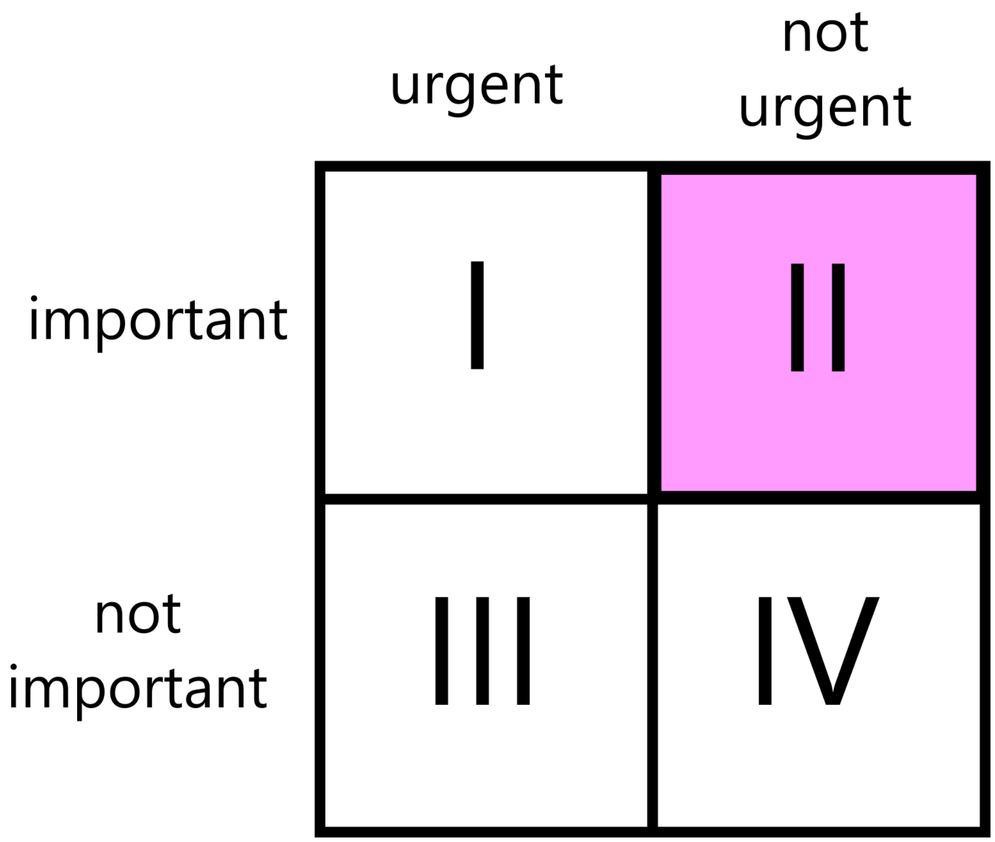
These items won’t currently be included, though at some point in the future they may become a higher priority. At some point they may also be completely removed from the list.

Sometimes “won’t have” is labelled “would have” or “would like”, i.e. if there was enough time and money available they would be completed.

Prioritisation with Covey Quadrants

The Four Quadrants or Time Management Matrix (popularised by Dr. Stephen Covey) is a way of splitting up tasks into four categories (quadrants) based on their *urgency* and *importance*.

The diagram below shows the four quadrants:



The Covey Quadrants

Given a list of items, each item can be classified into one of the four quadrants. The shaded quadrant II is regarded as containing the most valuable tasks in the long term, a greater focus should be placed on items in this quadrant.

Quadrant I - Urgent and Important

Items in this quadrant can be termed “fire-fighting” or “emergencies”. They are things that have to be done now, but may not contribute to long term success. This quadrant also includes things that have unmovable deadlines.

Quadrant II - Not Urgent but Important

These items represent long-term benefit and growth. They are often the most valuable and more time should be spent on items in this category than the other quadrants. For example, physical exercise is not urgent but is of great importance for long-term health.

Quadrant III - Urgent but Not Important

Items in this quadrant represent interruptions that could otherwise be avoided such as unimportant reports or emails, or answering a phone call that turns out to be unimportant. Items in this quadrant should be reduced or eliminated if possible.

Quadrant IV - Not Urgent and Not Important

These items represent trivial, time-wasting things such as manually deleting spam email or mindlessly browsing the web. Items in this quadrant should be eliminated if possible.

For more information see "The Seven Habits of Highly Effective People" and other publications by Dr. Stephen Covey.

Maintaining Energy with the Pomodoro Technique

The Pomodoro Technique is a simple time management system developed by Francesco Cirillo.

A "Pomodoro" is an indivisible, all-or-nothing unit of time that is exactly 25 minutes long. A Pomodoro is ether completed or not, for example there is no concept of a completing a half-Pomodoro.

The basics of the technique are:

- Work for 25 minutes (one Pomodoro)
- Record the completion of a Pomodoro
- Take a 5 minute rest break
- Repeat
- After every 4th Pomodoro, take a longer rest break of between 15-30 minutes

The splitting-up of time into 25 minute chunks helps to focus the mind on what small tasks(s) can be accomplished next. If a Pomodoro is horribly interrupted to the point where it is considered void, then a short break can be taken and a new one started. The recording of every completed Pomodoro contributes to the feeling of a sense of progress and can further enhance motivation and energy.

During the 5 minute rest break the mind should be allowed to rest; the current task should not be thought about but rather: go for walk, get a drink, stretch, gaze out of the window, etc.

On top of this basic idea, the technique also suggests recording the number of interruptions that occur during a Pomodoro.

Interruptions are classified as either:

- **Internal**: things that pop into a person's head that are unrelated to the current task, e.g. "what shall I have for lunch today"

- **External**: things that occur from outside influences such as phone calls, instant message notifications, etc.

By recording the types and number of interruptions, the data can be used to reduce or eliminate them over time. This can involve turning off telephones, email, and instant messaging during a Pomodoro and instead having dedicated Pomodoro(s) to responding to emails.

The technique, when applied correctly can result in less tiredness while programming and an overall increase in productivity. There are some programmers who do not like the forced "interruption" of a rest break every 25 minutes and feel that it interrupts their concentration. It is a technique that should be trialled by each individual for a reasonable period to see if it can be of benefit.

There are many electronic Pomodo timers available for a whole host of platforms to help time the 25 minutes.

Pomodoro Technique and Pomodoro are registered trademarks of Francesco Cirillo. For more details about the technique, see the official website[43].

Maintaining Motivation with the Power of Three

The idea of "the power of three" is a universal concept; it can be seen anywhere from religion (The Holy Trinity, Triquetra, etc.) to childhood learning ("ABC", "123") to entertainment (The Three Musketeers, The Three Stooges, etc.) to project management (the scope-cost-schedule triangle).

One simple way to exploit the power of three is to define outcomes/wins/achievements. These "three wins" can be at any level:

- Three wins for the project
- Three wins for the year
- Three wins for the month
- Three wins for the week
- Three wins for today

The three chosen things should not be simple tasks to be ticked off, they should be things that feel like a "win" and give a real sense of progress and satisfaction. The "wins" should be realistic and achievable.

Getting Results the Agile Way[44] by J. D. Meier makes extensive use of the power of three with its "Monday Vision, Daily Outcomes, Friday Reflection" and three types of "Hot Spot".

The power of three can also be applied to individual items of work/features being developed, for example a feature can be evaluated by: user value, business value, technical value. In agile team retrospectives the power of three can be used to decide: what went well, what didn't go well, what will be improved upon in the next iteration.

[43]http://www.pomodorotechnique.com/
[44]http://gettingresults.com

Technical Debt

Technical Debt is a metaphor that can be used to help explain the effects of low quality software to those who may not understand software development.

The basic premise of Technical Debt is that when it is accrued, interest must be paid on it over time. The cost of paying this interest reduces the amount of resources available that could otherwise be spent adding new features and business value.

At its worst, when there is too much Technical Debt and all effort is being spent on paying the interest, a system may have to be re-written before new features can start to be added again.

Most software contains a greater or lesser amount of Technical Debt.

Examples of Technical Debt

There are many different examples of what may be classed as Technical Debt. The list below is by no means exhaustive but it offers some examples:

- Tight coupling between internal and external dependencies
- Poor test coverage
- Poor quality tests
- Inconsistent code formatting styles
- Hardcoded values ("magic strings")
- Complex or bloated methods, functions, classes, etc.
- Code duplication
- Redundant or unused code
- Out of date or unsupported third party libraries
- Poor code commenting
- Missing or out-of-date documentation
- Shared global variables, statics, etc.

While Technical Debt is typically associated with code quality, it can also apply to other areas such as documentation and infrastructure.

Causes of Technical Debt

Possibly the single biggest cause of the accrual of Technical Debt is delivery schedule pressure. This pressure causes decisions to be made that create Technical Debt. Schedule pressure is by no means the only cause though, other contributors can include:

- Unmanaged, incorrect, or chaotically changing requirements
- Junior, inexperienced, or unmotivated programmers
- Parallel development streams followed by big-bang merges
- Delayed refactoring or no refactoring
- Poor communication and collaboration
- Poor or inexperienced management

Regardless of the cause(s) of the accrual of Technical Debt, an important and useful distinction is that of: was it was accrued with **mindfulness**?.

Mindfulness of Creating Technical Debt

If it is accepted that all software will contain Technical Debt, then it should be accrued with **mindfulness**.

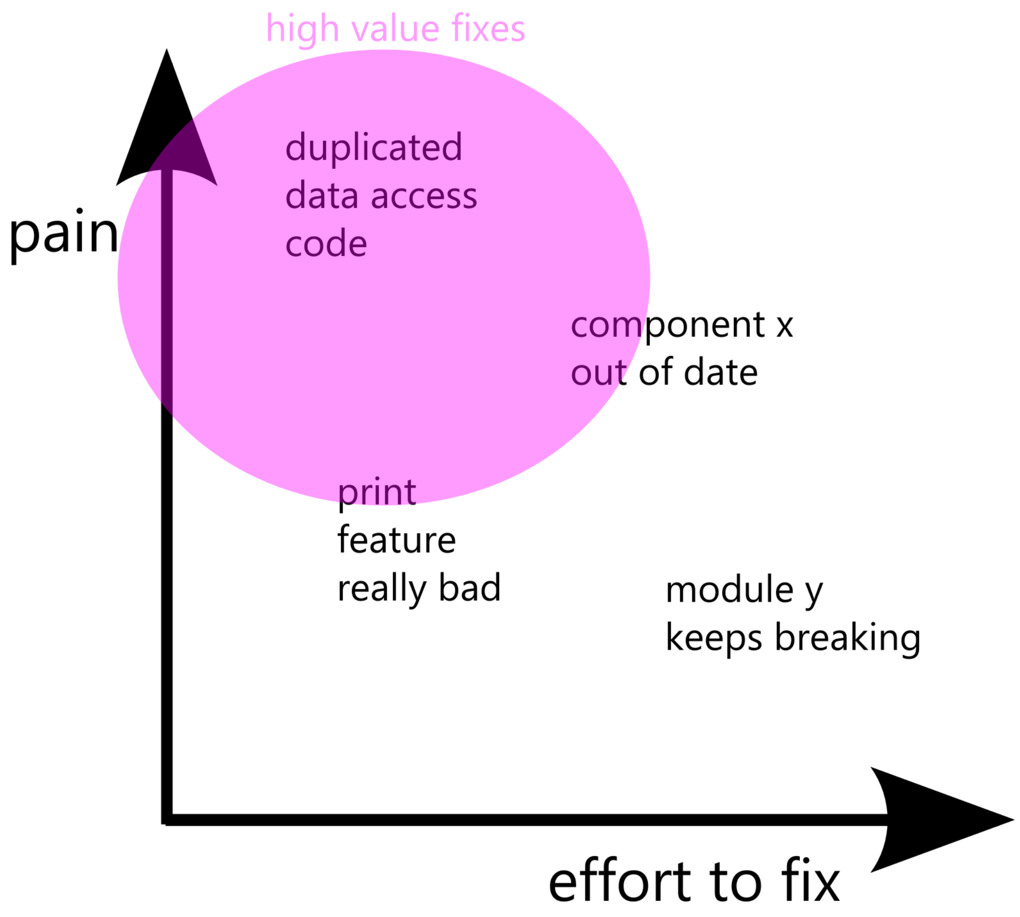
Mindful accrual means not only being fully aware of the fact that Technical Debt is being accrued, but also doing it in a measured, thought-out way. Mindful accrual also means that the team and customer are fully aware of the possible future consequences. Mindful accrual can be typified by: "*we need to ship on this date or we'll lose a lot of valuable business, but we've allocated some time for refactoring in the next release*".

Martin Fowler [TechnicalDebtQuadrant[45]] creates a useful conceptual model by using the terms "reckless", "prudent", "deliberate", and "inadvertent" and arranging them into quadrants.

Recording and Prioritising Technical Debt

Assuming that Technical Debt is being accrued with mindfulness, then it is useful to use some method to track the debt. One nice way to do this is to reserve a section of a wall and create a Technical Debt "chart". This chart has two axes: **pain** and **effort to fix**. Items of Technical Debt can be placed on the chart, the highest value items occur when the Technical Debt causes a lot of pain but is the easiest to fix.

[45] http://martinfowler.com/bliki/TechnicalDebtQuadrant.html



Technical Debt Chart

Having Technical Debt visible in this way can also help to make people aware it and promote conversations that might not have otherwise taken place.

The Effect of Technical Debt on People and Organisations

Technical Debt slows down the ability to change the software more quickly. This slower delivery may hurt the organisation in terms of profits but there can be subtler consequences of rampant, uncontrolled Technical Debt:

- Developer happiness: having to work with “crappy” code all the time and with no time to make it better can have a negative effect over time on developer happiness and thus productivity
- Staff retention and recruitment: An organisation may struggle to retain developers or recruit new ones if the pain of technical debt is ever-present and is never addressed by the organisation

Software Craftsmanship, Professionalism, and Personal Development

This chapter presents a number of concepts that focus on the people who write the code. By advancing the skills and outlook of individual software developers, the industry (craft) as a whole will benefit and hopefully result in softer software, happier programmers, and more satisfied customers.

Software Craftsmanship

While the author uses the conventional term “craftsman(ship)” in this chapter, the term is seen by some as too gendered and that something like “software crafter” would be more inclusive.

The Software Craftsmanship “movement”, at its core is about making it ok for programmers to talk about programming as an equally important part of the software development process. Some critics argue that it attempts to elevate the programmer (and by extension the code) above that of all other things, including the customer, though this is not the intention as the forth item in the Manifesto demonstrates: “*Not only customer collaboration, but also* ***productive partnerships***”. It can be seen as an attempt to make programmers who may have felt disenfranchised by some parts of the agile movement that have focussed too much on the people and not enough on the technical, to once again have a voice.

Terms such as “craft” also do not sit well with some people, but putting aside the political debate and arguments over metaphor applicability, the author sees Software Craftsmanship as: a group of people who want to get better at making well-crafted software so as to better serve the customer.

The Manifesto for Software Craftsmanship see appendix E contains a number of points that outline the meaning of Software Craftsmanship, individuals may also sign the manifesto online[46].

Following on from the idea of “crafting” introduces the concept of the apprentice-journeyman-master model. In software development there is no “universal guild” to administer such a system, however it is a useful conceptual model; for example this book is aimed primarily at apprentice and journeyman programmers.

In software development, the three levels of “craftsman” could be described as:

[46]http://manifesto.softwarecraftsmanship.org/

- **Apprentice programmer**: a new graduate employed in a graduate trainee role or a "hobbyist"/"enthusiast" employed in their first programming job
- **Journeyman programmer**: several years of building software for customers and now starting to teach other journeymen and apprentice programmers
- **Master programmer**: many years of building software for customers and teaching other programmers, well-respected in the software development community, and through continued thought and practice continues to advance the software development profession as a whole. Current Masters would probably include the likes of Martin Fowler, Kent Beck, "Uncle" Bob Martin, *et al*

As an example of applying this metaphor, the author considers himself to be a Journeyman and by writing this book hopes to make further progress toward eventual mastery.

Beyond Craftsmanship - Into Professionalism

Unlike professions such as law or medicine, programming is sometimes not seen as a profession at all. Programming is a relatively new occupation so this is unsurprising, however there are moves to increase the actual and perceived level of professionalism in the industry.

It could be argued that the Software Craftsmanship movement should not be required as all programmers should act in an ethical and professional manner; this is what the term "beyond craftsmanship" is intended to embody.

The various computing societies around the world have their own codes of ethics and principles. For example the Australian Computing Society[47] Code of Ethics outlines the following values:

- The Primacy of the Public Interest
- The Enhancement of Quality of Life
- Honesty
- Competence
- Professional Development
- Professionalism

Regardless of whether an individual belongs to a society or not, the author would like to present some possible common traits of Software Professionalism:

- Attempt at all times to advance what Software Professionalism means
- Challenge opinions and ways of working in a respectful, compassionate manner
- Embrace diversity of opinion, gender, race, sexual orientation, religion, and experience
- Seek to continuously self-improve technical and non-technical ("soft") skills
- Seek to help others improve their skills
- Work with the customer to help them realise and achieve the best outcomes

[47] http://www.acs.org.au

Personal Development

Software Craftsmanship and Software Professionalism have a common root in that of personal development. With personal development neither of these two movements could succeed.

Software is a reflection of the organisational structure and ethos, and the people involved in its creation.

Personal development includes both the technical skills and “soft” skills.

With continued personal development (and the advances of hardware, languages, compilers, and other tools) the “softness” of software can continue to be improved. This is not to say however that every programmer should be expected to become a Master. There is enough pressure (both internal and external) on programmers without the addition of this expectation.

What Programmers Want and What Your Manager Should Understand

Happier and more motivated programmers are more likely to produce higher quality software. This chapter covers common things that programmers often cite as being important to them.

It is hoped that this chapter can help those programmers who feel like they are struggling to be happy at work and also as a guide for managers so that they may increase happiness and by doing so increase code quality and better serve the customer.

What Programmers Want Survey

The What Programmers Want[48] survey asked programmers to rate the things that are most important to them. The top five items from the results can be categorised as: "personal growth" and "peers and management".

Personal Growth

In the survey, "Room for Growth of Skills/Knowledge" and "Opportunity to Use/Learn New Technologies" feature 1st and 4th place respectively. Both of these items can be said to relate to a programmer's personal growth.

There are two probable aspects to why personal growth ranks so highly among programmers: fear and love.

Fear

The programming sphere changes quickly. Programmers working in highly restrictive, highly risk-adverse environments or those with inexperienced management or huge technical debt-laden legacy codebases may be restricted by: time pressure to be able to learn new things; and by policy to be able to implement new ideas. Other than working on personal projects outside of work, this means that a programmer's technical skills will eventually fall into decline. The awareness of this decline can give rise to fears including those of personal inadequacy and future employability (along with the associated financial fears).

[48]http://blog.stackoverflow.com/wp-content/uploads/WPW-Summary.pdf

Love

Many programmers love to learn. Whether it is a new technology or language, a new process such as agile software development, or soft skills such as presenting; the passion for and capacity to learn sometimes seems like an ingrained genetic characteristic of some programmers. This love of learning, of self improvement and the associated feeling of self-progression explains why personal growth may rate so highly.

Peers and Management

The other top two items in the survey top 4 are: “High Caliber Team” and “Positive Organization Structure”. These can be interpreted as: working **with** great fellow programmers and working **for** great managers.

Working with great peers probably includes aspects from personal growth - working with great programmers enables the learning of new things.

Having great management/managers also relates to the 5th top item on the survey “Lots of Control Over Your Own Work”, in that management that becomes overly involved in technical decision making or micromanagement results in a loss of this control.

Challenging Work, Creative Solutions, Making a Difference

As Rob Walling[49] points out: “Developers love a challenge. Without them we get bored, our minds wander…”, or more plainly by Andrew Wulf[50]: “Good programmers like challenges. Good programmers hate stupid”. While Andrew Wulf uses the qualitative term “good”, it is worth noting that there are many types of programmer and the best teams are made up of a mix of individuals.

Whatever the domain or specific problems that need solving, the sense of doing something worthwhile that makes a difference and that gives rise to a feeling of “progress” can be a great motivational and happiness factor. According to the Harvard Business Review Breakthrough Ideas for 2010[51] it is this “power of progress” that really motivates people: “Having just completed a multiyear study tracking the day-to-day activities, emotions, and motivation levels of hundreds of knowledge workers in a wide variety of settings, we now know what the top motivator of performance is… progress”.

This sense of progress (and thus personal motivation and happiness) can be different for different types of programmer, at a high level this sense of progress could include:

- Seeing users actually using new features/defect fixes

[49]http://www.softwarebyrob.com/2006/10/31/nine-things-developers-want-more-than-money/
[50]http://thecodist.com/article/what_programmers_want_is_less_stupid_and_more_programming
[51]http://hbr.org/2010/01/the-hbr-list-breakthrough-ideas-for-2010/ar/1

- Improving internal code quality (“softness”)
- Increasing test coverage/quality
- Preventing defects
- Process improvements
- Keeping up-to-date with framework versions
- Using new language features to simplify code

The best managers will help their programmers realise what a sense of progress means to them and allow them to work in areas that maximise it.

Tools and Environment

It is amazing how many programmers have had to battle with management for as simple a thing as multiple monitors. The best managers and organisations realise that programmers are professionals and know what tools will best enable them to be as productive as possible. The fact that these debates happen points to a fundamental chasm of misunderstanding between traditional management and the programmers they manage.

As the Joel Test (below) asks: “Do you use the best tools money can buy?”. The best management know that programmers are (generally speaking) easily frustrated by repeated friction that could be easily solved by better hardware or tools, an example of this is using an older/slow computer that results in slow local build times. For a programmer forced to use use outdated tools, it can feel like death by a thousand cuts which ultimately results in demotivation or in them leaving the organisation.

The Joel Test

Another useful list is the one created by Joel Spolsky in his article The Joel Test: 12 Steps to Better Code[52]. The test involves answering a series of twelve yes/no question such as “Do you use source control?” and “Do you use the best tools money can buy?” and simply adding up the yeses.

In addition to enabling programmers to rate current and future employers, the test can also be used as a conversation starter with management.

[52]http://www.joelonsoftware.com/articles/fog0000000043.html

Appendices

Appendix A: Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

Copyright 2001, the above authors

This declaration may be freely copied in any form, but only in its entirety through this notice.

http://agilemanifesto.org/[53]

[53]http://agilemanifesto.org/

Appendix B: Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity–the art of maximizing the amount of work not done–is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

http://agilemanifesto.org/principles.html[54]

[54]http://agilemanifesto.org/principles.html

Appendix C: Testing Framework Attributes

The following table lists some common attributes as used by the most common .Net testing frameworks.

xUnit.net	**NUnit**	**MSTest**
[Fact]	[Test]	[TestMethod]
not required	[TestFixture]	[TestClass]
Assert.Throws Record.Exception	[ExpectedException]	[ExpectedException]
Use test class constructor	[SetUp]	[TestInitialize]
IDisposable.Dispose	[TearDown]	[TestCleanup]
Fact(Skip="reason")]	[Ignore]	[Ignore]
[Fact(Timeout=ms)]	[Timeout]	[Timeout]

For more information see NUnit 2.6 vs MSTest 2010/2012 vs xUnit.net [55] and the xUnit site[56].

[55]http://bit.ly/comparetestframeworks

[56]http://xunit.codeplex.com/wikipage?title=Comparisons

Appendix D: Design Pattern List

The following lists common design patterns and organises them by creational, structural, and behavioural patterns.

Creational Patterns

- Abstract factory
- Builder
- Factory method
- Object Pool
- Prototype
- Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioural Patterns

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer

- State
- Strategy
- Template Method
- Visitor

Appendix E: Manifesto for Software Craftsmanship

Raising the bar.

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software, but also **well-crafted software**

Not only responding to change, but also **steadily adding value**

Not only individuals and interactions, but also a **community of professionals**

Not only customer collaboration, but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

© 2009, the undersigned.

this statement may be freely copied in any form, but only in its entirety through this notice.

http://manifesto.softwarecraftsmanship.org/[57]

[57]http://manifesto.softwarecraftsmanship.org/