Clean C#

Readable, Maintainable, Pleasurable

Jason Roberts

Clean C#

Readable, Maintainable, Pleasurable C#

Jason Roberts

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

This version was published on 2015-06-01

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To all those giants of Software Engineering who have gone before and on whose shoulders we all now stand.

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

Keeping Software Soft

A Practical Guide for Developers, Testers, and Managers

Learn how to make software easier to change.

Keeping Software Soft[4] (also available on Kindle)

---

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

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]

Pluralsight Courses

Browse Pluralsight courses by Jason Roberts[6]

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

About this Book

Welcome.

This book will help you (and the readers of your code) be happier and more productive by writing cleaner, more maintainable, more readable, and generally more pleasant C#.

I hope you enjoy reading and using the information in this book as much as I did writing it. May it improve your software development experience and overall happiness.

Best Wishes,

Jason Roberts

Introduction

What is Clean C#?

The concept of clean C# is that which is *easily understandable*.

It has been described as: "…simple and direct… like well-written prose." (Grady Booch); makes it "hard for bugs to hide" (Bjarne Stroustrup); and that it "looks like it was written by someone who cares" (Michael Feathers).

Clean code emphasises the human reader of the source code. Just because the compiler can easily understand the code, it does not mean the human reader can.

Why Clean C#?

If the code in our application currently executes correctly and satisfies the end-user, why does it matter if it is clean?

If the code will never again be read or modified there could be an argument for not worrying about it being clean. Even in this instance, writing it in a clean way to start with may make it easier and quicker to understand while writing it in the first (and only) version.

If we assume that the code currently being written will also be read (and probably changed) many times in the future, writing clean C# benefits the reader. It is important to note that the reader may even be the programmer who originally wrote the code. Coming back to code that was written in the past often requires the programmer to reacquaint themselves with the code, even if it was they who originally authored it.

If cleaner code is more easily readable - more understandable by the programmer reading it - this implies that it should also be easier to change, whether this is to add new features or find and fix defects. This also benefits the business and the end-user. The business is more able to quickly respond to changing market conditions

and outmanoeuvre their competition. Users may also get new features more quickly and annoying defects removed sooner.

The human brain has limitations on the number of chunks of information it can hold in working short term memory. Therefore writing clean C# code offers the opportunity to reduce the cognitive load on subsequent readers, and also the originating programmer. As an example, imagine an overly long method that makes use of 15 different local variables. As the reader is scrolling through the method, they have to try and retain these 15 variables in their short term memory while trying to figure out what the code does.

Happiness is important in all areas of life. In a work setting, happier individuals may be as much as 10-12% more productive[7]. Constantly working with dirty code may reduce team morale, reducing the level of happiness in the programmers (and by extension the wider team). Clean code can reduce this level of unhappiness, thus increasing productivity. This results in obvious benefits to the both the business and the end-user.

Using this Book

Code Samples

The code samples in this book are generally divided into "clean" and "dirty". When viewing a code sample, the namespace will usually indicate one of these, for example: `namespace CleanCSharp.Comments.Dirty`.

Order

The book may be read in any order but it may prove beneficial for the reader to read in sequential order as later chapters may assume previous chapters have been read.

Prescriptive

Whilst the techniques in this book will help to create cleaner C# code, some of the suggestions may not suit the readers preferences or sense of style. The techniques

[7] http://www2.warwick.ac.uk/fac/soc/economics/staff/dsgroi/papers/manuscriptandappendix.pdf

in this book will help to create cleaner C# code, though ultimately the team should decide what and how clean C# code will be implemented by all the developers in the team.

Clean C# is a Spectrum

Often, developers can exhibit a boolean mindset: black or white, awesome or rubbish, dead or new. Most of the time there is a range rather than just two absolutes.

It is tempting to also think of code as either being completely clean or completely dirty, when it is more accurate to think of clean code as a spectrum or scale of cleanliness. One poorly named variable in an otherwise beautiful solution does not mean the entire system is dirty.

Comments

Comments can be a highly useful form of clarifying why code is like it is. Often they are not.

Dirty comments adversely affect the readability of the source code.

Repeating What the Code Already Says

It should be assumed that the reader knows the programming language they are reading.

Take the following code:

```
namespace CleanCSharp.Comments.Dirty
{
        // This defines a class called BasicCalculator
        public class Calculator
        {
                // Define default constructor
                public Calculator()
                {
                }

                // Define a method to add two numbers
                public int AddTwoNumbers(int a, int b)
                {
                        // declare an int to hold result
                        int result;

                        // set result to sum of a and b
                        result = a + b;
```

```
                        // return the result to the caller
                        return result;
                }
        }
}
```

Notice here how painful it is here to read through this code. Not only does the human reader have to parse the actual lines of code, they also have to spend mental energy wading though the repetitive comments.

There is also a subtle inconsistency here too:

```
// This defines a class called BasicCalculator
public class Calculator
```

Notice that the comment is not only pointless but also misleading, it is stating an incorrect class name.

Comments can easily become out of sync with the code they describe. To keep them in sync also requires additional time for no additional benefit.

Repetitive comments should be deleted.

The clean version of the above code looks like the following (note the redundant default constructor declaration has also been removed:

```
namespace CleanCSharp.Comments.Clean
{
        public class Calculator
        {
                public int AddTwoNumbers(int a, int b)
                {
                        int result;

                        result = a + b;
```

```
                        return result;
                }
        }
}
```

Change Control Comments

The source control system should store the information about the history of code files.

```
namespace CleanCSharp.Comments.Dirty
{
        /* 10 Oct 2010 Sarah Smith - Created initial version
         * Edited 20 Oct 2010 Amrit P - change calculation method
         * Edited 20 Nov 2010 Jane Q - fix defect 4286
         */
        public class MyClass
        {
        }
}
```

Notice all these comments describing why and when the file was changed.

If a capable version control system is being used, these comments are unnecessary and should be removed.

If a version control system is not being used, one should be implemented, and then these comments deleted if they are not critical.

Comments as a Substitute for Self Documenting Code

Excessive comments may be an indication that the purpose of the code is unclear due to poor naming and/or code structure.

Take the following example:

```
namespace CleanCSharp.Comments.Dirty
{
        public class SimpleCalculator
        {
                // Add two numbers together
                public int Calculate(int a, int b)
                {
                        return a + b;
                }
        }
}
```

Here the comment `// Add two numbers together` is a substitute for a well-named method.

In order to make this code self documenting and remove the need for the comment, the method could be rewritten as follows:

```
namespace CleanCSharp.Comments.Clean
{
        public class SimpleCalculator
        {
                public int AddNumbers(int a, int b)
                {
                        return a + b;
                }
        }
}
```

Here the well-named method obviates the need for an explanatory comment.

Commented-Out Code

Often, especially in legacy code, blocks of code can exist, but in commented-out form.

Take the following example:

```
namespace CleanCSharp.Comments.Dirty
{
        public class AnotherSimpleCalculator
        {
                public int AddNumbers(int a, int b)
                {
                        // a = a + 42;

                        return a + b;
                }
        }
}
```

Here when the `AddNumbers` method is read there is some code that is commented out. What does this code mean? Was it accidentally commented out? Should it be uncommented?

This introduces uncertainty, disrupts the reader's flow and harms readability.

This may have been from a previous change being made but the developer making the change either forgot to remove the code or felt like it *might* be needed in the future.

If a version control system is being used, once a logical series of changes is complete, any temporarily commented-out code should be deleted. The previous version is available in the version control history if it is ever needed.

Pointless XML Documentation Comments

If the organisational code standards blindly require XML documentation comments on everything then often developers simply add them to satisfy these standards, without adding any additional benefit.

```
namespace CleanCSharp.Comments.Dirty
{
        /// <summary>
        ///
        /// </summary>
        public class BasicCalculator
        {
                /// <summary>
                /// Adds two numbers
                /// </summary>
                /// <param name="a"></param>
                /// <param name="b"></param>
                /// <returns></returns>
                public int AddNumbers(int a, int b)
                {
                        return a + b;
                }
        }
}
```

In the preceding example about half of all the lines are taken up with meaningless comments. These obscure the actual code and increase the time required to visually process the code.

Compare this with the following clean version:

```
namespace CleanCSharp.Comments.Clean
{
        public class BasicCalculator
        {
                public int AddNumbers(int a, int b)
                {
                        return a + b;
                }
        }
}
```

If the project is creating a public API to be used by other people then XML comments on public types and members can be of great use to the consuming developer. In this case there is a good argument for adding XML comments.

Acceptable Use of Comments

Comments can occasionally benefit the reader. For example if the intent is not easily expressible in code, or some other important information needs to conveyed to a potential future modifier. Comments should however be a last resort; the combined techniques in this book should be used to obviate the need for the majority of comments.

Naming Things

The names given to classes, methods, variables, and other items have a huge impact on the cleanliness and understandability of the system.

Names are the fundamental method by which the intent of the writer is expressed.

It is very easy not to pay enough attention when creating a name for something, but clean names are of such fundamental importance it is probably something that should be held in the highest regard by the writer.

Qualities of Clean Names

Some qualities of clean names include:

- Expressive
- Accurate
- Suitable length
- Pronounceable

The name of something should indicate to the reader why it exists in the first place and what it may be used for.

Expressive

Names should be expressive, they should clearly convey the intent of the writer.

For example, consider the following three variable declarations (from least to most expressive):

```
var n = "Jason"; // default name of new user
var name = "Jason";
var defaultNewUserName = "Jason";
```

Notice in the preceding example that a comment has been used as a substitute for self documenting code, to make up for the poorly named variable `n`.

Accurate

Names should be accurate, they should not not mislead the reader into thinking they mean something else. The following method is named `Add` when it performs multiplication:

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

Whilst this is clearly wrong, if a reader decides to call the method without reading the code that it contains, they will get unexpected results. A more subtle example of inaccurate method names are methods with side effects.

Suitable Length

The length of a name should be suitable for the scope and context in which it will be used. The length of a name should be descriptive enough to convey the intent but not so long as to tire the reader when reading.

In the following code, try to locate four names with poor lengths:

```
namespace CleanCSharp.Naming.SuitableLength.Dirty
{
        public class Cal
        {
                public int AddTwoNumbersTogetherAndReturnTheResult(
                   int theFirstNumberToAdd, int theSecondNumberToAdd)
                {
                        return theFirstNumberToAdd + theSecondNumberToAdd;
                }
        }
}
```

In the preceding code the public class `Cal` is available throughout the system (large scope) but it has an abbreviated name that is too short. What is a `Cal` and what does it do? From the name of the class alone there is no way to tell, requiring the reader to dig into the internals of the class before they can even get a high level idea of what the class does.

Next the method `AddTwoNumbersTogetherAndReturnTheResult` is too verbose. If the name of the class were better (for example `Calculator`) then simply naming the method `Add` would be sufficient.

The final two items are the method parameter names. With a cleanly named class and method, these names are too verbose. The parameters do not posses any uniqueness in terms of what they represent: if you call an `Add` method of a `Calculator` you expect to provide some numbers to add together. Because of this, the individual parameters do not have a high level of semantic meaning when compared against each other, so using more terse names such as `a` and `b` may be acceptable.

The following is a cleaner version:

```
namespace CleanCSharp.Naming.SuitableLength.Clean
{
        public class Calculator
        {
                public int Add(int a, int b)
                {
                        return a + b;
                }
        }
}
```

In most cases however, individual parameter names do have a greater amount of semantic meaning. If individual parameters have greater semantic meaning then they should use longer, more expressive names.

One consideration when using terse parameter names such `a` and `b` are the effect they have on searchability. For example searching for "a" in Visual Studio is likely to find many useless matches. If the preceding example is part of a large codebase where searching for text has become difficult, then longer names may be more useful as in the following code:

```
namespace CleanCSharp.Naming.SuitableLength.Clean
{
        public class Calculator
        {
                public int Add(int firstNumber, int secondNumber)
                {
                        return firstNumber + secondNumber;
                }
        }
}
```

In the following example, terse parameter names have been used when the parameters do have individual semantic meaning:

```
namespace CleanCSharp.Naming.SuitableLength.Dirty
{
        public class NewUserValidator
        {
                public bool ValidateName(string a, string b)
                {
                        return true; // for demo code purposes
                }
        }
}
```

When calling `ValidateName` what do `a` and `b` do? We could guess that `a` represents the first name and `b` the last name but the internals of the method would need to be examined to confirm this assumption. In this example it makes sense to have more semantically rich and expressive parameter names as in the following cleaner version:

```
namespace CleanCSharp.Naming.SuitableLength.Clean
{
        public class NewUserValidator
        {
                public bool ValidateName(string firstName, string lastName)
                {
                        return true; // for demo code purposes
                }
        }
}
```

There are other options for cleanliness here, for example there could be separate methods that validate the first name and last name, the name of the method could also be improved. See the chapter on clean methods for more information.

Pronounceable Names

Reading is a natural thing for the brain to do. When reading code we are interpreting not only the C# language itself (keywords, etc.) but also the names of things. Having names that are pronounceable helps readability as the brain has less work to do to interpret the word(s).

The following code has some examples of unpronounceable names:

```
namespace CleanCSharp.Naming.Pronounceable.Dirty
{
        public class NewUsrValidtr
        {
                public bool ValidateNme(string fstNme, string lstNme)
                {
                        return true; // for demo code purposes
                }
        }
}
```

When reading this there is additional mental effort required for the brain to parse names such as `fstNme` into "first name". Imagine also talking with a fellow developer about this code, how would `fstNme` be pronounced: "fust numee", "fastnu me"? A cleaner version would be as follows:

```
namespace CleanCSharp.Naming.Pronounceable.Clean
{
        public class NewUserValidator
        {
                public bool ValidateName(string firstName, string lastName)
                {
                        return true; // for demo code purposes
                }
        }
}
```

Naming Specific Items

The remainder of this chapter discusses the naming of specific items in C# programs. The following conventions confirm to the general guidelines expressed in the MSDN documentation[8].

Pascal Casing refers to using an upper case letter for the first letter in the name, for example "ThisIsPascalCase". When using (well-understood) acronyms with Pascal Casing, if the acronym is two letters then both letters are capitalised (e.g. "**IO**Encoder") and if the acronym is more than two letters only the first is capitalised, for example: "**Html**Encoder".

Camel Casing refers to using an lower case letter for the first letter in the name, for example "thisIsCamelCase". Using Camel Casing, the above two acronyms would be written: "ioEncoder" and "htmlEncoder".

Concerning acronyms, it is generally acceptable to use universally recognised acronyms in names; in the above examples "IO" for input/output and "Html" for "HyperText Markup Language".

If using brand names that feature specific capitalisation as part of the brand then the above rules may be overridden.

Namespaces

Namespaces should use Pascal Casing. They should accurately describe what the reader is likely to find contained within.

Generally speaking, catch-all generic namespaces like `Helpers` or `Utilities` should be avoided, though more specific versions such as `HtmlHelpers` or `StringUtilities` are usually more indicative of what may be contained within the namespace.

Namespaces should not usually be versioned. For example "MyCompany.AwesomeLibV1" and "MyCompany.AwesomeLibV2" would usually be considered bad practice.

MSDN[9] specifies the following naming convention for namespaces:

[8] http://msdn.microsoft.com/en-us/library/ms229002%28v=vs.110%29.aspx

[9] http://msdn.microsoft.com/en-us/library/ms229026%28v=vs.110%29.aspx

```
[CompanyName].[ProductOrTechnology].[Feature].[Subnamespace]
```

The argument for the CompanyName is to prevent conflicts when working with other libraries. It should be noted that not all authors or projects conform to this naming convention.

It is acceptable to use plural namespace names where appropriate, for example in the `System.Collections` namespace.

A namespace should also not be the same name as a type defined within that namespace, for example a namespace of `Chocolate` that contains a class also called `Chocolate`.

Interfaces

Interfaces should use Pascal Casing and be prefixed with the letter "I", for example `IControl`.

Interface names should use adjectives or adjective-phrases such as "ITransformable" and "ISavable" or nouns/noun-phrases such as "IShape" and "IShapeTransformer". In the case of nouns/noun-phrases it *may* indicate that the interface be better defined as a class or abstract class instead.

In the book "Clean Code", Robert C. Martin argues for the dropping of the "I" that precedes interface names. Whilst we should never keep doing something just because "that's the way it's always been done", the "I" convention in C# is one that may cause more confusion to the general reader if it were omitted, than benefit gained by its omission. As always the team should decide and all the developers should conform to the team's expectations.

Classes

Classes should use Pascal Casing and use nouns or noun-phrases such as `Customer`, `Order`, and `ProspectiveCustomer`.

Class name should never be prefixed with encodings such as "c" or "cls" such as `cCustomer` or `clsCustomer`.

If the class is the single implementation of an interface in a "class-interface pair" MSDN[10] then it should be named the same as the interface minus the "I". For example the class `ShapeTransformer` implements the interface `IShapeTransformer`.

If a class inherits from another class, it can *sometimes* improve readability by appending the base class name to the new class. For example in .NET the `ApplicationException` inherits from `Exception`. It does not always make sense to do this so should be something that is considered on a case-by-case basis, rather than an absolute rule.

Methods

The names of methods should use Pascal Casing and be verb or verb-phrases that signify the performing of some action. So a method called `Customer` is poorly named as it is a noun rather than a verb. On the other hand a method called `SaveCustomer` is a verb-phrase that indicates some action will be performed.

Properties

Properties should use be Pascal Casing and should use adjective or noun/noun-phases such as `Color` or `CustomerNumber`.

If the property represents a collection of things, rather than simply adding the word "List" or "Collection" as in `OrderList` it is usually more readable to simply pluralise the property name such as: `Orders`.

Events

Events should use Pascal Casing and be named using verb/verb-phrases such as `Clicked`, `Opened`, and `Closed`.

If the event describes a concept of something that happens before/after, use meaningful past or present tense verbs; so rather than `BeforeClose`, use `Closing` and instead of `AfterClose` use `Closed`.

[10] http://msdn.microsoft.com/en-us/library/ms229040%28v=vs.110%29.aspx

In event handlers, favour the parameter naming conventions "sender" and "e", for example: `object sender, SomeEventArgs e`.

Fields

There are no recommended guidelines from MSDN[11] for internal or private fields.

For public static fields and protected fields use Pascal Casing and noun/noun-phrases or adjectives.

One tradition for private fields is to prefix the identifier with an underscore, such as `int _age;` This is probably an unnecessary form of "encoding" other information in the names of things. In a small, highly focussed class the underscore would probably be unnecessary, so just: `int age;`. In this case follow Camel Casing rules.

Attributes

When defining custom attributes, use Pascal Casing and add the suffix "Attribute", for example use `public class ThisIsAwesomeAttribute : Attribute` rather than: `public class ThisIsAwesome : Attribute`.

Method Parameters

Use Camel Casing for method parameters and favour descriptive names for semantically rich parameters.

Variables

Use Camel Casing for local variables. The length of loop counters may be single letters, such as using `i` as a `for` loop counter. If the intent can be increased by using a more descriptive (longer) loop counter variable name then this is also acceptable.

[11] http://msdn.microsoft.com/en-us/library/ms229012%28v=vs.110%29.aspx

Booleans

When naming variables, methods, properties, etc. that represent Boolean values consider naming them so they can read in such a way as they answer a yes or no question. Consider how they would read when used in an `if` statement for example.

The following are some dirty examples:

```
namespace CleanCSharp.Naming.Booleans.Dirty
{
        class BooleanRelatedNames
        {
                public void SomeMethodWithBooleanVariables()
                {
                        bool close = false;

                        if (close)
                        {
                                // etc.
                        }

                        bool user = false;

                        if (user)
                        {
                                //
                        }
                }

                public bool Open { get; set; }

                public bool Value()
                {
                        return false;
                }
        }
```

```
}
```

And some cleaner versions:

```
namespace CleanCSharp.Naming.Booleans.Clean
{
        class BooleanRelatedNames
        {
                public void SomeMethodWithBooleanVariables()
                {
                        bool isClosed = false;

                        if (isClosed)
                        {
                                // etc.
                        }

                        bool loggedIn = false;

                        if (loggedIn)
                        {
                                //
                        }
                }

                public bool IsOpen { get; set; }

                public bool HasValue()
                {
                        return false;
                }
        }
}
```

Rather than just prefixing all Booleans with “is”, consider how the statement would read, maybe “has” improves the readability or maybe something like (in

the preceding example) `loggedIn` reads fine. Notice that even without "is"/"has", `loggedIn` answers a yes/no, true/false question: "is the user logged in?".

Generic Types Parameters

If a single letter is descriptive and obvious enough use the letter `T` such as in `Nullable<T>`. If a single letter ("T") is not descriptive enough use longer names with Pascal Casing such as `TKey`, `TValue`, `TUserSession`, etc.

Enums

Use Pascal Casing for enums and use a singular name unless the enum is a bitwise/flags enum in which case use plural names. Do not add suffixes such as "Flags" or "Enum" to names.

For example a bitwise/flags enum for display options would be named `DisplayOptions` not `DisplayOption`.

Some General Rules

Do not use almost identical / extremely similar names. These can easily be mistaken for one another by the reader and can lead to bugs.

Do not rely on case sensitivity to differentiate two things.

Do not try and assert a sense of superior intellect with deliberately cryptic names that challenge the reader.

Do not try to be a comedian and name things to be "cute" or "funny".

Do not add pointless pre/postfix context to all names: for example in a Tic Tac Toe game codebase, do not name things: `TTTBoard` `TTTPlayer` `TTTGame`, etc.

Do not use different words to express the same concept, for example naming some things "management" and some things "boss" when they refer to the same concept (in this example the "person in charge").

Methods

Methods in C# are the most common place where work actually takes place. Because of this, this entire chapter is dedicated to the creation of clean methods.

Method Size and Clarity

Generally speaking, clean methods are smaller than larger methods. A cleaner method will have fewer lines of code than a dirtier method.

When trying to reduce the number of lines in methods, this should not be taken as a simple case of trying to fit as many logical operations on a single line as possible using cryptic and hard to read code. If a method has a high level of functional cohesion with each line of code at the same level of abstraction, then it should not require hundreds of lines of code.

There are other factors (below) that contribute to clean methods, however method size is one of the key things to look for. This does not mean that a method with fewer lines of code is **automatically** clean, it could still be poorly named or have too many parameters for example.

Some signs that a method may be too large include:

- Many levels of indentation (e.g. multiple nested `if` statements)
- Doing too many logically different things
- Need to scroll up/down in the editor to read it all
- Work being performed at multiple abstraction levels

It is impossible to state an absolute maximum number of lines that a method should have, however here are some rough guidelines:

- less than 11 lines : probably clean
- 11 to 20 lines : possibly clean, but take notice
- 21 to 50 lines : probably dirty
- over 50 lines : dirty

Again these are just approximate guidelines - while method size is a good indicator of cleanliness, the other key factor is how many different things the method is doing.

Cohesive Methods

There are a number of different types of cohesion (how strongly related things are); one of these is *functional cohesion*.

All of the lines of code inside a method that has a high level of functional cohesion will all relate to performing a single logical task.

It is harder for methods to remain small if they are doing too much or too varied a task.

In the following code, notice that the `Process` method is doing two different things: validation and saving a `Customer` to a service.

```
namespace CleanCSharp.Methods.Dirty
{
        class Utils
        {
                public int Process(Customer customer)
                {
                        if (string.IsNullOrWhiteSpace(customer.FirstName)
                            || string.IsNullOrWhiteSpace(customer.LastName))
                        {
                                return -1;
                        }
                        else
                        {
                                var service = new CustomerService();
```

```
                                    if (!service.Save(customer))
                                    {
                                            return -1;
                                    }
                                    else
                                    {
                                            return 1;
                                    }
                            }
                    }
            }
}
```

In the preceding code (dirty naming aside) the method `Process` is responsible for doing two separate logical things: validating that the first name or last name is not null/empty; and saving the customer (using a `CustomerService`). The number of lines of code in the method body (excluding blank lines) is 16, which falls into the category "possibly clean, but take notice". This method is clearly not clean because it is doing too much: validating **and** saving.

This is a key point. The cleanliness of code is a holistic matter. Just because the number of lines of code in a method may seem "clean", the method can still be dirty in other ways.

The `Process` method in the preceding code has low functional cohesion. Also because it is doing two different logical things, it is also harder for the method to remain smaller.

This method could be refactored into two separate methods, each method now being more functionally cohesive as shown in the following code:

```
namespace CleanCSharp.Methods.Clean
{
        class Utils
        {
                public int Process(Customer customer)
                {
                        const int customerNotSaved = -1;
                        const int customerSavedSuccessfully = 1;

                        if (!IsValidCustomer(customer))
                        {
                                return customerNotSaved;
                        }

                        if (!SaveCustomer(customer))
                        {
                                return customerNotSaved;
                        }

                        return customerSavedSuccessfully;
                }

                private bool IsValidCustomer(Customer customer)
                {
                        if (string.IsNullOrWhiteSpace(customer.FirstName)
                            || string.IsNullOrWhiteSpace(customer.LastName))
                        {
                                return false;
                        }

                        return true;
                }

                private bool SaveCustomer(Customer customer)
                {
```

```
                    var service = new CustomerService();

                    var successfullySaved = service.Save(customer);

                    return successfullySaved;
                }
        }
}
```

In this cleaner version, the `Process` method is now only 11 lines of code. Some constants have been introduced to replace the magic values (-1 and 1) and the nesting has been reduced by eliminating an unnecessary `else`. There is more that could be done here, but for the purpose of exploring functional cohesiveness, notice that both the `IsValidCustomer` and `SaveCustomer` methods each now do one well-defined thing. Both of these methods have high functional cohesion, because of this, notice that the methods are also short: 5 and 3 lines of code respectively.

Mixing Abstraction Levels

In the preceding refactored code all three methods now contain code at the same level of abstraction. In the `Process` method the details of how a customer is validated and where a customer is stored are left to the methods at a lower abstraction level. The `Process` method can be read at a higher abstraction level and if the reader needs the details, they can examine the lower level `IsValidCustomer` and `SaveCustomer` methods.

From the point of view of the reader, their brain is not having to mentally switch between varying levels of abstraction in a single method. This concept is one of the factors that contribute to Structuring Programs for Readability.

Action or Answering Methods

One way to determine if a method may not be functionally cohesive is to evaluate it as either:

- Performs some action/function; **or**
- Answers a question for the caller

If a method is doing both of these things there may be an opportunity to refactor it.

Take the following `IsValidCustomer` method:

```
private bool IsValidCustomer(Customer customer)
{
        if (string.IsNullOrWhiteSpace(customer.FirstName)
            || string.IsNullOrWhiteSpace(customer.LastName))
        {
                return false;
        }

        return true;
}
```

This clearly falls into the "answers a question for the caller" category; it allows the caller to answer the question "is the customer valid?".

Contrast this with the following `SaveCustomer` method:

```
private bool SaveCustomer(Customer customer)
{
        var service = new CustomerService();

        var successfullySaved = service.Save(customer);

        return successfullySaved;
}
```

This method is both "performing some action/function" **and** "answering a question for the caller". It is performing the action of saving a customer and also answering the caller's question "did the customer get saved successfully?".

> In this example the Boolean return value of the method is being used as an error flag. Ideally this would be refactored to use an exception based approach.

Like the other aspects that make up clean code, there may be valid reasons to allow these two concepts to be mixed in a single method. For example, in ORM code when the `Update`, `SaveChanges`, etc. methods are called they may return an integer representing the number of database records affected.

Method Parameters

The more parameters a method has, the harder it becomes to understand.

Methods can be divided into a number of categories based on the number of parameters they take:

- zero parameters : clean
- one parameter : probably clean
- two parameters : possibly dirty
- three parameters : probably dirty
- more than three parameters : dirty

Again these are just guidelines, the other factors of method cleanliness should also be considered.

Another consideration when it come to the number of method parameters is that of testing. The more parameters (and range of input values) a method has, the harder it is to test. This is because the combinations of all possible input parameter values is going to rapidly increase as more parameters are added.

Methods with Zero Parameters

Methods with no parameters are the easiest to understand because the reader or maintainer does not have to expend mental cycles on thinking what should be passed to the method when calling it.

Methods with One Parameter

Methods with one parameter (monadic methods) are the next easiest to understand.

The `IsValidCustomer` method is an example of a monadic method:

```
private bool IsValidCustomer(Customer customer)
{
        if (string.IsNullOrWhiteSpace(customer.FirstName)
            || string.IsNullOrWhiteSpace(customer.LastName))
        {
                return false;
        }

        return true;
}
```

This method takes a single parameter called `customer`. Even though a `Customer` contains multiple properties it is still classed as a single parameter.

Monadic methods generally fall more easily into one of the two "action" or "answering method types".

Methods with Two Parameters

Methods with two parameters (dyadic methods) start to become harder to understand.

When reading a dyadic method, mental effort must be expended to now understand two parameters.

Because there are now two parameters, it can become easier for bugs to be introduced. For example if a method has two parameters, both of type `string` then it is easier to get the values passed to these two strings mixed up.

If it is a simple matter to refactor a dyadic method into method(s) that take only a single parameter this may increase readability and cleanliness, but only where it makes sense and is not done in a contrived way. For example a method to plot a co-ordinate could naturally require two parameters: `x` and `y`; or a method to add two numbers together for example.

Methods with Three Parameters

Methods with three parameters (triadic methods) are harder still to understand. There are now three parameters to be parsed by the reader and more potential for accidentally mixing up what is passed to the parameters.

Methods with More Than Three Parameters

Methods that have four or more parameters are very hard to read, call, and maintain. These methods should be refactored and reduced to methods that taker fewer parameters or alternatively the individual parameters encapsulated inside a class.

Refactoring to Reduce the Number of Parameters

One way to deal with cumbersome methods that take too many parameters is to wrap the parameters into their own class and instead pass a single argument of this class instead.

Take the following method that plots a coordinate:

```
public void Plot(int x, int y)
{
        // etc
}
```

This method could be reduced to monadic form by introducing the following `PlotPoint` class:

```
class PlotPoint
{
        public int X { get; set; }
        public int Y { get; set; }
}
```

The method can now be written using the monadic form as follows:

```
public void Plot(PlotPoint point)
{
        // etc
}
```

If the original method also contained a parameter to choose the size of the plotted point then this refactoring would have reduced a triadic method to a monadic one by also adding a `Size` property to the `PlotPoint` class.

Params

The `params` keyword allows a method parameter to take a variable number of values when the method is called. Generally speaking a `params` parameter still counts as a single parameter. For example a method with a single `params` parameter is still a monadic method. However, `params` should not be used as a hack to reduce the number of parameters in a method.

Output Parameters

C# allows a parameter to be defined as an `out` parameter. This means that what looks like an input parameter can also function as an output from the method, perhaps in addition to an actual return value.

Output parameters may be occasionally useful, for example the various `TryParse` methods in the .NET framework that return a boolean if a value can be parsed from a string, in addition to the output value in an `out` parameter.

The natural interpretation of method parameters is that they pass some information **into** the method for it to use, rather than as a mechanism for the method to return a value. Because of this, `out` parameters should be avoided unless there is a good reason to use them.

Named Arguments

If a method with multiple parameters exists but cannot be refactored for some reason, one way to increase the readability of the calling code is to explicitly state the parameter names.

Assuming the following `Plot` method can not be refactored:

```
public void Plot(int x, int y, int size)
{
        // etc
}
```

This method takes three parameters all of type `int` and can be called using positional arguments with `Plot(10, 15, 10);` This is not particularly readable without needing to dig into the method parameters. This call could instead be re-written using named arguments: `Plot(x: 10, y: 15, size: 10);` This is slightly more verbose but removes the ambiguity of what each argument means.

If a monadic method needs a named argument to improve readability, it may be a sign that the method itself is not well-named.

Boolean Switching Arguments

If a method defines a Boolean parameter that decides what the method does, this can indicate that the method is doing more than one logical thing and should be refactored.

The following code shows an example of a method with a “boolean switching argument”:

```
namespace CleanCSharp.Methods.Dirty
{
        class BooleanSwitchingArgumentsExample
        {
                public void CallingCode()
                {
                        if (DateTime.Now.Hour < 12)
                        {
                                OutputGreeting(true);
                        }
                        else
                        {
                                OutputGreeting(false);
                        }
                }

                public void OutputGreeting(bool isMorning)
                {
                        if (isMorning)
                        {
                                Console.WriteLine("Good Morning");
                        }
                        else
                        {
                                Console.WriteLine("Good Day");
                        }
                }
        }
}
```

In the preceding code, the `OutputGreeting` method defines a boolean switching argument that decides what the method does. Notice the calling code is not very understandable: `OutputGreeting(true)` and `OutputGreeting(false)`.

The following code is a refactored version with two separate methods for the two kinds of greeting. Notice in this code that the two new methods do not require any

parameters which further contributes to readability.

```
namespace CleanCSharp.Methods.Clean
{
        class BooleanSwitchingArgumentsExample
        {
                public void CallingCode()
                {
                        if (DateTime.Now.Hour < 12)
                        {
                                OutputMorningGreeting();
                        }
                        else
                        {
                                OutputDaytimeGreeting();
                        }
                }

                private static void OutputDaytimeGreeting()
                {
                        Console.WriteLine("Good Day");
                }

                private static void OutputMorningGreeting()
                {
                        Console.WriteLine("Good Morning");
                }
        }
}
```

Multiple Returns

It is acceptable to have multiple `return` statements in a method if this improves the clarity. This may also reduce the number of lines of code in the method.

In the following example, a strict “only ever have one return statement” policy has been implemented:

```
namespace CleanCSharp.Methods.Dirty
{
        class MethodExitPoints
        {
                public string GenerateAgeAppropriateGreeting(
                            int customerAgeInYears)
                {
                        string greeting;

                        if (customerAgeInYears < 16)
                        {
                                greeting = "Yo!";
                        }
                        else if (customerAgeInYears < 25)
                        {
                                greeting = "Hi there";
                        }
                        else
                        {
                                greeting = "Dear Sir/Madam";
                        }

                        return greeting;
                }
        }
}
```

Compare this to the following method that uses multiple `return` statements:

```
namespace CleanCSharp.Methods.Clean
{
        class MethodExitPoints
        {
                public string GenerateAgeAppropriateGreeting(
                            int customerAgeInYears)
                {
                        if (customerAgeInYears < 16)
                        {
                                return "Yo!";
                        }

                        if (customerAgeInYears < 25)
                        {
                                return "Hi there";
                        }

                        return "Dear Sir/Madam";
                }
        }
}
```

This version of the `GenerateAgeAppropriateGreeting` method is easier to read and is shorter than the single `return` version.

Code Duplication in Methods

As in other areas of the codebase, if multiple methods have a lot of duplicated code, this should ideally be refactored out into a method that contains the common (previously duplicated) code, so that it can be shared by the other methods.

Methods with Side Effects

A method with side effects misleads the caller. For example if a `ValidateCustomer` method also saved the customer to the database, this saving is a hidden side effect

and is clearly bad practice. Not only this, it also means the method is doing more than one logical thing.

Other side effects may include the unexpected changing of field/property values, raising unexpected events, and changing input method parameter object values.

Structuring Programs for Readability

The overall structure of large codebases can help to enhance readability or reduce it.

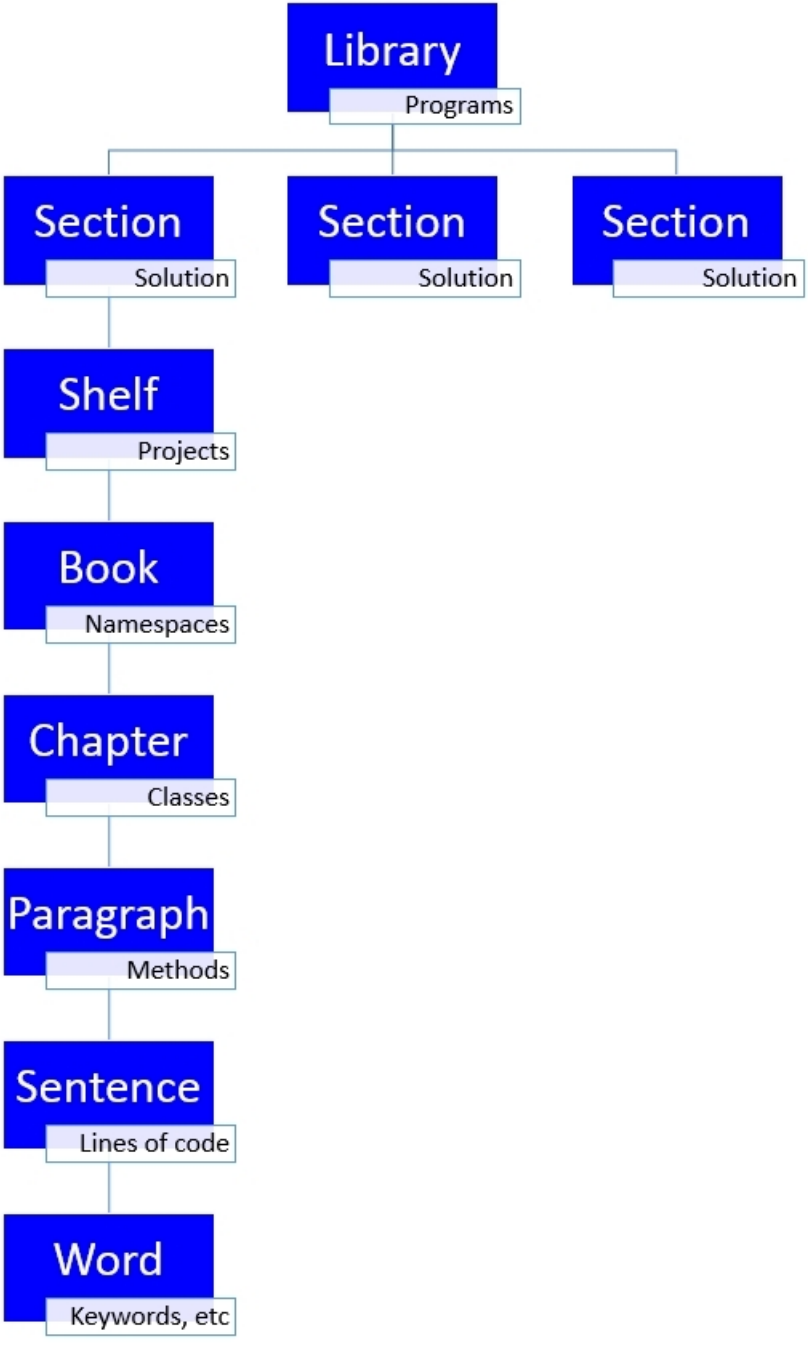
One way to improve readability is to think of the reader's brain as only being able to work at a similar level of abstraction at any given time.

For example, suppose a method that is named at a higher abstraction level such as `ValidateCustomer`. The level of abstraction that the brain would expect for this method (and the code it contains) is to describe validation/business rules. If the `ValidateCustomer` method also performs work at a lower abstraction level such as setting database fields, then the reader's brain is having to switch mental abstraction levels when trying to read and understand the code.

Levels of Abstraction

One metaphor to represent this idea of abstractions is that of traditional paper-based books in a public lending library.

The following diagram represents the abstraction levels of source code using this metaphor.



Levels of Abstraction

While this metaphor is not perfect, it seeks to illustrate the fact that items higher up in the diagram need a different level of cognitive processing than items lower down.

Imagine a library building that had no shelves or books but rather pages of books stuck to all the walls of the building. In this case there are greatly reduced levels of abstraction, essentially reduced to pages and paragraphs. Clearly this harms readability, even it might make a good art installation.

Grouping concepts into similar levels of abstraction can help in a number of ways. Firstly it can help to improve navigation around the codebase. For example in a traditional book it is easy to gradually "drill down" to find the right paragraph by scanning the chapters first (higher abstraction level, less cognitive load) and then drilling down into specific paragraphs. The second way is the reduction of unnecessary cognitive processing. If for example a method is operating at multiple abstractions levels, then it takes more mental energy to "read between the lines" and only focus on the abstraction level that is required for the current task.

Method Level Abstractions

As a simple example, assume that the following classes exist:

```
public class Customer
{
    public string FirstName { get; set; }
    public string SecondName { get; set; }
    public bool IsPriorityCustomer { get; set; }
    public decimal AnnualIncome { get; set; }
}

    public class ProspectiveCustomer
{
    public string FirstName { get; set; }
    public string SecondName { get; set; }
    public decimal AnnualIncome { get; set; }
}
```

The following version of the `ProspectiveCustomerValidator` class has mixed abstraction levels in the same method:

```
public class ProspectiveCustomerValidator
{
    public Customer CreateValidatedCustomer(
                            ProspectiveCustomer prospectiveCustomer)
    {
        if (string.IsNullOrWhiteSpace(
                            prospectiveCustomer.FirstName))
        {
            throw new ArgumentException("Invalid FirstName");
        }

        if (string.IsNullOrWhiteSpace(
                            prospectiveCustomer.SecondName))
        {
            throw new ArgumentException("Invalid SecondName");
        }


        var newValidCustomer = new Customer
                    {
                        FirstName = prospectiveCustomer.FirstName,
                        SecondName = prospectiveCustomer.SecondName
                    };


        if (prospectiveCustomer.AnnualIncome > 100000 )
        {
            newValidCustomer.IsPriorityCustomer = true;
        }

        return newValidCustomer;
    }
}
```

Notice in the preceding code, other than the method doing too many different things, it is also hard to get an overall higher-level-of-abstraction understanding of what is going on. For example the exact rules that determine what a valid `FirstName` are are lower in abstraction level than the overall process of creating a new customer. Also notice the logic to determine if a new customer is a priority customer includes lower level detail such as the annual income being greater than 100000.

Compare the preceding code to the following refactored version:

```
public class ProspectiveCustomerValidator
{

    // Higher abstraction level

    public Customer CreateValidatedCustomer(
                            ProspectiveCustomer prospectiveCustomer)
    {
        EnsureValidDetails(prospectiveCustomer);

        var validatedCustomer =
                        CreateNewCustomerFrom(prospectiveCustomer);

        SetCustomerPriority(validatedCustomer);

        return validatedCustomer;
    }

    // Medium abstraction level

    private static void EnsureValidDetails(
                            ProspectiveCustomer prospectiveCustomer)
    {
        EnsureValidFirstName(prospectiveCustomer);

        EnsureValidSecondName(prospectiveCustomer);
    }
```

```
    private static Customer CreateNewCustomerFrom(
                            ProspectiveCustomer prospectiveCustomer)
    {
        return new Customer
        {
            FirstName = prospectiveCustomer.FirstName,
            SecondName = prospectiveCustomer.SecondName,
            AnnualIncome = prospectiveCustomer.AnnualIncome
        };
    }

    // Low abstraction level

    private static void EnsureValidFirstName(
                            ProspectiveCustomer prospectiveCustomer)
    {
        if (string.IsNullOrWhiteSpace(prospectiveCustomer.FirstName))
        {
            throw new ArgumentException("Invalid FirstName");
        }
    }

    private static void EnsureValidSecondName(
                            ProspectiveCustomer prospectiveCustomer)
    {
        if (string.IsNullOrWhiteSpace(
                                prospectiveCustomer.SecondName))
        {
            throw new ArgumentException("Invalid SecondName");
        }
    }

    private static void SetCustomerPriority(Customer customer)
    {
```

```
        if (customer.AnnualIncome > 100000)
        {
            customer.IsPriorityCustomer = true;
        }
    }
}
```

In the preceding code, the class has been refactored in an attempt to represent different levels of abstractions (as noted by the comments that have been included purely for demo purposes). This means that a reader can more easily choose what abstraction level they need to perform a particular task. For example if the reader just wants a high level understanding of the steps to convert a `ProspectiveCustomer` to a `Customer` their brain can operate at that level of abstraction without being distracted by lower level details (such as specifics on annual income numbers).

Errors and Exceptions

Error handling is an essential part of most codebases. There are a number of ways of handling errors and allowing calling code to respond to errors.

Returning Error Codes

One way to indicate to the caller that an error has occurred is to return some value from a method call to indicate whether the method was successful or not. The following code shows an example of a method that uses an `int` return value to signal to the caller what the outcome of the operation was.

```
namespace CleanCSharp.Errors.Dirty
{
        public class SomeClass
        {
                public int DoSomeProcess(int? id)
                {
                        if (id == null)
                        {
                                return -1; // null id
                        }

                        string data = LoadData();

                        if (string.IsNullOrWhiteSpace(data))
                        {
                                return -2; // data is corrupt
                        }

                        ProcessData(data);
```

```
                return 0; // no error, all good
        }

        private string LoadData()
        {
                return "some data";
        }

        private void ProcessData(string data)
        {
                // do something
        }
    }
}
```

A consumer of this code will need to check the various status codes to know what has happened as the following code shows.

```
namespace CleanCSharp.Errors.Dirty
{
    public class ConsumerOfSomeClass
    {
        public void Consume()
        {
            var sc = new SomeClass();

            const int idToProcess = 42;

            int returnCode = sc.DoSomeProcess(idToProcess);

            switch (returnCode)
            {
                case -1: // null id
                    // do something
```

```
                                        break;
                                case -2: // corrupt data
                                        // do something
                                        break;
                                case 0: // no error
                                        Save(idToProcess);
                                        break;
                        }
                }

                private void Save(int id)
                {
                        // save
                }
        }
}
```

Notice in the preceding code that there is a lot of clutter; the business logic/application flow is harder to recognize due to all the error handling code.

There are a number of other problems with the error code approach. First, every time the `DoSomeProcess` method is called anywhere in the codebase, the calling code must check the return codes. Assuming the programmer remembers to do this and the correct error codes are used, code duplication creeps in and readability is reduced. Second, these magic numbers representing the error codes do not have a lot of meaning, for example when a reader sees `-2` they will need to do further reading to try and understand the error that is being handled.

Returning error codes is also limited in other ways, for example if the method already returns a value, how is an additional error return code added? The same limitation exists when accessing properties.

Using Exceptions

Rather than returning error codes, in C#, error handling can be better implemented using exceptions. Exceptions can simplify the calling code and make it easier for

the reader to reason about the error handling that has been implemented. In C#, the `throw` statement is used to create an error condition. The `try`, `catch`, `finally` keywords can be used to detect and respond to error conditions.

The following code shows a refactored version of `SomeClass` that uses exceptions rather than error return codes. Notice that the `DoSomeProcess` method appears more readable than the preceding version.

```
using System;
using System.IO;

namespace CleanCSharp.Errors.Clean
{
        public class SomeClass
        {
                public void DoSomeProcess(int? id)
                {
                        if (id == null)
                        {
                                throw new ArgumentNullException("id");
                        }

                        string data = LoadData();

                        ProcessData(data);
                }

                private string LoadData()
                {
                        var demoData = "";

                        if (string.IsNullOrWhiteSpace(demoData))
                        {
                                throw new
                                  InvalidDataException(
                                    "The data stream contains no data.");
```

```
                        }

                        return demoData;
                }

                private void ProcessData(string data)
                {
                        // do something
                }
        }
}
```

The consumer can also now be changed to the following:

```
using System;
using System.Diagnostics;
using System.IO;

namespace CleanCSharp.Errors.Clean
{
        public class ConsumerOfSomeClass
        {
                public void Consume()
                {
                        var sc = new SomeClass();

                        const int idToProcess = 42;

                        try
                        {
                                sc.DoSomeProcess(idToProcess);
                        }
                        catch (ArgumentNullException ex)
                        {
                                // null id
```

```
                        // do something such as logging

                        // if cannot respond to this
                        // exception propagate up the
                        // call stack
                        throw;

                        // Notice the throw is not: throw ex;
                }
                catch (InvalidDataException ex)
                {
                        // bad data

                        // do something

                        throw;
                }
                catch (Exception ex)
                {
                        // any other exceptions that may occur

                        // do something

                        throw;
                }

                Save(idToProcess);
        }

        private void Save(int id)
        {
                // save
        }
}
```

```
}
```

Notice that rather than harder to understand error codes, the reader can now understand what types of error may occur by looking at the exception types in the catch blocks. Also notice that the exceptions in the catch blocks are caught from the more specific exceptions down to the most general (the `Exception` class).

Usually, an exception is only caught if it needs to be handled in some way. When an exception has been caught in a `catch` block, the problem can either be fixed or if it cannot be fixed it can be propagated "rethrown" to higher level callers. When rethrowing exceptions, using `throw;` will throw the same exception to higher level callers, if `throw ex;` is used the exception will still be rethrown but the stack trace of the thrown exception will not be that of the originally caught exception.

Supplied Framework Exceptions

There are many pre-defined exceptions that can be used, including:

- System.ArgumentException
- System.ArgumentNullException
- System.ArgumentOutOfRangeException
- System.InvalidOperationException
- System.NotSupportedException

There are a number of good practices when using `System.Exception` or `System.SystemException`:

- Do not manually throw either of these exceptions
- In framework code, only catch these exceptions if they are rethrown
- Only catch these exceptions in top level handlers

For more detailed guideline see MSDN[12].

[12] https://msdn.microsoft.com/en-us/library/vstudio/ms229007%28v=vs.100%29.aspx

Defining Custom Exceptions

If existing exceptions do not satisfy a particular use case then custom exceptions can be defined.

The following are guidelines when defining custom exception types:

- Generally, inherit from System.Exception
- End the custom class name with "Exception"
- Implement the standard exception constructor overloads
- Do not have deep exception hierarchies
- Implement `ISerializable` if the exception will be used across app domain/remoting boundaries

The following code defines a custom exception called `MyCustomException` that provides the three standard constructors plus an additional custom property called `SomeId` that can be used to pass additional information to callers/handlers when this exception is thrown.

```
using System;

namespace CleanCSharp.Errors.Clean
{
        public class MyCustomException : Exception
        {
                public MyCustomException()
                {
                }

                public MyCustomException(string message) : base(message)
                {
                }

                public MyCustomException(string message, Exception inner)
                        : base(message, inner)
```

```
                {
                }

                public int SomeId { get; set; }
        }
}
```

Alternatives to Error Codes, Exceptions, and Returning Nulls

Try Methods

In addition to throwing exceptions, an additional option that can be provided to consumers is to call a Try method. A Try method returns true if the operation succeeded and false if it failed (and does *not* throw an exception). The Try method usually has an `out` parameter by which the result of the operation can be passed to the caller.

The following code shows an example of implementing a method (`Parse`) that throws an exception and a companion Try version (`TryParse`).

```
using System;

namespace CleanCSharp.Errors.Clean
{
        public class Color
        {
                public static Color Parse(string colorName)
                {
                        if (!IsValidColor(colorName))
                        {
                                throw new ArgumentOutOfRangeException(
                                        "colorName",
                                        colorName + " is not a valid color");
```

```
                }

                return new Color();
        }


        public static bool TryParse(string colorName, out Color color)
        {
                if (!IsValidColor(colorName))
                {
                        color = null;
                        return false;
                }

                color = new Color();
                return true;
        }

        private static bool IsValidColor(string colorName)
        {
                return true; // validation logic goes here
        }
    }
}
```

Calling code can make use of the `TryParse` method as shown in the following code.

```
Color c;

if (!Color.TryParse("blue", out c))
{
        // error handling locic
}
else
{
        // c now is a valid Color object
}
```

Special Case Design Pattern

An alternative to returning error codes, throwing exceptions, or returning a `null` value is to return a special case object. This special case object looks identical (same interface) to a "real" object but means that the consumer does not have to write null checking logic or possibly catch `ArgumentNullExceptions`.

The following code shows a simple example where the `EmailCustomer` method has to first perform a null check before sending emails.

```
namespace CleanCSharp.Errors.Dirty
{
        public class Customer
        {
                public string EmailAddress { get; set; }

                public void SendEmail(string message)
                {
                        // send email to customer
                }
        }

        public static class CustomerFinder
        {
                public static Customer Find(int id)
```

```
                {
                        // if cannot find customer
                        return null;
                }
        }

        public class CustomerFinderConsumer
        {
                public void EmailCustomer()
                {
                        Customer c = CustomerFinder.Find(42);

                        // consumer has to check for nulls
                        if (c != null)
                        {
                                c.SendEmail("Hello!");
                        }
                }
        }
}
```

A refactored version follows, notice that no null check is required as `CustomerFinder.Find` never returns null, instead it returns the special case `CustomerNotFound` whose `SendEmail` does nothing.

```
namespace CleanCSharp.Errors.Clean
{
        public class Customer
        {
                public string EmailAddress { get; set; }

                public virtual void SendEmail(string message)
                {
                        // send email to customer
                }
```

```
        }

        public class CustomerNotFound : Customer
        {
                public override void SendEmail(string message)
                {
                        // DO NOTHING
                }
        }

        public static class CustomerFinder
        {
                public static Customer Find(int id)
                {
                        // if cannot find customer
                        return new CustomerNotFound();
                }
        }

        public class CustomerFinderConsumer
        {
                public void EmailCustomer()
                {
                        Customer c = CustomerFinder.Find(42);

                        c.SendEmail("Hello!");
                }
        }
}
```

Visual Formatting

The way that source code is visually formatted can have a great effect on the readability of programs. Even if the code is otherwise clean, poor visual formatting can still hurt readability.

The following code shows the version of `ProspectiveCustomerValidator` that maintains some of the other clean code practices outlined in this book, but with some vertical whitespace removed.

```
using System; using CleanCSharp.StructringCode;

namespace CleanCSharp.VisualFormatting.Dirty{
        public class ProspectiveCustomerValidator{
                public Customer CreateValidatedCustomer(
                                        ProspectiveCustomer prospectiveCustomer){
                        EnsureValidDetails(prospectiveCustomer);
                        var validatedCustomer=CreateNewCustomerFrom(
                                        prospectiveCustomer);
                        SetCustomerPriority(validatedCustomer);
                        return validatedCustomer;
                }
                private static void EnsureValidDetails(
                                        ProspectiveCustomer prospectiveCustomer){
                        EnsureValidFirstName(prospectiveCustomer);
                        EnsureValidSecondName(prospectiveCustomer);}
                private static Customer CreateNewCustomerFrom(
                                        ProspectiveCustomer prospectiveCustomer){
                        return new Customer{
                                FirstName=prospectiveCustomer.FirstName,
                                SecondName=prospectiveCustomer.SecondName,
                                AnnualIncome=prospectiveCustomer.AnnualIncome};}
```

```
                private static void EnsureValidFirstName(
                                        ProspectiveCustomer prospectiveCustomer){
                        if (string.IsNullOrWhiteSpace(
                                        prospectiveCustomer.FirstName)){
                                throw new ArgumentException("Invalid FirstName");
                        }}
                private static void EnsureValidSecondName(
                                        ProspectiveCustomer prospectiveCustomer){
                        if (string.IsNullOrWhiteSpace(
                                        prospectiveCustomer.SecondName)){
                                throw new ArgumentException("Invalid SecondName")
                        }
                }private static void SetCustomerPriority(Customer customer)
                {
                        if (customer.AnnualIncome > 100000){
                                customer.IsPriorityCustomer = true;
                        }
                }
        }
}
```

Notice that when reading the preceding code that the eye has to try very hard to differentiate different logical parts of the code.

There are some accepted design principles that can be applied to source code formatting. These principles are often referred to as the Gestalt principles.

The Principle of Proximity

The Gestalt principle of Proximity states that things that are closer to each other seem more related. For example, in a restaurant we perceive that people on the same table are more related to each other than to people on different tables.

In the following code, there is an automatic sense that method `A` and `B` feel related to each other. This is because they are in closer proximity to each other, method `C` feels more distant and unrelated.

```
namespace CleanCSharp.VisualFormatting.ExampleCode
{
        public class Class1
        {
                public void A()
                {
                        //
                }

                public void B()
                {
                        //
                }









                public void C()
                {
                        //
                }
        }
}
```

The principle of Proximity can be used when declaring variables as the following code demonstrates.

```
public void SomeMethod()
{
        bool isVipCustomer;
        int years;

        string x;

        decimal y;
}
```

In the preceding code there is a sense that `isVipCustomer` and `years` are related (though `years` should be renamed to something like `yearsAtVipStatus` rather than relying solely on Proximity).

Proximity also applies to where variables are declared, for example the traditional approach of declaring all variables at the top of the method (lower proximity), versus declaring them throughout the method close to where they are first needed (higher proximity).

The Principal of Similarity

Things that are similar in some way seem more related, this similarity could be in shape, size, color, texture, etc.

IDEs such as Visual Studio use similarity of color to help us perceive what a piece of code is; blue for keywords, green for comments, etc.

Naming conventions in source code can increase or decrease the level of similarity to other pieces of code. In the following code `_firstName` and `_lastName` feel similar due to their preceding underscores. `fullName` does not have a preceding underscore so is perceived as less similar.

```
_firstName = "Sarah";
_lastName = "Smith";
fullName = _firstName + " " + _lastName;
```

The perception of "difference" can be heightened by also using proximity to separate the lines of code as in the following example.

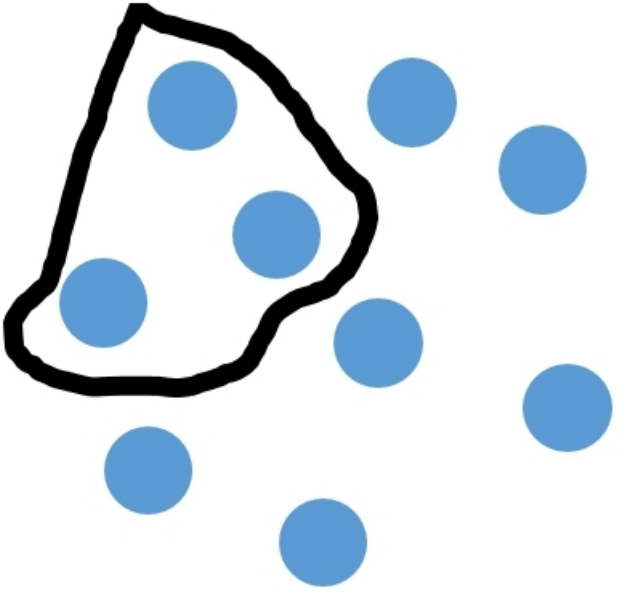
```
_firstName = "Sarah";
_lastName = "Smith";

fullName = _firstName + " " + _lastName;
```

The Principal of Uniform Connectedness

The feeling of relatedness can be strongly increased by using the principle of Uniform Connectedness.

The following diagram show a series of dots, all the same size and color, however by enclosing some of the dots, they are perceived as being related in some way.



Uniform Connectedness

In C#, code blocks contain groups of (hopefully) related code - lines of code are contained inside sets of braces `{}`. These braces *contain* code (much like the line containing the dots) and can increase the feeling of relatedness.

Take the following code:

```
bool a;
bool b;
bool c;
bool d;
```

These Booleans feel related due to their proximity and their similarity of names (single letters, ascending order). If these same variables are contained in additional braces then this changes the perception of their relatedness as the following code demonstrates.

```
{
        bool a;
        bool b;
}
{
        bool c;
        bool d;
}
```

In the preceding code there are now two strongly distinct groups.

The Principal of Symmetry

Humans tend to like the appearance of symmetry. One style of C# code that may appear jarring is the use of unbalanced (asymmetrical) braces. In the following code, the first (asymmetrical) `if` statement may feel less symmetrical than the second.

```
if (true) {
        Console.Write("true");
}

if (true)
{
        Console.Write("true");
}
```

Cohesion and Coupling

The concepts of cohesion and coupling are important concerns when it comes to creating clean C# code.

Cohesion

Cohesion is the "relatedness" of different pieces of code, it is the degree to which pieces of code belong together. Often the level of cohesion is referred to as "low cohesion" or "high cohesion" though there are a number of "degrees of cohesion", rather than it being a binary proposition.

The concept of cohesion can be applied at different levels in the source code. For example an individual method could exhibit low cohesion if the lines of code within it do not really belong together. The same thinking can be applied at the class level, i.e. do all the methods, properties, etc. belong together? At a higher level, cohesion can also be applied to namespaces and assemblies.

The following code shows a class that would be described as having low cohesion.

```
namespace CleanCSharp.CohesionAndCoupling.Dirty
{
        public static class Utils
        {
                public static int AddNumbers(int a, int b)
                {
                        return a + b;
                }

                public static bool IsValidCustomerAge(int ageInYears)
                {
                        return ageInYears > 17 && ageInYears < 100;
```

```
                }

                public static void ProcessOrder(Order order)
                {
                        // Validate order
                        if (order.Quantity < 1)
                        {
                                throw new ArgumentException(
                                        "order quantity must be greater that 0");
                        }

                        // code to save to SQL db

                }
        }
}
```

In the preceding code, the `Utils` class itself exhibits low cohesion. The methods inside the class are not highly related, the addition of two numbers has no relation to validating a customer's age or processing an `Order`. The `ProcessOrder` method itself also exhibits low cohesion; the method is validating a customer **and** saving it to a database. Notice in both these cases the names of the class and method are not very specific, the class itself is called `Utils` which does not have much meaning. The `ProcessOrder` method is also very non-specific. Often these generalized, non-specific names are an indication that cohesion may be lacking.

There are various degrees of cohesion, from best to worst:

- Functional Cohesion: code that does highly related and well-defined task(s)
- Sequential Cohesion: code grouped by the (data) output of one thing being the (data) input to the next, like a car assembly line
- Communicational Cohesion: code grouped together because it uses the same set/table/entity of data
- Procedural Cohesion: code grouped by being parts of a task that need to be executed in a particular order

- Temporal Cohesion: code grouped by when it is executed in the program execution life cycle
- Logical Cohesion: code grouped by seemingly logically related operations, but being functionally different, such as a `LoadCustomer` class that can load from a database, a CSV file, and an Excel file.
- Coincidental Cohesion: code is grouped arbitrarily without meaning, such as the preceding example `Utils` class

A higher cohesion codebase will often consist of a greater number of smaller classes/methods with each one being functionally cohesive.

Coupling

Whereas cohesion refers to the relatedness of pieces of code, coupling refers to how strong the connection is between pieces of code. Pieces of code that are strongly coupled, like two pieces of plastic superglued together, are hard to separate, reuse and test independently.

Code that is "superglued" together is referred to as highly coupled, strongly coupled, or tightly coupled. Conversely, non-"superglued" code is referred to using terms like low coupling, loosely coupled, or weakly coupled.

There are many ways that pieces of code can be coupled. The following code shows an example of Global Coupling (also known as Common Coupling).

```
namespace CleanCSharp.CohesionAndCoupling.Dirty
{
        public class ClassA
        {
                public static string SomeSharedData;
        }

        public class ClassB
        {
                public void SomeMethod()
```

```
                {
                        ClassA.SomeSharedData = "xxxx";
                }
        }

        public class ClassC
        {
                public void SomeMethod()
                {
                        var someVariable = ClassA.SomeSharedData;

                        // do something with someVariable
                }
        }
}
```

In the preceding code `ClassB` and `ClassC` are coupled to `ClassA` by accessing the static `SomeSharedData` field. The coupling here occurs in the lines `ClassA.SomeSharedData = "xxxx";` and `var someVariable = ClassA.SomeSharedData;`. The classes `ClassB` and `ClassC` are also harder to test in isolation. In a test `ClassA` will also need to be accessed because it contains the shared data.

Another way that tight coupling can manifest itself is by relying on instantiation of concrete dependencies in code rather than instead relying on abstractions (interfaces, abstract classes) and having the dependency passed into to the class. Thus the consuming class does not control the creation of its dependencies, but something external to the class creates a concrete instance and passes this to the class.

The following code shows the `SendWelcomeEmails` method creating a new concrete `EmailGateway` before using it. The line `var gateway = new EmailGateway();` creates a tight coupling between the `NewCustomerWelcomeEmailSender` class and the `EmailGateway` class.

```
namespace CleanCSharp.CohesionAndCoupling.Dirty
{
        public class EmailGateway
        {
                public void SendEmail(string address, string messageBody)
                {
                        // etc.
                }
        }


        public class NewCustomerWelcomeEmailSender
        {
                public void SendWelcomeEmails()
                {
                        var emailAddresses = GetNewCustomerEmailAddresses();

                        var gateway = new EmailGateway();

                        foreach (var emailAddress in emailAddresses)
                        {
                                gateway.SendEmail(emailAddress, "Welcome!");
                        }
                }

                private IEnumerable<string> GetNewCustomerEmailAddresses()
                {
                        // get emails addresses
                        yield return "some@email430340i0m0imd3.net";
                }
        }
}
```

One way to reduce this coupling is for the `NewCustomerWelcomeEmailSender` class to instead rely on an abstraction that represents an email gateway and for this dependency to be passed to it from the caller.

The following code shows a refactored version with reduced coupling.

```
namespace CleanCSharp.CohesionAndCoupling.Clean
{
        public interface IEmailGateway
        {
                void SendEmail(string address, string messageBody);
        }

        public class EmailGateway : IEmailGateway
        {
                public void SendEmail(string address, string messageBody)
                {
                        // etc.
                }
        }

        public class NewCustomerWelcomeEmailSender
        {
                private readonly IEmailGateway _gateway;

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

                public void SendWelcomeEmails()
                {
                        var emailAddresses = GetNewCustomerEmailAddresses();

                        foreach (var emailAddress in emailAddresses)
                        {
                                _gateway.SendEmail(emailAddress, "Welcome!");
                        }
                }
```

```
                private IEnumerable<string> GetNewCustomerEmailAddresses()
                {
                        // get emails addresses
                        yield return "some@email430340i0m0imd3.net";
                }
        }
}
```

Notice in the preceding code the introduction of the `IEmailGateway` abstraction and the `NewCustomerWelcomeEmailSender` constructor that takes one as a parameter. The `SendWelcomeEmails` method now uses this supplied `IEmailGateway` rather than creating its own.

Clean Tests

Test code should be clean. It should usually be as clean as the production code it tests. In this regard, all of the information given in preceding chapters applies to test code.

There are a number of additional considerations when it comes to clean C# test code, these additional consideration are covered in this chapter.

Qualities of Good Test

There are a number of qualities that clean tests possess.

Execution Speed

When executing unit tests, tests that execute only a small part (perhaps just a single class) of the overall codebase, they should execute very quickly. While there is no single absolute rule for the maximum time, if developers are waiting for 30 minutes to run all the unit tests there may be a problem.

One of the purposes of unit tests is to get quick feedback once changes are made. Whilst a developer may run a subset of tests, as a general rule the entire suite of all unit tests should ideally execute in multiples of seconds rather than minutes. The ideal maximum time will depend on the size and complexity of the code being tested.

Integration tests may rely on communication with out of process resources such as the file system or a database. These types of tests will naturally run slower than unit tests.

Independent and Isolated

Unit tests should be able to be executed in any order without affecting the test results. A test should not rely on another test having previously been executed to set up some state.

Repeatable and Reliable

Tests should pass or fail consistently. If a test passes sometimes and fails sometimes it is not repeatable and therefore not reliable. The results of tests should be able to be relied upon, without having to second guess if they are working properly.

Valuable

Tests should provide some value. There is a cost to both create them and maintain them over time. It seems like an obvious statement, but there is little value in testing auto-implemented C# property setters and getters.

Resilient to Production Code Changes

Changes to production code usually require changes to the test code that runs against it. It is a necessary and expected cost that test code must be changed when the production code it **directly** tests is changed. Wherever possible, test code should insulate itself from unnecessary changes due to unrelated/tangentially-related changes in the production code.

The Three Logical Phases of Tests

A test can be thought of as 3 distinct phase, the Arrange phase, the Act phase, and the Assert phase. Each phase has a specific responsibility as described below.

The Arrange Phase

In the Arrange phase, the starting state of the system under test (SUT) and the test code itself is created. This could mean creating an instance of the SUT class and setting properties to put it into a know starting state. In an integration test, the Arrange phase could mean ensuring that expected files exist on the file system or specific records exist in a database.

The Act Phase

In the Act phase, the SUT is executed or exercised in a specific way to produce some expected result or change in state.

The Assert Phase

The previous Act phase caused some change to happen. In the Assert phase this expected change is tested to ensure it meets the expected outcome. If the expectation meets the actual result then the test will pass, otherwise it will fail.

How Many Asserts?

A test could contain one or more asserts. The strict approach is that a test may only have a single assert statement. While this a good starting point, it may make sense to allow multiple assert statements in some tests.

If a test contains multiple asserts then it is important to ensure that all the asserts relate to testing the same single behaviour/concept. If the asserts relate to differing behaviours then it is a sign that the test may need to be split into multiple tests with more highly related (or single) asserts.

Simplifying Arrange Phase code with AutoFixture

AutoFixture is an open source library that allows anonymous test data values to be generated. This anonymous test data are values that need to be populated for the test to work but where the actual value itself is unimportant.

When combined with a testing framework such as xUnit.net, AutoFixture can greatly reduce (or even eliminate) the Arrange phase.

The following code shows two “production” classes:

```
public class Person
{
        public string Name { get; set; }
}

public class PersonWriter
{
        private readonly Person _person;

        public PersonWriter(Person person)
        {
                _person = person;
        }

        public Person Person
        {
                get { return _person; }
        }

        public string Write()
        {
                return "The name of the person is " + Person.Name;
        }
}
```

To test the `PersonWriter.Write` method the following simple test could be written (using the xUnit.net testing framework):

```
[Fact]
public void WithoutAutoFixture()
{
        // Arrange phase

        var person = new Person
                {
                        Name = "Amrit"
                };

        var sut = new PersonWriter(person);


        // Act phase

        var result = sut.Write();


        // Assert phase

        Assert.Equal("The name of the person is Amrit", result);
}
```

Notice in this preceding test, a `Person` needs to be created (with a name) to be able test the `PersonWriter`.

The test can be refactored to use AutoFixture to create a `Person` for us and automatically set the `Name` property:

```
[Fact]
public void WithAutoFixture()
{
        // Arrange phase

        var fixture = new Fixture();

        // Create a Person with automatically created anon Name
        var person = fixture.Create<Person>();

        var sut = new PersonWriter(person);


        // Act phase

        var result = sut.Write();


        // Assert phase

        Assert.Equal("The name of the person is " + person.Name, result);
}
```

In the preceding test, the actual `Name` of the `Person` is irrelevant, it is now "anonymous".

Combining AutoFixture with xUnit.net theories allow the test to be reduced further as shown in the following test:

```
[Theory]
[AutoData]
public void WithAutoData(PersonWriter sut)
{
        // Arrange phase performed automatically for us


        // Act phase
        var result = sut.Write();


        // Assert phase

        Assert.Equal("The name of the person is " +
                                sut.Person.Name, result);
}
```

To learn more about AutoFixture, see the project site readme document[13] or the Author's Pluralsight course[14].

[13] https://github.com/AutoFixture/AutoFixture/blob/master/README.md

[14] http://bit.ly/psautofixture

Building On Clean Code

There are a number of useful principles that build on top of or compliment the other Clean C# concepts covered in this book. These principles can further improve the overall cleanliness of the codebase.

The Single Responsibility Principle (SRP)

This principle states than any given class should do one well-defined thing, and **only** that one well-defined thing. Put 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 for a class to change, 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. This is a violation of the Single Responsibility Principle. 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.

Some of the benefits of adhering to SRP include:

- Classes only need to be changed when the "thing" that they are responsible for change
- Classes have fewer lines of more highly-related code, making it easier for a developers to understand them
- More individual source code files (one for each class) could result in fewer merge conflicts in larger teams
- Classes may be more easily testable

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 its own class).

The primary benefit of adhering to OCP is the potential reduction in code defects because existing code that is currently in use is not being changed.

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.

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

The Interface Segregation Principle (ISP)

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

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 readable, 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) KISS suggests that the simplest solution should usually be chosen.

Don't Repeat Yourself (DRY)

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

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

You Aren't Gonna Need It (YAGNI)

This principle advocates for the resistance of the temptation to second-guess what code *might* be needed at some future point and going and adding it now ("just in case"), 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.

Violation of the YAGNI principle can result in:

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