Clean Code

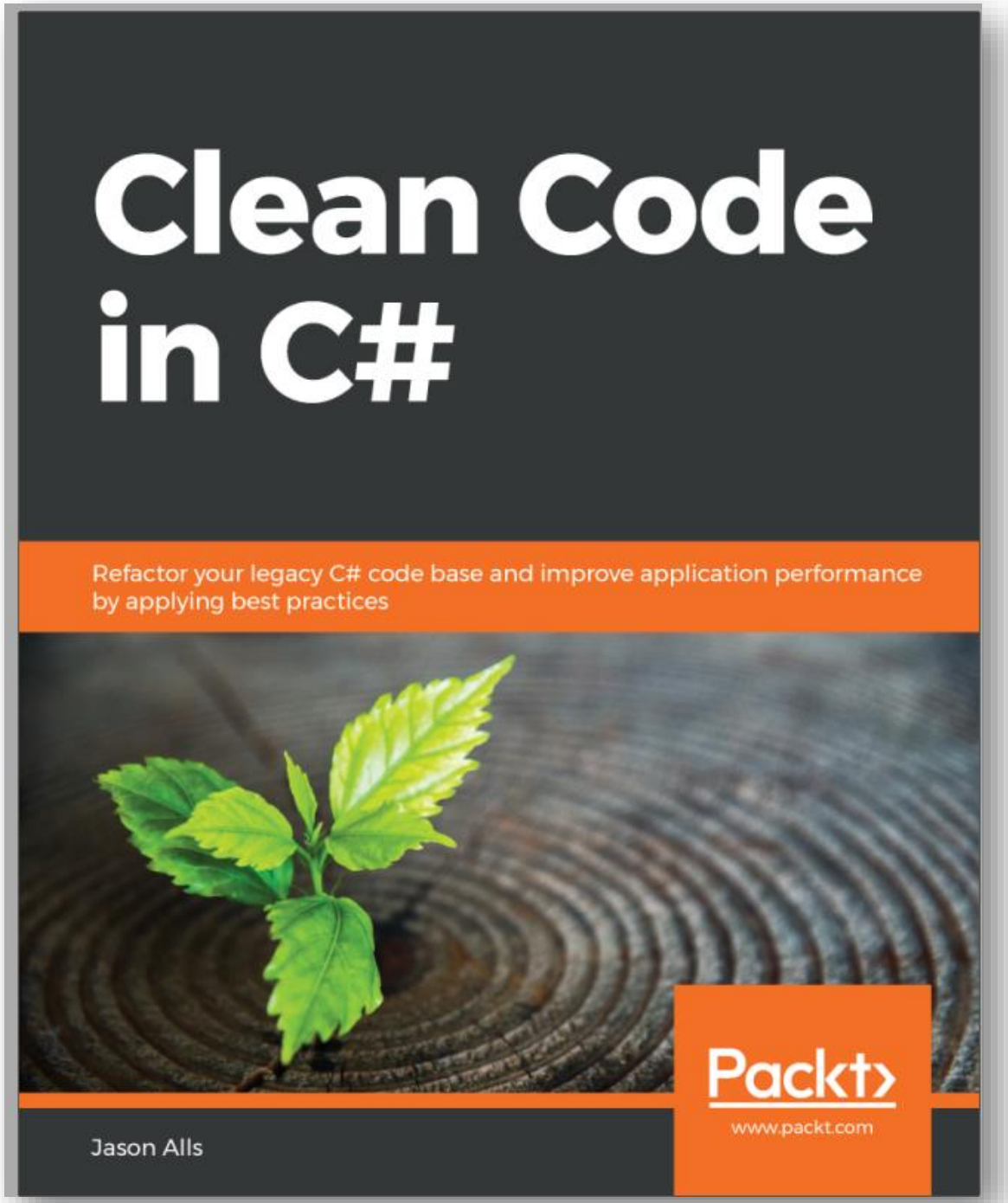
هذه الملاحظات الخاصة في الدورة
لمتابعة الدورة على قناة تكنو يوم
https://www.youtube.com/c/tecnou

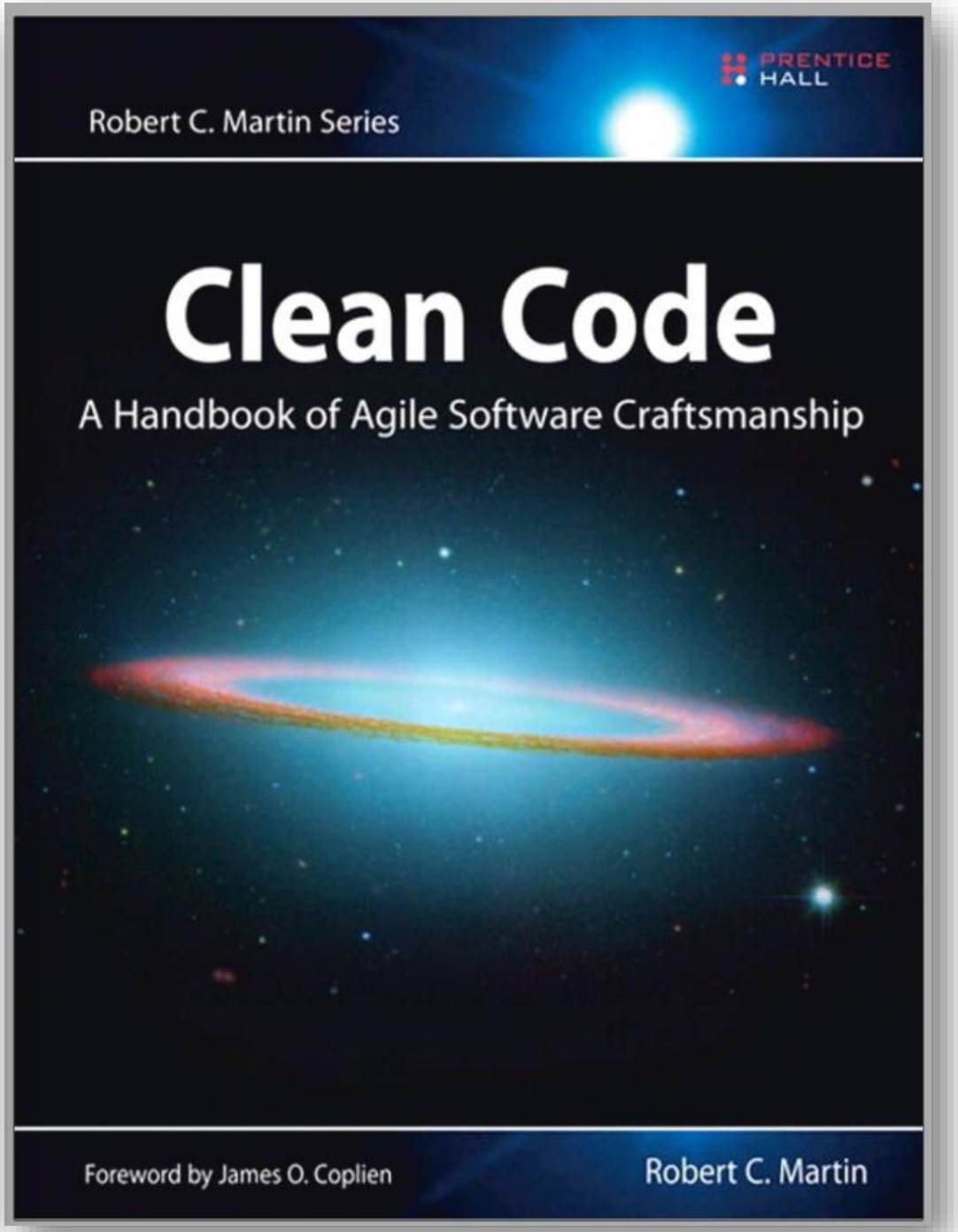
او اضغط هنا للا نتقال للدورة بشكل مباشر

Main Goal

Write Clean Code

The References

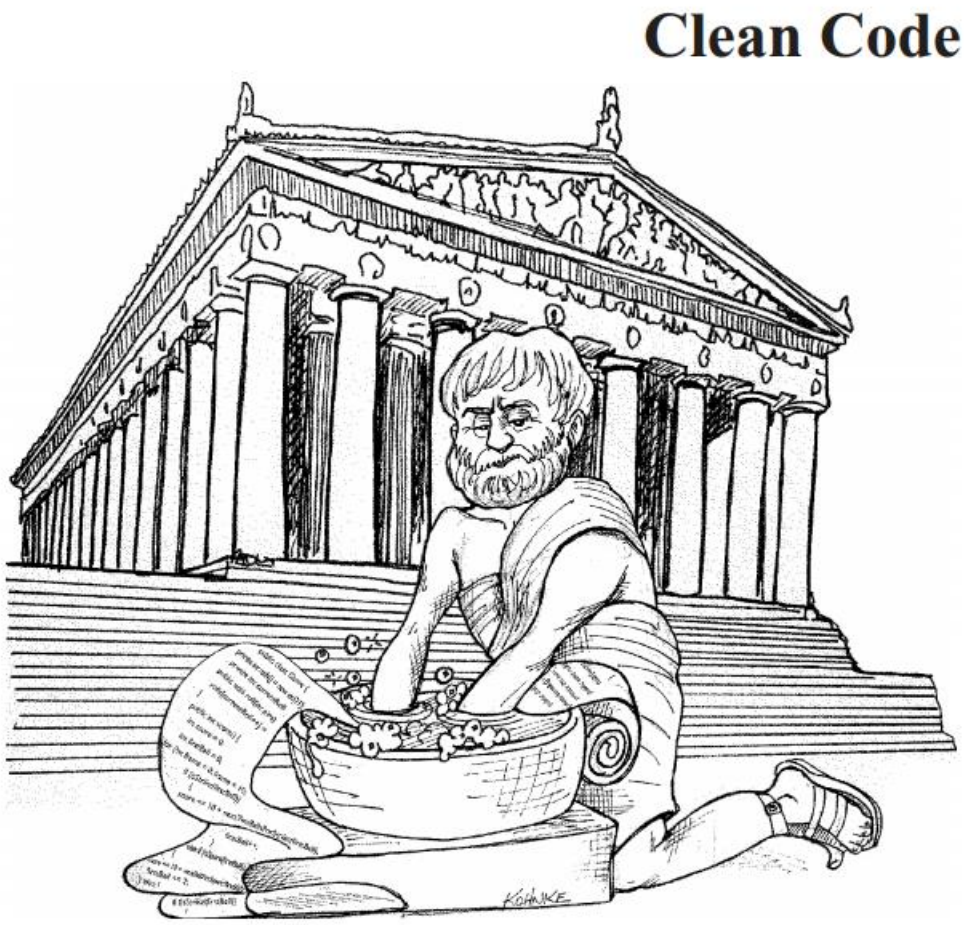




What we learn in this course?

1. Clean code and bad code
2. Code Review – Process and Importance
3. Meaningful Names
4. Functions
5. Comments
6. Formatting
7. Classes, Objects, and Data Structures
8. Writing Clean Functions
9. Exception Handling
10. Unit Testing
11. Threading and Concurrency
12. Using Tools to Improve Code Quality
13. Refactoring Code
14. Implementing Design Patterns

Clean code and bad code



Clean code and bad code

- ✓ Both good code and bad code compile. That's the first thing to understand.
- ✓ The next thing to understand is that bad code is bad for a reason, and likewise, good code is good for a reason.
- ✓ Let's have a look at some of those reasons in the following comparison table:

Improper indentation

```
public void DoSomething()
{
for (var i = 0; i < 1000; i++)
{
var productCode = $"PRC000{i}";
//...implementation
}
}
```

Comments that state the obvious

```
public int _value; // This is used for storing integer values.
```

Comments that excuse bad code

```
...
int value = GetDataValue(); // This sometimes causes a divide by zero
error. Don't know why!
...
```

Commented-out lines of code

```
/* No longer used as has been replaced by DoSomethinElse().
public void DoSomething()
{
    // ...implementation...
}
*/
```

Comments that excuse bad code

```
...
int value = GetDataValue(); // This sometimes causes a divide by zero
error. Don't know why!
...
```

Improper organization of namespaces

```
namespace MyProject.TextFileMonitor
{
    + public class Program { ... }
    + public class DateTime { ... }
    + public class FileMonitorService { ... }
    + public class Cryptography { ... }
}
```

We can see that all classes in the preceding code are under one namespace. Yet, we have the opportunity to add three further namespaces to better organize this code:

- `MyProject.TextFileMonitor.Core`: Core classes that define commonly used members will be placed here, such as our `DateTime` class.
- `MyProject.TextFileMonitor.Services`: All classes that act as a service will be placed in this namespace, such as `FileMonitorService`.
- `MyProject.TextFileMonitor.Security`: All security-related classes will be placed in this namespace, including the `Cryptography` class in our example.

Bad naming conventions

In the days of Visual Basic 6 programming, we used to use Hungarian Notation. I remember using it when I first switched to Visual Basic 1.0. It is no longer necessary to use Hungarian Notation. Plus, it makes your code look ugly. So instead of using names such as `lblName`, `txtName`, or `btnSave`, the modern way is to use `NameLabel`, `NameTextBox`, and `SaveButton`, respectively

The use of cryptic names and names that don't seem to match the intention of the code can make reading code rather difficult. What does **ihridx** mean? It means **Human Resources Index** and is an *integer*. Really! Avoid using names such as `mystring`, `myint`, and `mymethod`. Such names really don't serve a purpose.

Bad naming conventions

Don't use underscores between words in a name either, such as `Bad_Programmer`. This can cause visual stress for developers and can make the code hard to read. Simply remove the underscore.

Don't use the same code convention for variables at the class level and method level. This can make it difficult to establish the scope of a variable. A good convention for variable names is to use camel case for variable names such as `alienSpawn`, and Pascal case for method, class, struct, and interface names such as `EnemySpawnGenerator`.

Following the good variable name convention, you should distinguish between local variables (those contained within a constructor or method), and member variables (those placed at the top of the class outside of constructors and methods) by prefixing the member variables with an underscore. I have used this as a coding convention in the workplace, and it does work really well and programmers do seem to like this convention.

Classes that do multiple jobs

A good class should only do one job. Having a class that connects to a database, gets data, manipulates that data, loads a report, assigns the data to the report, displays the report, saves the report, prints the reports, and exports the report is doing too much. It needs to be refactored into smaller, better-organized classes. All-encompassing classes like this are a pain to read. I personally find them daunting. If you come across classes like this, organize the functionality into regions. Then move the code in those regions into new classes that perform one job.

Let's have a look at an example of a class that is doing multiple things:

Classes that do multiple jobs

```
public class DbAndFileManager
{
#region Database Operations
public void OpenDatabaseConnection() { throw new
NotImplementedException(); }
public void CloseDatabaseConnection() { throw new
NotImplementedException(); }
public int ExecuteSql(string sql) { throw new
NotImplementedException(); }
public SqlDataReader SelectSql(string sql) { throw new
NotImplementedException(); }
public int UpdateSql(string sql) { throw new
NotImplementedException(); }
public int DeleteSql(string sql) { throw new
NotImplementedException(); }
public int InsertSql(string sql) { throw new
NotImplementedException(); }
#endregion
#region File Operations
public string ReadText(string filename) { throw new
NotImplementedException(); }
public void WriteText(string filename, string text) { throw new
NotImplementedException(); }
public byte[] ReadFile(string filename) { throw new
NotImplementedException(); }
public void WriteFile(string filename, byte[] binaryData) { throw new
NotImplementedException(); }
#endregion

}
```

Classes that do multiple jobs

the Single

Responsibility Principle (SRP) is broken

Methods that do many things

```
public string security(string plainText)
{
    try
    {
        byte[] encrypted;
        using (AesManaged aes = new AesManaged())
        {
            ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream())
                using (CryptoStream cs = new CryptoStream(ms, encryptor,
                 CryptoStreamMode.Write))
                {
                    using (StreamWriter sw = new StreamWriter(cs))
                        sw.Write(plainText);
                    encrypted = ms.ToArray();
                }
        }
        Console.WriteLine($"Encrypted data:
         {System.Text.Encoding.UTF8.GetString(encrypted)}");
        using (AesManaged aesm = new AesManaged())
        {
            ICryptoTransform decryptor = aesm.CreateDecryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream(encrypted))
            {
```

Methods that do many things

```
                using (CryptoStream cs = new CryptoStream(ms, decryptor,
                 CryptoStreamMode.Read))
                {
                    using (StreamReader reader = new StreamReader(cs))
                        plainText = reader.ReadToEnd();
                }
            }
        }
        Console.WriteLine($"Decrypted data: {plainText}");
    }
    catch (Exception exp)
    {
        Console.WriteLine(exp.Message);
    }
    Console.ReadKey();
    return plainText;
}
```

Methods that do many things

As you can see in the preceding method, it has 10 lines of code and is hard to read. Plus, it is doing more than one thing. This code can be broken down into two methods that each perform a single task. One method would encrypt a string, and the other method would decrypt the string. This leads us nicely into why methods should have no more than 10 lines of code.

Methods with more than 10 lines of code

Methods with more than two parameters

Using exceptions to control program flow

Exceptions used to control program flow may hide the intention of the code. They can also lead to unexpected and unintended results. The very fact that your code has been programmed to expect one or more exceptions shows your design to be wrong. A typical scenario that is covered in more detail in `Chapter 5`, *Exception Handling*.

A typical scenario is when a business uses **Business Rule Exceptions** (**BREs**). A method will perform an action anticipating that an exception will be thrown. The program flow will be determined by whether the exception is thrown or not. A much better way is to use available language constructs to perform validation checks that return a Boolean value.

Using exceptions to control program flow

The following code shows the use of a BRE to control program flow:

```
public void BreFlowControlExample(BusinessRuleException bre)
{
    switch (bre.Message)
    {
        case "OutOfAcceptableRange":
            DoOutOfAcceptableRangeWork();
            break;
        default:
            DoInAcceptableRangeWork();
            break;
    }
}
```

The method accepts `BusinessRuleException`. Depending upon the message in the exception, `BreFlowControlExample()` either calls the `DoOutOfAcceptableRangeWork()` method or the `DoInAcceptableRangeWork()` method.

Using exceptions to control program flow

A much better way to control the flow is through Boolean logic. Let's look at the following `BetterFlowControlExample()` method:

```
public void BetterFlowControlExample(bool isInAcceptableRange)
{
    if (isInAcceptableRange)
        DoInAcceptableRangeWork();
    else
        DoOutOfAcceptableRangeWork();
}
```

In the `BetterFlowControlExample()` method, a Boolean value is passed into the method. The Boolean value is used to determine which path to execute. If the condition is in the acceptable range, then `DoInAcceptableRangeWork()` is called. Otherwise, the `DoOutOfAcceptableRangeWork()` method is called.

Next, we will consider code that is difficult to read.

Code that is difficult to read

Code that is tightly coupled

```
public class Database
{
    private SqlServerConnection _databaseConnection;

    public Database(SqlServerConnection databaseConnection)
```



```
    {
        _databaseConnection = databaseConnection;
    }
}
```

As you can see from the example, our database class is tied to using SQL Server and would require a hardcoded change to accept any other type of database. We will be covering refactoring of code in later chapters with actual code examples.

Code Low cohesion

Low cohesion consists of unrelated code that performs a variety of different tasks all grouped together. An example would be a utility class that contains a number of different utility methods for handling dates, text, numbers, doing file input and output, data validation, and encryption and decryption.

Objects left hanging around

When objects are left hanging around in memory, they can lead to memory leaks.

Static variables can lead to memory leaks in several ways. If you're not using `DependencyObject` or `INotifyPropertyChanged`, then you are effectively subscribing to events. The **Common Language Runtime** (**CLR**) creates a strong reference by using the `ValueChanged` event via the `PropertyDescriptors AddValueChanged` event, which results in the storage of `PropertyDescriptor` that references the object it is bound to.

Objects left hanging around

Unless you unsubscribe your bindings, you will end up with a memory leak. You will also end up with memory leaks using static variables that reference objects that don't get released. Any object that is referenced by a static variable is marked as not to be collected by the garbage collector. This is because static variables that reference objects are **Garbage Collection** (**GC**) roots, and anything that is a GC root is marked by the garbage collector as *do not collect*.

When you use anonymous methods that capture class members, the instance of the class is referenced. This causes a reference to the class instance to remain alive while the anonymous methods stay alive.

When using **unmanaged code** (**COM**), if you do not release any managed and unmanaged objects and explicitly deallocate any memory, then you will end up with memory leaks.

Use of the Finalize() method

While finalizers can help free up resources from objects that have not been correctly disposed of and help to prevent memory leaks, they do have a number of drawbacks.

You do not know when finalizers will be called. They will be promoted by the garbage collector along with all dependants on the graph to the next generation, and will not be garbage-collected until the garbage collector decides to do so. This can mean that objects stay in memory for a long time. Out-of-memory exceptions could occur using finalizers as you can be creating objects faster then they are getting garbage-collected.

Over-engineering

Over-engineering can be an utter nightmare. The biggest reason for this is that as a mere human, wading through a massive system, trying to understand it, how you are to use it, and what goes where is a time-consuming process. All the more so when there is no documentation, you are new to the system, and even people who have been using it much longer than you are unable to answer your questions.

This can be a major cause of stress when you are expected to work on it with set deadlines.

Lack of regions in large classes

Large classes with lots of regions are very hard to read and follow, especially when related methods are not grouped together. Regions are very good for grouping similar to members within a large class. But they are no good if you don't use them!

Lost-intention code

If you are viewing a class and it is doing several things, then how do you know what its original intention was? If you are looking for a date method, for example, and you find it in a file class in the input/output namespace of your code, is the date method in the right location? No. Will it be hard for other developers who don't know your code to find that method? Of course it will. Take a look at this code:

```
public class MyClass
{
    public void MyMethod()
    {
        // ...implementation...
    }

    public DateTime AddDates(DateTime date1, DateTime date2)
```

Directly exposing information

Classes that directly expose information are bad. Apart from producing tight coupling that can lead to bugs, if you want to change the information type, you have to change the type everywhere it is used. Also, what if you want to perform data validation before the assignment? Here's an example:

```
public class Product
{
    public int Id;
    public int Name;
    public int Description;
    public string ProductCode;
    public decimal Price;
    public long UnitsInStock
}
```

In the preceding code, if you wanted to change `UnitsInStock` from type `long` to type `int`, you would have to change the code *everywhere* it is referenced. You would have to do the same with `ProductCode`. If new product codes had to adhere to a strict format, you would not be able to validate product codes if the string could be directly assigned by the calling class.

Directly exposing information

Classes that directly expose information are bad. Apart from producing tight coupling that can lead to bugs, if you want to change the information type, you have to change the type everywhere it is used. Also, what if you want to perform data validation before the assignment? Here's an example:

```
public class Product
{
    public int Id;
    public int Name;
    public int Description;
    public string ProductCode;
    public decimal Price;
    public long UnitsInStock
}
```

In the preceding code, if you wanted to change `UnitsInStock` from type `long` to type `int`, you would have to change the code *everywhere* it is referenced. You would have to do the same with `ProductCode`. If new product codes had to adhere to a strict format, you would not be able to validate product codes if the string could be directly assigned by the calling class.

Clean Code

Proper indentation

When you use proper indentation, it makes reading the code much easier. You can tell by the indentation where code blocks start and end, and what code belongs to those code blocks:

```
public void DoSomething()
{
    for (var i = 0; i < 1000; i++)
    {
        var productCode = $"PRC000{i}";
        //...implementation
    }
}
```

In the preceding simple example, the code looks nice and is readable. You can clearly see where each code block starts and finishes.

Meaningful comments

Meaningful comments are comments that express the programmer's intention. Such comments are useful when the code is correct but may not be easily understood by anyone new to the code, or even to the same programmer in a few week's time. Such comments can be really helpful.

API documentation comments

A good API is an API that has good documentation that is easy to follow. API comments are XML comments that can be used to generate HTML documentation. HTML documentation is important for developers wanting to use your API. The better the documentation, the more developers are likely to want to use your API. Here's an example:

```
/// <summary>
/// Create a new <see cref="KustoCode"/> instance from the text and
globals. Does not perform
/// semantic analysis.
/// </summary>
/// <param name="text">The code text</param>
/// <param name="globals">
///   The globals to use for parsing and semantic analysis. Defaults to
<see cref="GlobalState.Default"/>
/// </param>.
 public static KustoCode Parse(string text, GlobalState globals = null) {
... }
```

Proper organization using namespaces

Code that is properly organized and placed in appropriate namespaces can save developers a good amount of time when looking for a particular piece of code. For instance, if you are looking for classes and methods to do with dates and times, it would be a good idea to have a namespace called `DateTime`, a class called `Time` for time-related methods, and a class called `Date` for date-related methods.

The following is an example of the proper organization of namespaces:

Name	Description
`CompanyName.IO.FileSystem`	The namespace contains classes that define file and directory operations.
`CompanyName.Converters`	The namespace contains classes for performing various conversion operations.
`CompanyName.IO.Streams`	The namespace contains types for managing stream input and output.

Good naming conventions

It is good to follow the Microsoft C# naming conventions. Use Pascal casing for namespaces, classes, interfaces, enums, and methods. Use camel case for variable names and argument names, and make sure to prefix member variables with an underscore.

Have a look at this example code:

```
using System;
using System.Text.RegularExpressions;

namespace CompanyName.ProductName.RegEx
{
  /// <summary>
  /// An extension class for providing regular expression extensions
  /// methods.
  /// </summary>
  public static class RegularExpressions
  {
    private static string _preprocessed;

    public static string RegularExpression { get; set; }
    public static bool IsValidEmail(this string email)
```

Good naming conventions

```
    {
      // Email address: RFC 2822 Format.
      // Matches a normal email address. Does not check the
      // top-level domain.
      // Requires the "case insensitive" option to be ON.
      var exp = @"\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.
       [a-z0-9!#$%&'*+/=?^_`{|}~-]+)@(?:[a-z0-9](?:[a-z0-9-]
       [a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?)\Z";
      bool isEmail = Regex.IsMatch(email, exp, RegexOptions.IgnoreCase);
      return isEmail;
    }

    // ... rest of the implementation ...

  }
}
```

snake_case

Pros: Concise when it consists of a few words.

Cons: Redundant as hell when it gets longer.

```
push_something_to_first_queue, pop_what, get_whatever…
```

PascalCase

Pros: Seems neat.

```
GetItem, SetItem, Convert, ...
```

Cons: Barely used. (why?)

camelCase

Pros: Widely used in the programmer community.

Cons: Looks ugly when a few methods are n-worded.

```
push, reserve, beginBuilding, ...
```

Classes that only do one job

A good class is a class that does only one job. When you read the class, its intention is clear. Only the code that should be in that class is in that class and nothing else.

Methods that do one thing

Methods should only do one thing. You should not have a method that does more than one thing, such as decrypting a string and performing string replacement. A method's intent should be clear. Methods that do only one thing are more inclined to be small, readable, and intentional.

Methods with less than 10 lines, and preferably no more than 4

Ideally, you should have methods that are no longer than 4 lines of code. However, this is not always possible, so you should aim to have methods that are no more than 10 lines in length so that they are easy to read and maintain.

Methods with no more than two parameters

It is best to have methods with no parameters, but having one or two is okay. If you start having more than two parameters, you need to think about the responsibility of your class and methods: are they taking on too much? If you do need more than two parameters, then you are better placed to pass an object.

Any method with more than two parameters can become difficult to read and follow. Having no more than two parameters makes the code readable, and a single parameter that is an object is way more readable than a method with several parameters.

Proper use of exceptions

Never use exceptions to control program flow. Handle common conditions that might trigger exceptions in such a way that an exception will not be raised or thrown. A good class is designed in such a way that you can avoid exceptions.

Recover from exceptions and/or release resources by using `try/catch/finally` exceptions. When catching exceptions, use specific exceptions that may be thrown in your code, so that you have more detailed information to log or assist in handling the exception.

Sometimes, using the predefined .NET exception types is not always possible. In such cases, it will be necessary to produce your own custom exceptions. Suffix your custom exception classes with the word `Exception`, and make sure to include the following three constructors:

Proper use of exceptions

- `Exception()`: Uses default values
- `Exception(string)`: Accepts a string message
- `Exception(string, exception)`: Accepts a string message and an inner exception

If you have to throw exceptions, don't return error codes but return exceptions with meaningful information.

Code that is readable

The more readable the code is, the more developers will enjoy working with it. Such code is easier to learn and work with. As developers come and go on a project, newbies will be able to read, extend, and maintain the code with little effort. Readable code is also less inclined to be buggy and unsafe.

Code that is loosely coupled

Loosely coupled code is easier to test and refactor. You can also swap and change loosely coupled code more easily if you need to. Code reuse is another benefit of loosely coupled code.

Let's use our bad example of a database being passed a SQL Server connection. We could make that same class loosely coupled by referencing an interface instead of a concrete type. Let's have a look at a good example of the refactored bad example from earlier:

```
public class Database
{
    private IDatabaseConnection _databaseConnection;

    public Database(IDatabaseConnection databaseConnection)
    {
        _databaseConnection = datbaseConnection;
    }
}
```

High cohesion

Common functionality that is correctly grouped together is known to be highly cohesive. Such code is easy to find. For example, if you look at the `Microsoft.Diagnostics` namespace, you will find that it only contains code that pertains to diagnostics. It would not make sense to include collections and filesystem code in the `Diagnostics` namespace.

Objects are cleanly disposed of

When using disposable classes, you should always call the `Dispose()` method to cleanly dispose of any resources that are in use. This helps to negate the possibility of memory leaks.

Objects are cleanly disposed of

There are times when you may need to set an object to `null` for it to go out of scope. An example would be a static variable that holds a reference to an object that you no longer require.

The `using` statement is also a good clean way to use disposable objects, as when the object is no longer in scope it is automatically disposed of, so you don't need to explicitly call the `Dispose()` method. Let's have a look at the code that follows:

```
using (var unitOfWork = new UnitOfWork())
{
 // Perform unit of work here.
}
// At this point the unit of work object has been disposed of.
```

The code defines a disposable object in the `using` statement and does what it needs to between the opening and closing curly braces. The object is automatically disposed of before the braces are exited. And so there is no need to manually call the `Dispose()` method, because it is called automatically.

Avoiding the Finalize() method

When using unmanaged resources, it is best to implement the `IDisposable` interface and avoid using the `Finalize()` method. There is no guarantee of when finalizers will run. They may not always run in the order you expect or when you expect them to run. Instead, it is better and more reliable to dispose of unmanaged resources in the `Dispose()` method.

The right level of abstraction the Finalize() method

You have the right level of abstraction when you expose to the higher level only that which needs exposure, and you do not get lost in the implementation.

If you find that you are getting lost in the implementation details, then you have over-abstracted. If you find that multiple people have to work in the same class at the same time, then you have under-abstracted. In both cases, refactoring would be needed to get the abstraction to the right level.

Using regions in large classes

Regions are very useful for grouping items within a large class as they can be collapsed. It can be quite daunting reading through a large class and having to jump back and forth between methods, so grouping methods that call each other in the class is a good way to group them. The methods can then be collapsed and expanded as needed when working on a piece of code.

Good Code	Bad Code
Proper indentation.	Improper indentation.
Meaningful comments.	Comments that state the obvious.
API documentation comments.	Comments that excuse bad code. Commented out lines of code.
Proper organization using namespaces.	An improper organization using namespaces.
Good naming conventions.	Bad naming conventions.
Classes that do one job.	Classes that do multiple jobs.
Methods that do one thing.	Methods that do many things.
Methods with less than 10 lines, and preferably no more than 4.	Methods with more than 10 lines of code.
Methods with no more than two parameters.	Methods with more than two parameters.
Proper use of exceptions.	Using exceptions to control program flow.
Code that is readable.	Code that is difficult to read.
Code that is loosely coupled.	Code that is tightly coupled.
High cohesion.	Low cohesion.
Objects are cleanly disposed of.	Objects left hanging around.
Avoidance of the Finalize() method.	Use of the Finalize() method.
The right level of abstraction.	Over-engineering.
Use of regions in large classes.	Lack of regions in large classes.
Encapsulation and information hiding.	Directly exposing information.
Object-oriented code.	Spaghetti code.
Design patterns.	Design anti-patterns.

Coding standards

Coding standards

Coding standards set out several dos and don'ts that must be adhered to. Such standards can be enforced through tools such as FxCop and manually via peer code reviews. All companies have their own coding standards that must be adhered to. But what you will find in the real world is that when the business expects a deadline to be met, those coding standards can go out of the window as the deadline can become more important than the actual code quality. This is usually rectified by adding any required refactoring to the bug list as technical debt to be addressed after the release.

Coding standards

https://www.dofactory.com/csharp-coding-standards

Coding principles

Coding principles are a set of guidelines for writing high-quality code, testing and debugging that code, and performing maintenance on the code. Principles can be different between programmers and programming teams.

Coding methodologies

Coding methodologies break down the process of developing software into a number of predefined phases. Each phase will have a number of steps associated with it. Different developers and development teams will have their own coding methodologies that they follow. The main aim of coding methodologies is to streamline the process from the initial concept, through the coding phase, to the deployment and maintenance phases.

Coding conventions

It is best to implement the Microsoft C# coding conventions. You can review them at `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions`.

By adopting Microsoft's coding conventions, you are guaranteed to write code in a formally accepted and agreed-upon format. These C# coding conventions help people to focus on reading your code and spend less time focusing on the layout. Basically, Microsoft's coding standards promote best practices.

https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Modularity

Breaking large programs up into smaller modules makes a lot of sense. Small modules are easy to test, are more readily reused, and can be worked on independently from other modules. Small modules are also easier to extend and maintain.

A modular program can be divided into different assemblies and different namespaces within those assemblies. Modular programs are also much easier to work on in team environments as different modules can be worked on by different teams.

In the same project, code is modularized by adding folders that reflect namespaces. A namespace must only contain code that is related to its name. So, for instance, if you have a namespace called `FileSystem`, then types related to files and directories should be placed in that folder. Likewise, if you have a namespace called `Data`, then only types related to data and data sources should be located in that namespace.

Modularity

In the same project, code is modularized by adding folders that reflect namespaces. A namespace must only contain code that is related to its name. So, for instance, if you have a namespace called `FileSystem`, then types related to files and directories should be placed in that folder. Likewise, if you have a namespace called `Data`, then only types related to data and data sources should be located in that namespace.

Another beautiful aspect of correct modularization is that if you keep modules small and simple, they are easy to read. Most of a coder's life apart from coding is spent reading and understanding code. So the smaller and more correctly modularized the code is, then the more easier it is to read and understand the code. This leads to a greater understanding of the code and improves developer take-up and use of the code.

KISS

You may be the super genius of the computer programming world. You may be able to produce code that is so sexy that other programmers can only stare at it in awe and end up drooling on their keyboard. But do those other programmers know what the code is by just looking at it? If you found that code in 10 weeks' time when you head deep into a mountain of different code with deadlines to meet, would you be able to explain with absolute clarity what your code does and the rationale behind your choice of coding method? And have you considered that you may have to work on that code further down the road?

Have you ever programmed some code, gone away, and then looked at it more than a few days later and thought to yourself, *I didn't write this rubbish, did I? What was I thinking!?* I know I've been guilty of it and so have some of my ex-colleagues.

KISS

When programming code, it is essential to keep the code simple and in a human-readable format that even newbie junior programmers can understand. Often juniors are exposed to code to read, understand, and then maintain. The more complex the code, the longer it takes for juniors to get up to speed. Even seniors can struggle with complex systems to the point that they leave to find work elsewhere that's less taxing on the brain and their well-being.

For example, if you are working on a simple website, ask yourself a few questions. Does it really need to use microservices? Is the brownfield project you are working on really complicated? Is it possible to simplify it to make it easier to maintain? When developing a new system, what are the minimum number of moving parts you need to write a robust, maintainable, and scalable solution that performs well?

YAGNI

YAGNI is a discipline in the agile world of programming that stipulates that a programmer should not add any code until it is absolutely needed. An honest programmer will write failing tests based on a design, then write just enough production code for the tests to work, and finally, refactor the code to remove any duplication. Using the YAGNI software development methodology, you keep your classes, methods, and overall lines of code to an absolute minimum.

The primary goal of YAGNI is to prevent the over-engineering of software systems by computer programmers. Do not add complexity if it is not needed. You must remember to only write the code that you need. Don't write code that you don't need, and don't write code for the sake of experimentation and learning. Keep experimental and learning code in sandboxed projects specifically for those purposes.

DRY

I said *Don't Repeat Yourself!* If you find that you are writing the same code in multiple areas, then this is a definite candidate for refactoring. You should look at the code to see if it can be genericized and placed in a helper class for use throughout the system or in a library for use by other projects.

If you have the same piece of code in multiple locations, and you find the code has a fault and needs to be modified, you must then modify the code in other areas. In situations like this, it is very easy to overlook code that requires modification. The result is code that gets released with the problem fixed in some areas, but still existing in others.

That is why it is a good idea to remove duplicate code as soon as you encounter it, as it may cause more problems further down the road if you don't.

SOLID

- **Single Responsibility Principle**: Classes and methods should only perform a single responsibility. All the elements that form a single responsibility should be grouped together and encapsulated.
- **Open/Closed Principle**: Classes and methods should be open for extension and closed for modification. When a change to the software is required, you should be able to extend the software without modifying any of the code.
- **Liskov Substitution**: Your function has a pointer to a base class. It must be able to use any class derived from the base class without knowing it.
- **Interface Segregation Principle**: When you have large interfaces, the clients that use them may not need all the methods. So, using the **Interface Segregation Principle** (**ISP**), you extract out methods to different interfaces. This means that instead of having one big interface, you have many small interfaces. Classes can then implement interfaces with only the methods they need.
- **Dependency Inversion Principle**: When you have a high-level module, it should not be dependent upon any low-level modules. You should be able to switch between low-level modules freely without affecting the high-level module that uses them. Both high-level and low-level modules should depend upon abstractions.

SOLID

- **Single Responsibility Principle**: Classes and methods should only perform a single responsibility. All the elements that form a single responsibility should be grouped together and encapsulated.
- **Open/Closed Principle**: Classes and methods should be open for extension and closed for modification. When a change to the software is required, you should be able to extend the software without modifying any of the code.
- **Liskov Substitution**: Your function has a pointer to a base class. It must be able to use any class derived from the base class without knowing it.
- **Interface Segregation Principle**: When you have large interfaces, the clients that use them may not need all the methods. So, using the **Interface Segregation Principle** (**ISP**), you extract out methods to different interfaces. This means that instead of having one big interface, you have many small interfaces. Classes can then implement interfaces with only the methods they need.
- **Dependency Inversion Principle**: When you have a high-level module, it should not be dependent upon any low-level modules. You should be able to switch between low-level modules freely without affecting the high-level module that uses them. Both high-level and low-level modules should depend upon abstractions.

Summary

In this chapter, you have had an introduction to good code and bad code and, hopefully, you now understand why good code matters. You have also been provided with the link to the Microsoft C# coding conventions so that you can follow Microsoft best practices for coding (if you are not already doing so).

You have also briefly been introduced to various software methodologies including DRY, KISS, SOLID, YAGNI, and Occam's Razor.

Questions

1. What are some of the outcomes of bad code?
2. What are some of the outcomes of good code?
3. What are some of the benefits of writing modular code?
4. What is DRY code?
5. Why should you KISS when writing code?
6. What does the acronym SOLID stand for?
7. Explain YAGNI.

Code Review

The primary motivation behind any code review is to improve the overall quality of the code. Code quality is very important. This almost goes without saying, especially if your code is part of a team project or is accessible to others, such as open source developers and customers through escrow agreements.

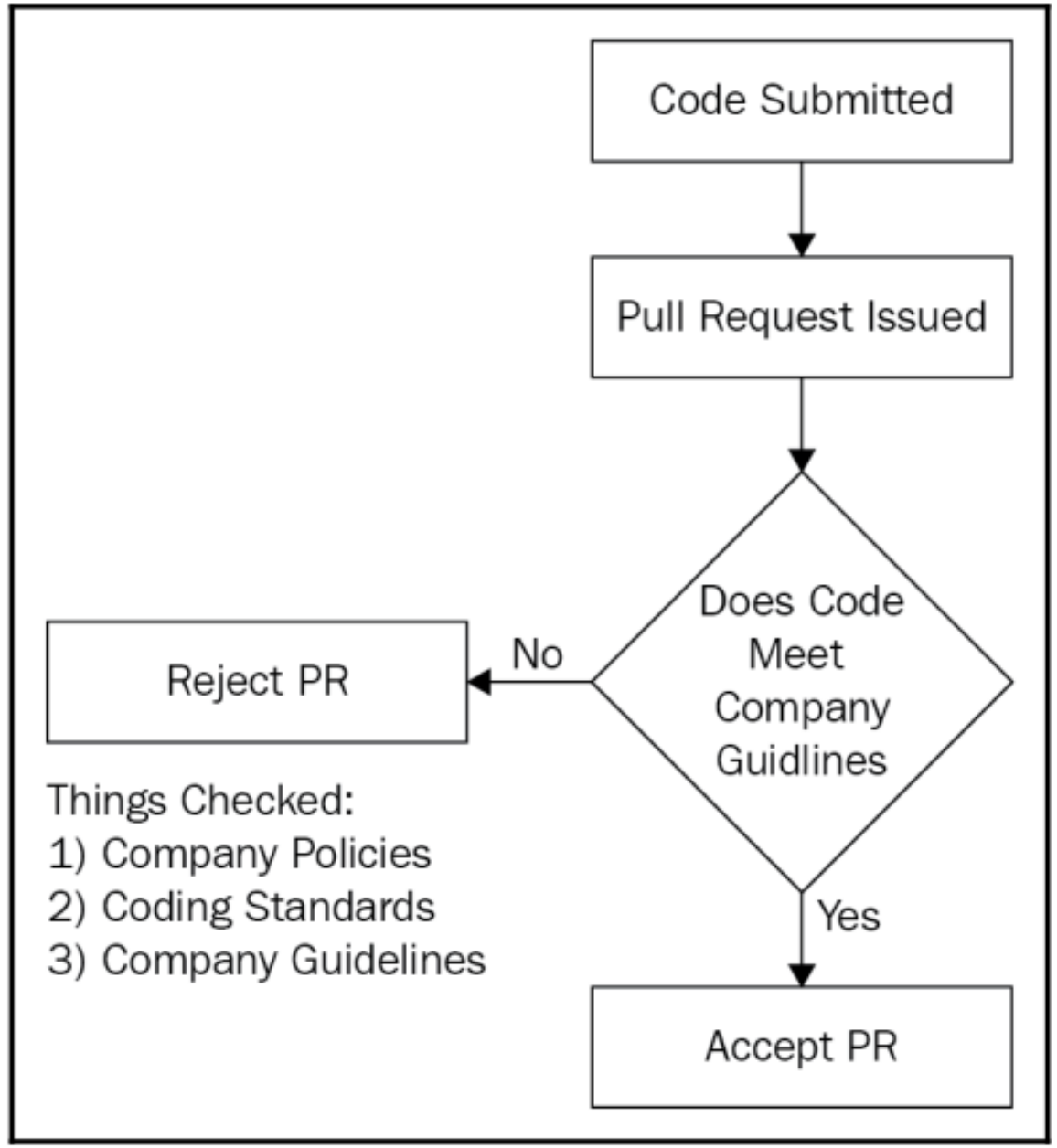
If every developer was free to code as they pleased, you would end up with the same kind of code written in so many different ways, and ultimately the code would become an unwieldy mess. That is why it is important to have a coding standards policy that outlines the company's coding practices and code review procedures that are to be followed.

- Preparing code for review
- Leading a code review
- Knowing what to review
- Knowing when to send code for review
- Providing and responding to review feedback

The code review process

The normal procedure for carrying out a code review is to make sure your code compiles and meets the requirements set. It should also pass all unit tests and end-to-end tests. Once you are confident that you are able to compile, test, and run your code successfully, then it is checked in to the current working branch. Once checked in, you will then issue a pull request.

A peer reviewer will then review your code and share comments and feedback. If your code passes the code review, your code review is completed and you can then merge your working branch into the main trunk. Otherwise, the peer review will be rejected, and you will be required to review your work and address the issues raised in the comments provided by your reviewer.


Code Submitted
Pull Request Issued
Does Code Meet Company Guidlines
No
Reject PR
Yes
Accept PR
Things Checked:
1) Company Policies
2) Coding Standards
3) Company Guidelines

Preparing code for review

Preparing for a code review can be a right royal pain at times, but it does work for better overall quality of code that is easy to read and maintain. It is definitely a worthwhile practice that teams of developers should carry out as standard coding procedures. This is an important step in the code review process, as perfecting this step can save the reviewer considerable time and energy in performing the review.

Here are some standard points to keep in mind when preparing your code for review:

- **Always keep the code review in mind**: When beginning any programming, you should have the code review in mind. So keep your code small. If possible, limit your code to one feature.
- **Make sure that all your tests pass even if your code builds**: If your code builds but you have failing tests, then deal immediately with what's causing those tests to fail. Then, when the tests pass as expected, you can move on. It is important to make sure that all unit tests are passed, and that end-to-end testing passes all tests. It is important that all testing is complete and gets the green light, since releasing code that works but was a test fail could result in some very unhappy customers when the code goes to production.

- **Remember YAGNI**: As you code, make sure to only add code that is necessary to meet the requirement or feature you are working on. If you don't need it yet, then don't code it. Only add code when it is needed and not before.
- **Check for duplicate code**: If your code must be object-oriented and be DRY and SOLID, then review your own code to see whether it contains any procedural or duplicate code. Should it do so, take the time to refactor it so that it is object-oriented, DRY, and SOLID.
- **Use static analyzers**: Static code analyzers that have been configured to enforce your company's best practices will check your code and highlight any issues that are encountered. Make sure that you do not ignore information and warnings. These could cause you issues further down the line.

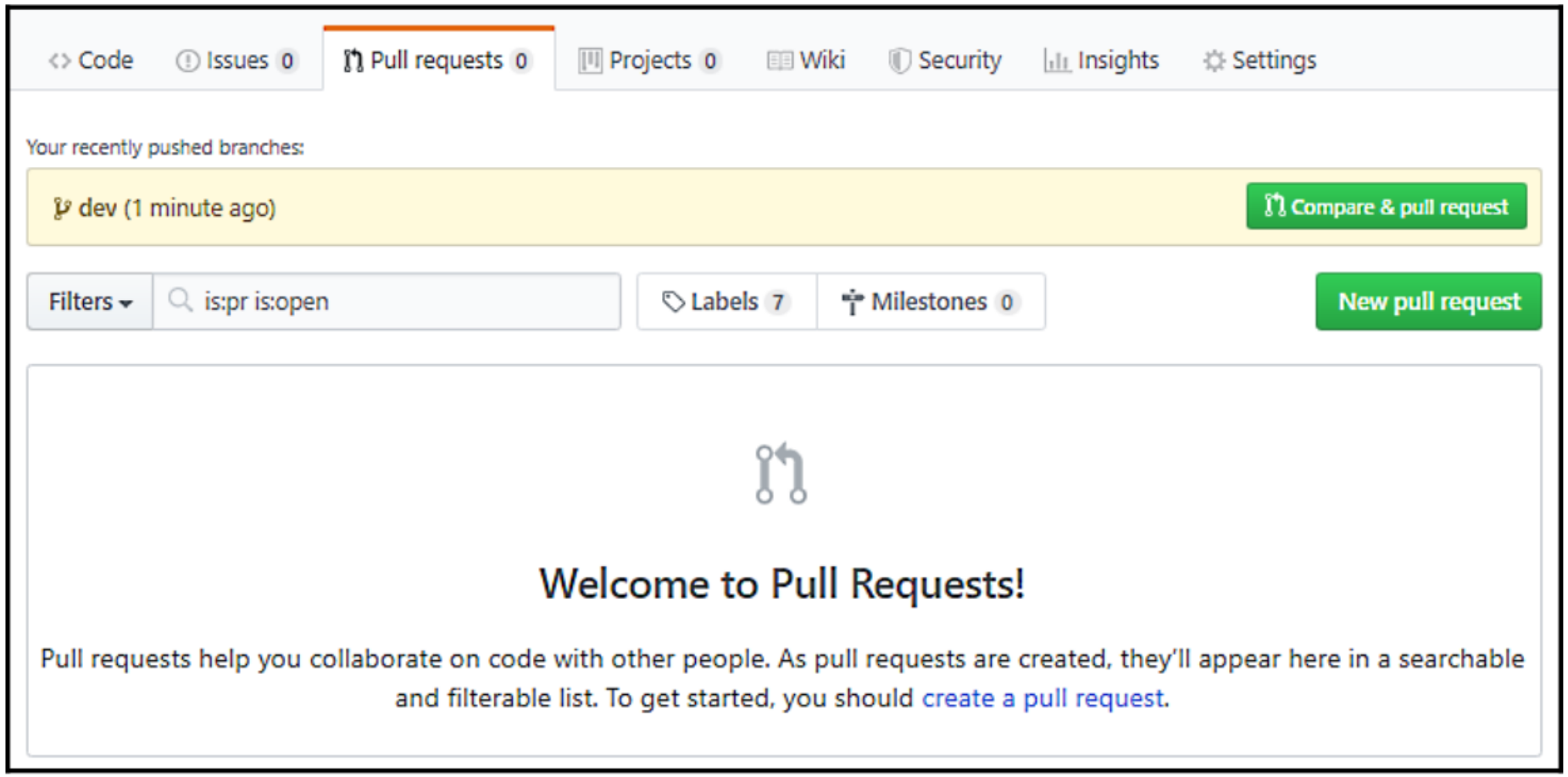
Issuing a pull request

When you have finished coding and you are confident in the quality of your code and that it builds, you are able to then push or check in your changes, depending on what source control system you use. When your code has been pushed, you can then issue a pull request. When a pull request is raised, other people that are interested in the code are notified and able to review your changes. These changes can then be discussed and comments made regarding any potential changes that you need to make. In essence, your pushing to your source control repository and issuing a pull request is what kick-starts the peer code review process.

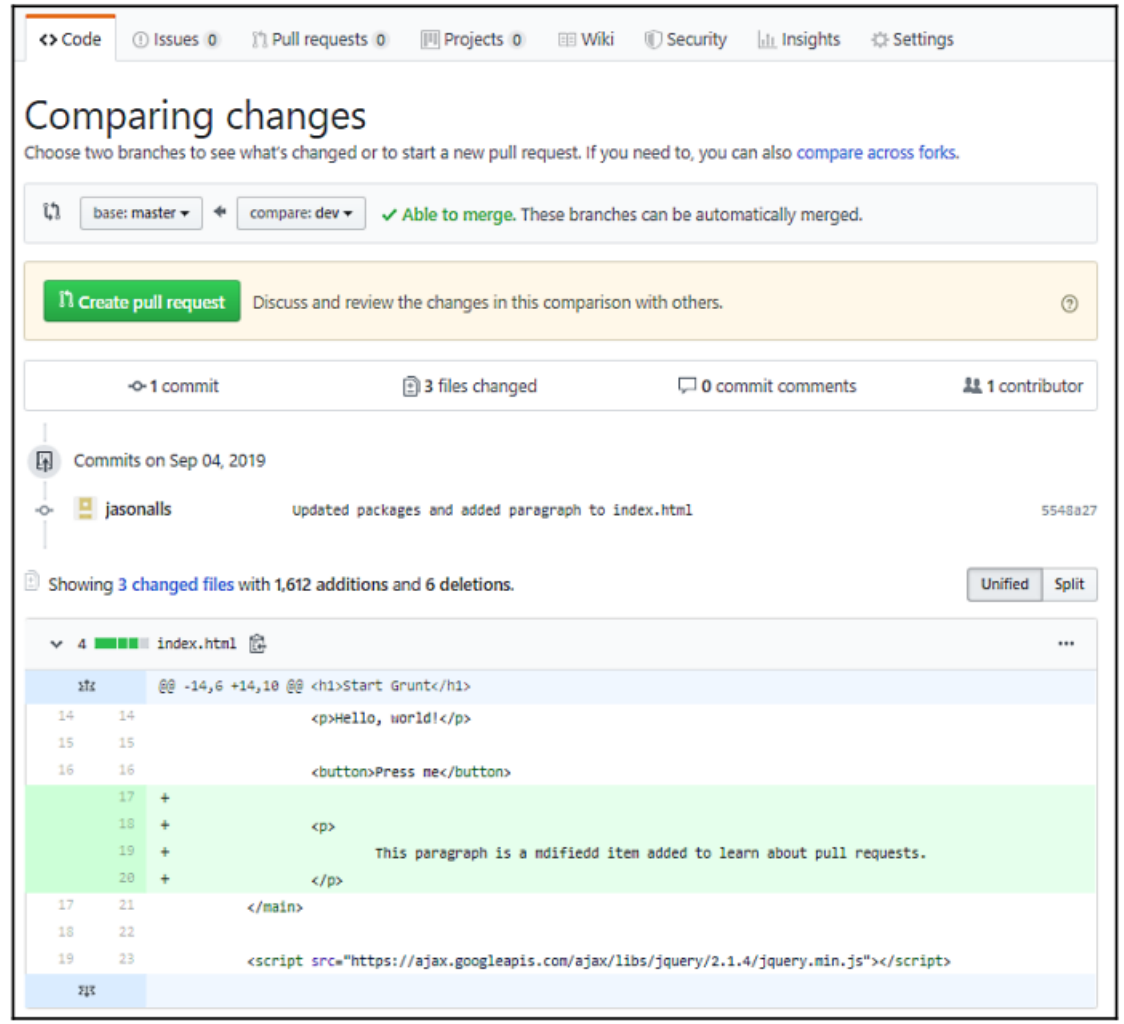
To issue a pull request, all you have to do (once you've checked your code in or pushed it) is click on the **Pull requests** tab of your version control. There will then be a button you can click on – **New pull request**. This will add your pull request to a queue to be picked up by the relevant reviewers.

In the following screenshots, we will see the process of requesting and fulfilling a pull request via GitHub:

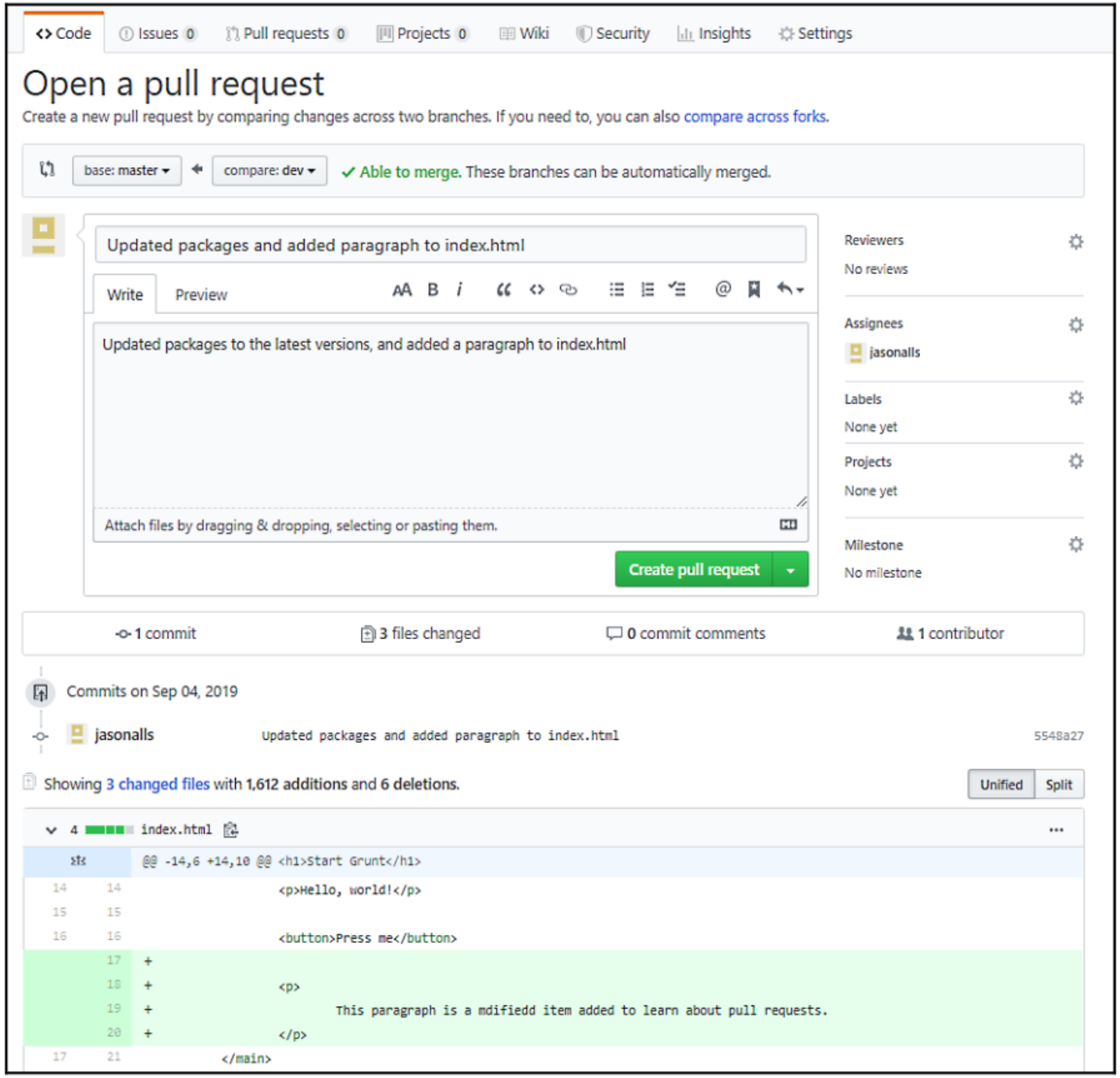
1. On your GitHub project page, click on the **Pull requests** tab:



2. Then, click on the **New pull request** button. This will display the **Comparing changes** page:



3. If you are happy, then click on the **Create pull request** button to start the pull request. You will then be presented with the **Open a pull request** screen:

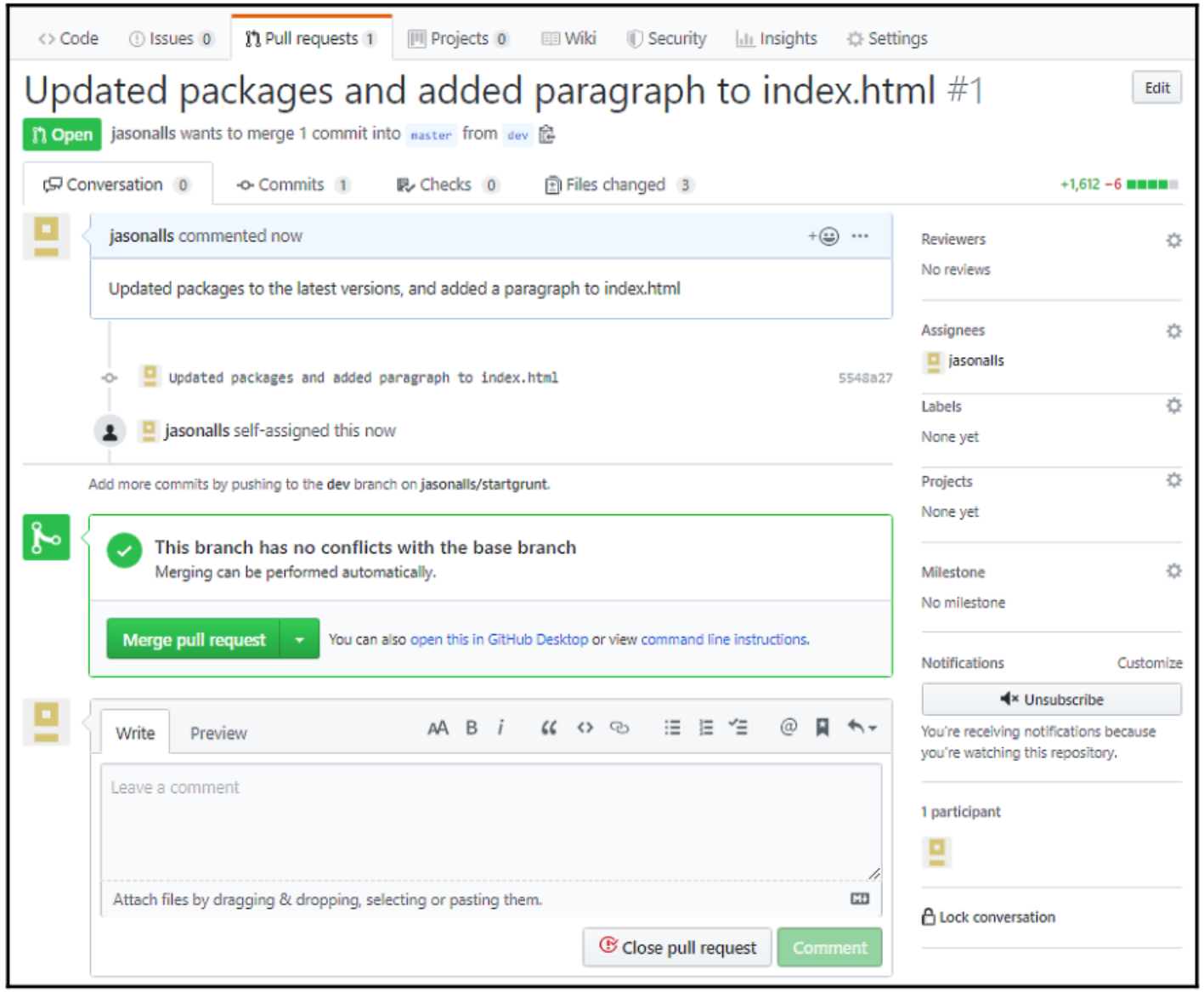

Code
Issues 0
Pull requests 0
Projects 0
Wiki
Security
Insights
Settings
Open a pull request
Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.
base: master
compare: dev
Able to merge. These branches can be automatically merged.
Updated packages and added paragraph to index.html
Write
Preview
Updated packages to the latest versions, and added a paragraph to index.html
Attach files by dragging & dropping, selecting or pasting them.
Create pull request
Reviewers
No reviews
Assignees
jasonalls
Labels
None yet
Projects
None yet
Milestone
No milestone
1 commit
3 files changed
0 commit comments
1 contributor
Commits on Sep 04, 2019
jasonalls
Updated packages and added paragraph to index.html
5548a27
Showing 3 changed files with 1,612 additions and 6 deletions.
Unified
Split
4
index.html
@@ -14,6 +14,10 @@ <h1>Start Grunt</h1>
Hello, world!
<button>Press me</button>
+
+
+ This paragraph is a mdifiedd item added to learn about pull requests.
+
</main>

4. Write your comment regarding the pull request. Provide all the necessary information for the code reviewer, but keep it brief and to the point. Useful comments include identification of what changes have been made. Modify the **Reviewers**, **Assignees**, **Labels**, **Projects**, and **Milestones** fields as necessary. Then, once you are happy with the pull request details, click on the **Create pull request** button to create the pull request. Your code will now be ready to be reviewed by your peers.

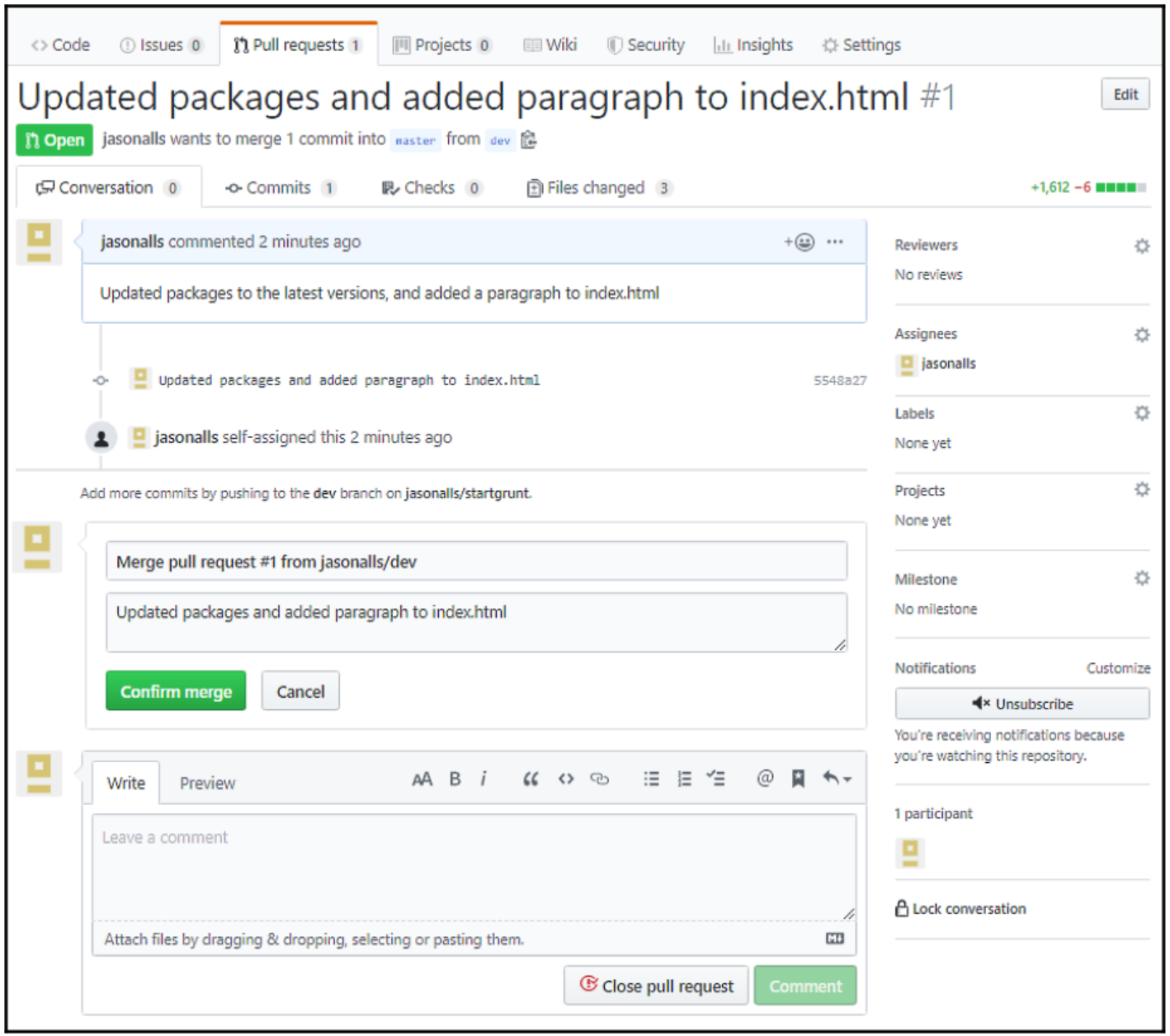
Responding to a pull request

Since the reviewer is responsible for reviewing pull requests prior to merges of branches, we would do well to look at responding to pull requests:

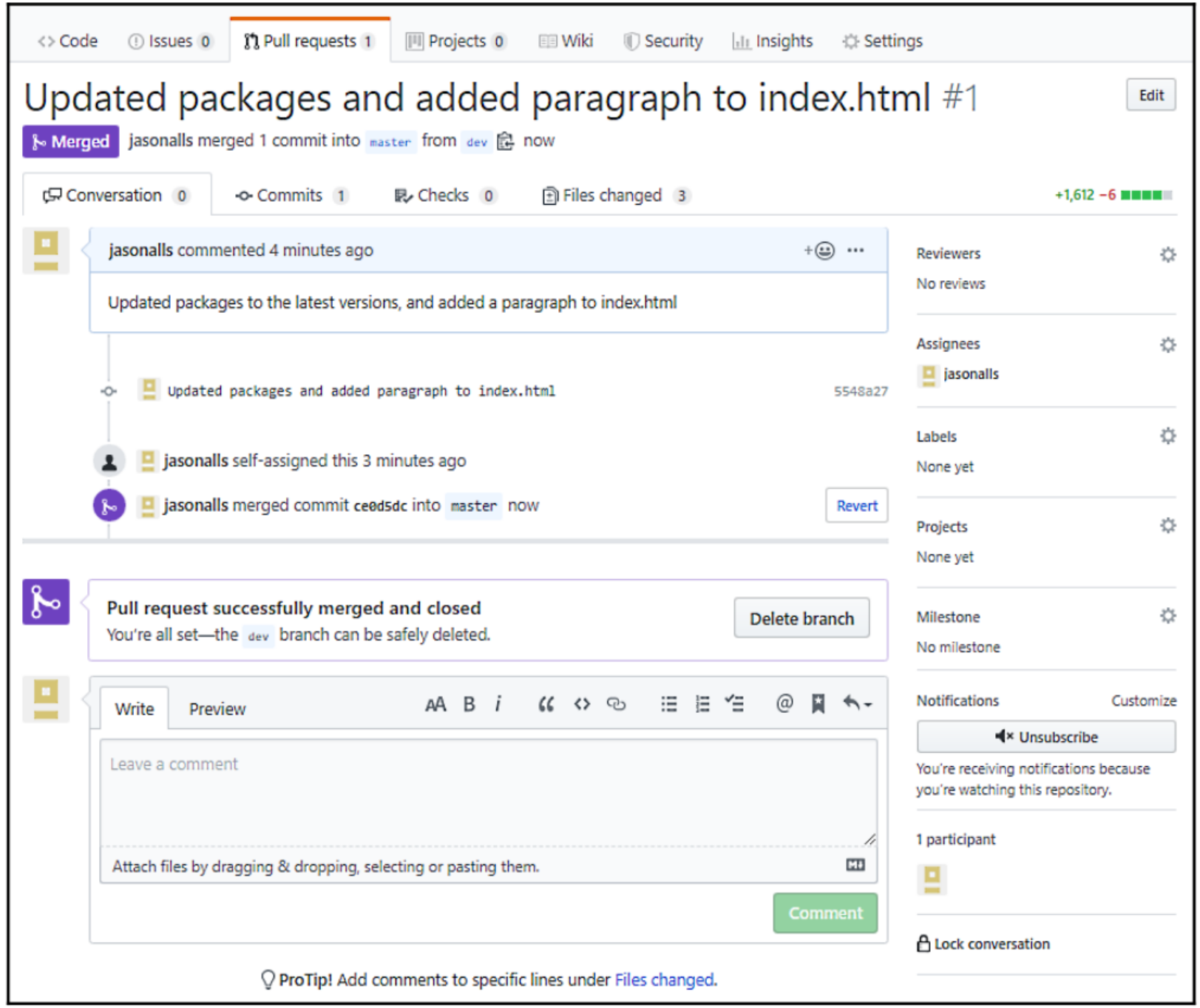
1. Start by cloning a copy of the code under review.
2. Review the comments and changes in the pull request.
3. Check that there are no conflicts with the base branch. If there are, then you will have to reject the pull request with the necessary comments. Otherwise, you can review the changes, make sure the code builds without errors, and make sure there are no compilation warnings. At this stage, you will also look out for code smells and any potential bugs. You will also check that the tests build, run, are correct, and provide good test coverage of the feature to be merged. Make any comments necessary and reject the pull request unless you are satisfied. When satisfied, you can add your comments and merge the pull request by clicking on the **Merge pull request** button, as shown here:


Code
Issues 0
Pull requests 1
Projects 0
Wiki
Security
Insights
Settings
Updated packages and added paragraph to index.html #1
Edit
Open
jasonalls wants to merge 1 commit into master from dev
Conversation 0
Commits 1
Checks 0
Files changed 3
+1,612 −6
jasonalls commented now
Updated packages to the latest versions, and added a paragraph to index.html
Updated packages and added paragraph to index.html 5548a27
jasonalls self-assigned this now
Add more commits by pushing to the dev branch on jasonalls/startgrunt.
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.
Write
Preview
Leave a comment
Attach files by dragging & dropping, selecting or pasting them.
Close pull request
Comment
Reviewers
No reviews
Assignees
jasonalls
Labels
None yet
Projects
None yet
Milestone
No milestone
Notifications
Customize
Unsubscribe
You're receiving notifications because you're watching this repository.
1 participant
Lock conversation

4. Now, confirm the merge by entering a comment and clicking on the **Confirm merge** button:

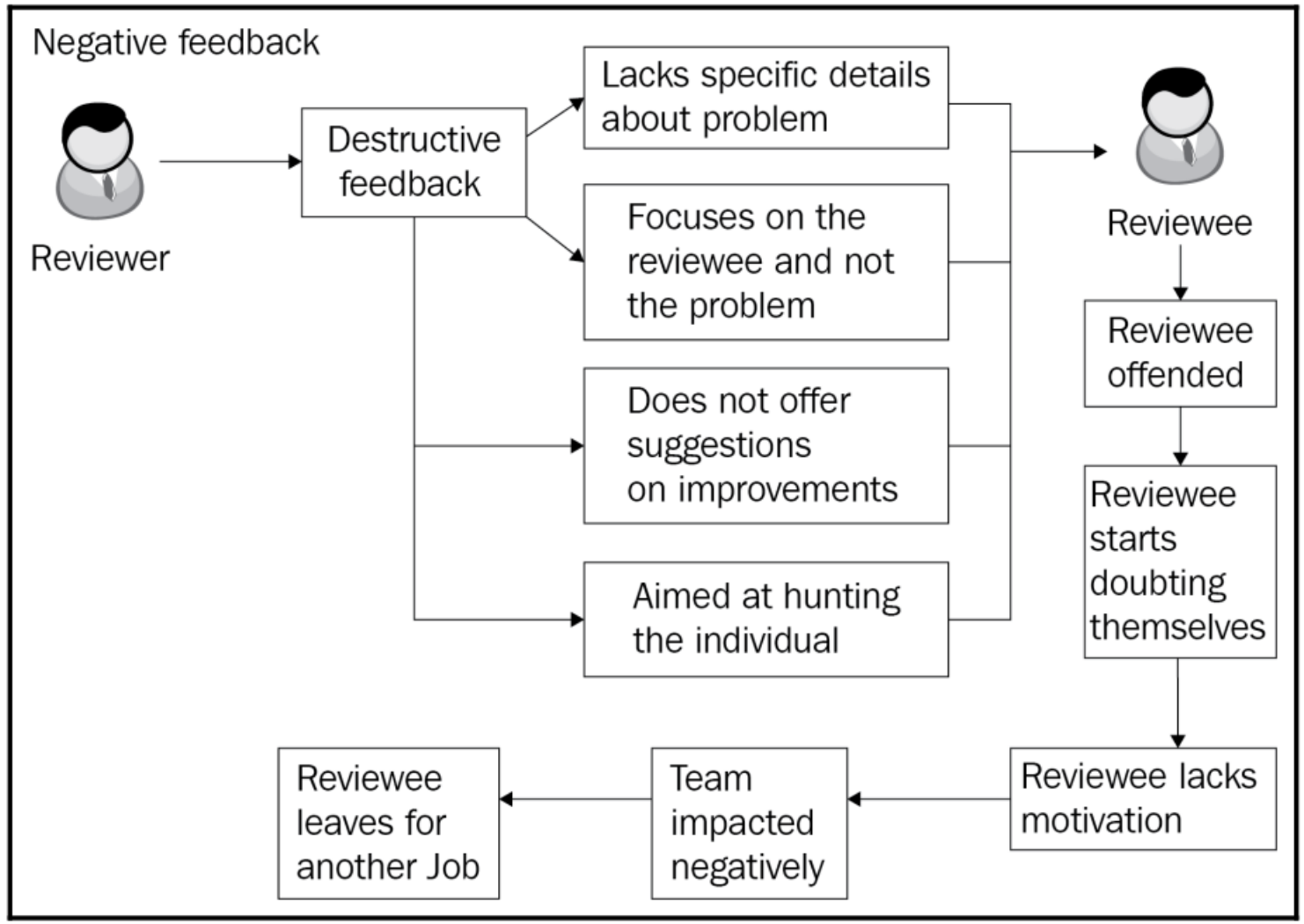


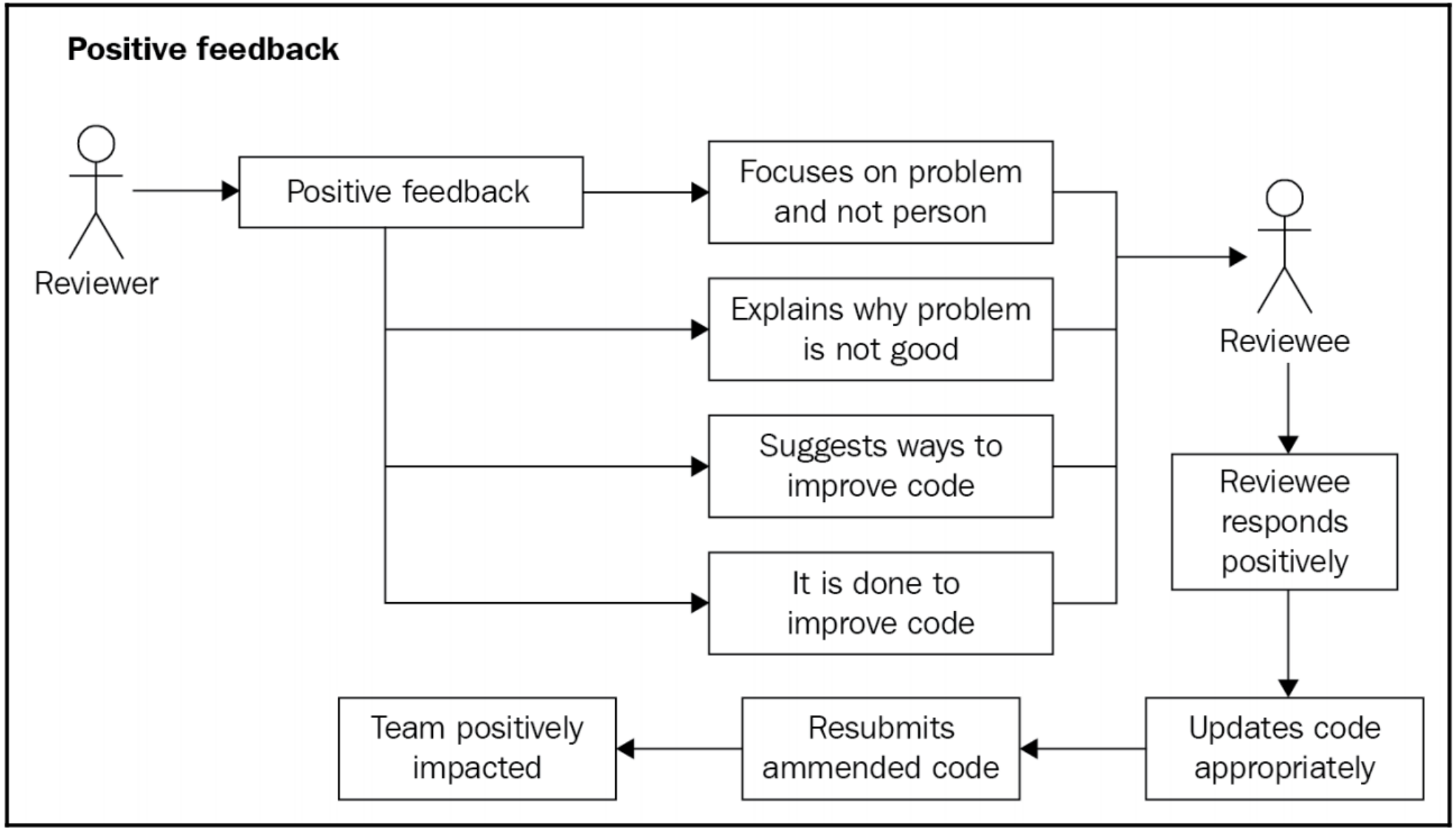
5. Once the pull request has been merged and the pull request closed, the branch can be deleted by clicking on the **Delete branch** button, as can be seen in the following screenshot:


Code
Issues 0
Pull requests 1
Projects 0
Wiki
Security
Insights
Settings
Updated packages and added paragraph to index.html #1
Edit
Merged
jasonalls merged 1 commit into master from dev now
Conversation 0
Commits 1
Checks 0
Files changed 3
+1,612 −6
jasonalls commented 4 minutes ago
Updated packages to the latest versions, and added a paragraph to index.html
Updated packages and added paragraph to index.html
5548a27
jasonalls self-assigned this 3 minutes ago
jasonalls merged commit ce0d5dc into master now
Revert
Pull request successfully merged and closed
You're all set—the dev branch can be safely deleted.
Delete branch
Write
Preview
Leave a comment
Attach files by dragging & dropping, selecting or pasting them.
Comment
ProTip! Add comments to specific lines under Files changed.
Reviewers
No reviews
Assignees
jasonalls
Labels
None yet
Projects
None yet
Milestone
No milestone
Notifications
Customize
Unsubscribe
You're receiving notifications because you're watching this repository.
1 participant
Lock conversation

Effects of feedback on reviewees

When performing a code review of your peer's code, you must also take into consideration the fact that feedback can be positive or negative. Negative feedback does not provide specific details about the problem. The reviewer focuses on the reviewee and not on the problem. Suggestions for improving the code are not offered to the reviewee by the reviewer, and the reviewer's feedback is aimed at hurting the reviewee.


Negative feedback
Reviewer
Destructive feedback
Lacks specific details about problem
Focuses on the reviewee and not the problem
Does not offer suggestions on improvements
Aimed at hunting the individual
Reviewee
Reviewee offended
Reviewee starts doubting themselves
Reviewee lacks motivation
Team impacted negatively
Reviewee leaves for another Job


Positive feedback
Reviewer
Positive feedback
Focuses on problem and not person
Explains why problem is not good
Suggests ways to improve code
It is done to improve code
Reviewee
Reviewee responds positively
Updates code appropriately
Resubmits ammended code
Team positively impacted

Knowing what to review

There are different aspects of the code that have to be considered when reviewing it. Primarily, the code being reviewed should only be the code that was modified by the programmer and submitted for review. That's why you should aim to make small submissions often. Small amounts of code are much easier to review and comment on.

Let's go through different aspects a code reviewer should assess for a complete and thorough review.

Company's coding guidelines and business requirement(s)

Naming conventions

The code should be checked to see whether the naming conventions have been followed for the various code constructs, such as classes, interfaces, member variables, local variables, enumerations, and methods. Nobody likes cryptic names that are hard to decipher, especially if the code base is large.

Here are a couple of questions that a reviewer should ask:

- Are the names long enough to be human-readable and understandable?
- Are they meaningful in relation to the intent of the code, but short enough to not irritate other programmers?

As the reviewer, you must be able to read the code and understand it. If the code is difficult to read and understand, then it really needs to be refactored before being merged.

Formatting

Formatting goes a long way to making code easy to understand. Namespaces, braces, and indentation should be employed according to the guidelines, and the start and end of code blocks should be easily identifiable.

Again, here is a set of questions a reviewer should consider asking in their review:

- Is code to be indented using spaces or tabs?
- Has the correct amount of white space been employed?
- Are there any lines of code that are too long that should be spread over multiple lines?
- What about line breaks?
- Following the style guidelines, is there only one statement per line? Is there only one declaration per line?
- Are continuation lines correctly indented using one tab stop?
- Are methods separated by one line?
- Are multiple clauses that make up a single expression separated by parentheses?
- Are classes and methods clean and small, and do they only do the work they are meant to do?

Testing

Tests must be understandable and cover a good subset of use cases. They must cover the normal paths of execution and exceptional use cases. When it comes to testing the code, the reviewer should check for the following:

- Has the programmer provided tests for all the code?
- Is there any code that is untested?
- Do all the tests work?
- Do any of the tests fail?
- Is there adequate documentation of the code, including comments, documentation comments, tests, and product documentation?
- Do you see anything that stands out that, even if it compiles and works in isolation, could cause bugs when integrated into the system?
- Is the code well documented to aid maintenance and support?

Architectural guidelines and design patterns

The new code must be checked to see whether it conforms to the architectural guidelines for the project. The code should follow any coding paradigms that the company employs, such as SOLID, DRY, YAGNI, and OOP. In addition, where possible, the code should employ suitable design patterns.

Performance and security

Other things that may need to be considered include performance and security:

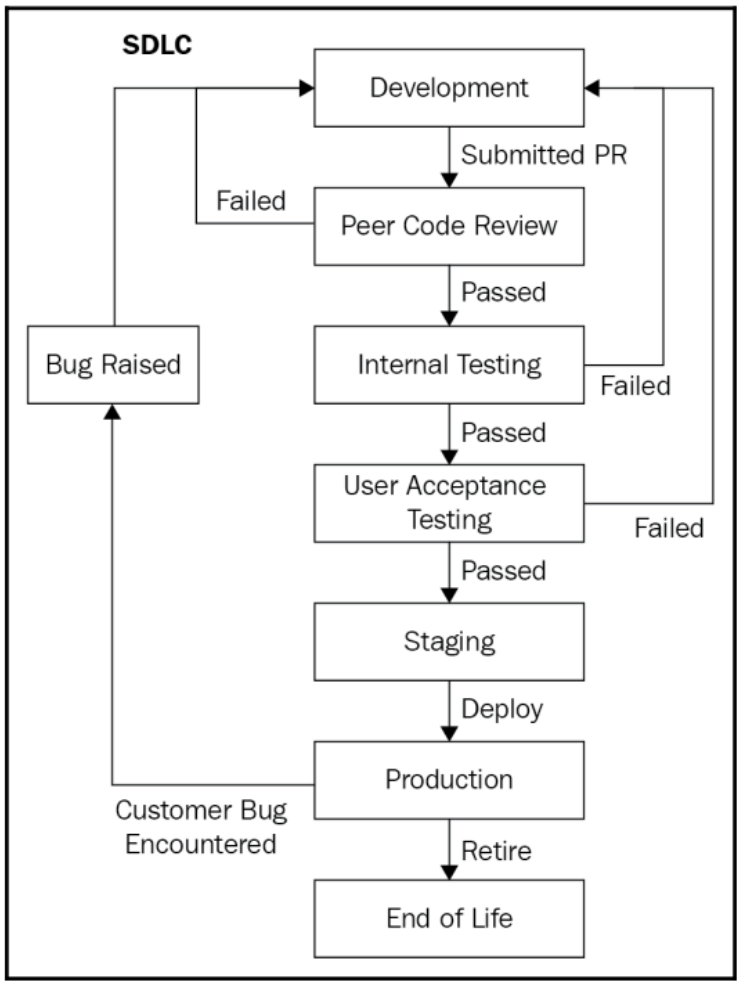
- How well does the code perform?
- Are there any bottlenecks that need to be addressed?
- Is the code programmed in such a way to protect against SQL injection attacks and denial-of-service attacks?
- Is code properly validated to keep the data clean so that only valid data gets stored in the database?
- Have you checked the user interface, documentation, and error messages for spelling mistakes?
- Have you encountered any magic numbers or hard coded values?
- Is the configuration data correct?
- Have any secrets accidentally been checked in?

Knowing when to send code for review

Code reviews should take place when the development is complete and before the programmer of the code passes the code on to the QA department. Before any code is checked into version control, all the code should build and run without errors, warnings, or information. You can ensure this by doing the following:

- You should run static code analysis on your programs to see whether any issues are raised. If you receive any errors, warnings, or information, then address each point raised. Do not ignore them as they can cause problems further down the line. You can access the **Code Analysis** configuration dialog on the **Code Analysis** page of the Visual Studio 2019 **Project Properties** tab. Right-click on your project and select **Properties | Code Analysis**.
- You should also make sure that all your tests run successfully, and you should aim to have all your new code to be fully covered by normal and exceptional use cases that test the correctness of your code against the specification you are working on.
- If you employ a continuous development software practice within your place of work that integrates your code into a larger system, then you need to make sure that the system integration is successful and that all tests run without failing. If any errors are encountered, then you must fix them before you go any further.

When your code is complete, fully documented, and your tests work, and your system integration all works without any issues, then that is the best time to undergo a peer code review. Once you have reached the point that your peer code review is approved, your code can then be passed on to the QA department. The following diagram shows the **Software Development Life Cycle** (**SDLC**) from the development of the code through to the end of the life of the code:



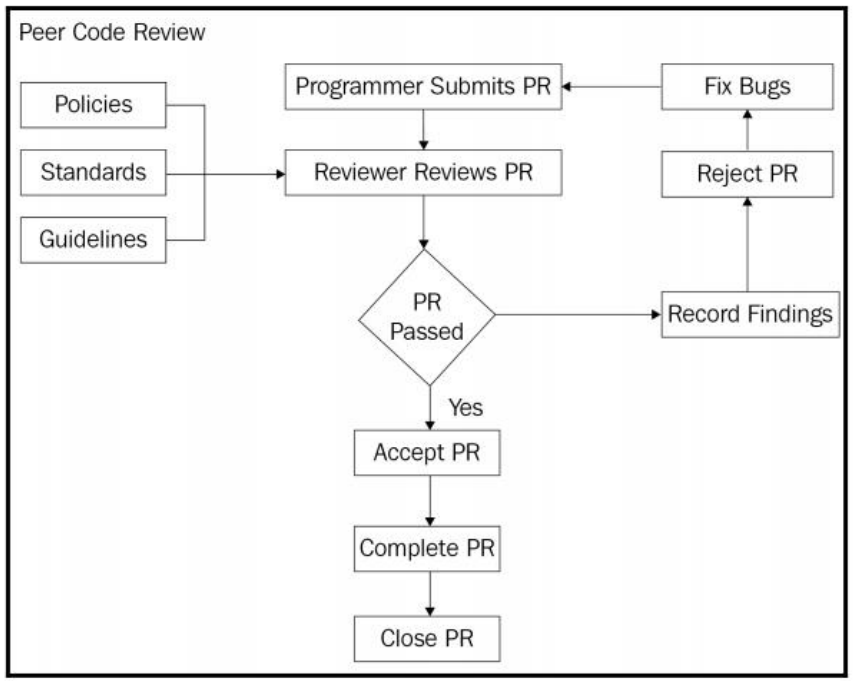
The programmer codes the software as per specifications. They submit the source code to the version control repository and issue a pull request. The request is reviewed. If the request fails, then the request is rejected with comments. If the code review passes, then the code is deployed to the QA team that carry out their own internal testing. Any bugs found are raised for the developers to fix. If the internal testing passes QA, then it is deployed into **User Acceptance Testing** (**UAT**).

If UAT fails, then bugs are raised with the DevOps team, who could be developers or infrastructure. If UAT passes QA, then it is deployed to staging. Staging is the team responsible for deploying the product in the production environment. When the software is in the hands of the customer, they raise a bug report if they encounter any bugs. Developers then work on fixing the customer's bugs, and the process is restarted. Once the product reaches the end of its life, it is retired from service.

Providing and responding to review feedback

It is worth remembering that code reviews are aimed at the overall quality of code in keeping with the company's guidelines. Feedback, therefore, should be constructive and not used as an excuse to put down or embarrass a fellow colleague. Similarly, reviewer feedback should not be taken personally and responses to the reviewer should focus on suitable action and explanation.

The following diagram shows the process of issuing a **Pull Request** (**PR**), performing a code review, and either accepting or rejecting the PR:



Providing feedback as a reviewer

Workplace bullying can be a problem, and programming environments are not immune. Nobody likes a cocky programmer who thinks they are big. So, it is important that the reviewer has good soft skills and is very diplomatic. Bear in mind that some people can easily be offended and take things the wrong way. So know who you are dealing with and how they are likely to respond; this will help you choose your approach and your words carefully.

As the peer code reviewer, you will be responsible for understanding the requirements and making sure the code meets that requirement. So, look for the answers to these questions:

- Are you able to read and understand the code?
- Can you see any potential bugs?

- Have any trade-offs been made?
- If so, why were the trade-offs made?
- Do the trade-offs incur any technical debt that will need to be factored into the project further down the line?

Once your review is complete, you will have three categories of feedback to choose from: positive, optional, and critical. With **positive feedback**, you can provide commendations on what the programmer has done really well. This is a good way to bolster morale as it can often run low in programming teams. **Optional feedback** can be very useful in helping computer programmers to hone their programming skills in line with the company guidelines, and they can work to improve the overall wellbeing of the software being developed.

Finally, we have critical feedback. **Critical feedback** is necessary for any problems that have been identified and must be addressed before the code can be accepted and passed on to the QA department. This is the feedback where you will need to choose your words carefully to avoid offending anyone. It is important that your critical comments address the specific issue being raised with valid reasons to support the feedback.

Responding to feedback as a reviewee

As the reviewee programmer, you must effectively communicate the background of your code to your reviewer. You can help them by making small commits. Small amounts of code are much easier to review than large amounts of code. The more code being reviewed, the easier it is for things to be missed and slip through the net. While you are waiting for your code to be reviewed, you must not make any further changes to it.

As you can guess, you will receive either positive, optional, or critical feedback from the reviewer. The positive feedback works to boost your confidence in the project as well as your morale. Build upon it and continue with your good practices. You may choose to act or not upon optional feedback, but it's always a good idea to talk it through with your reviewer.

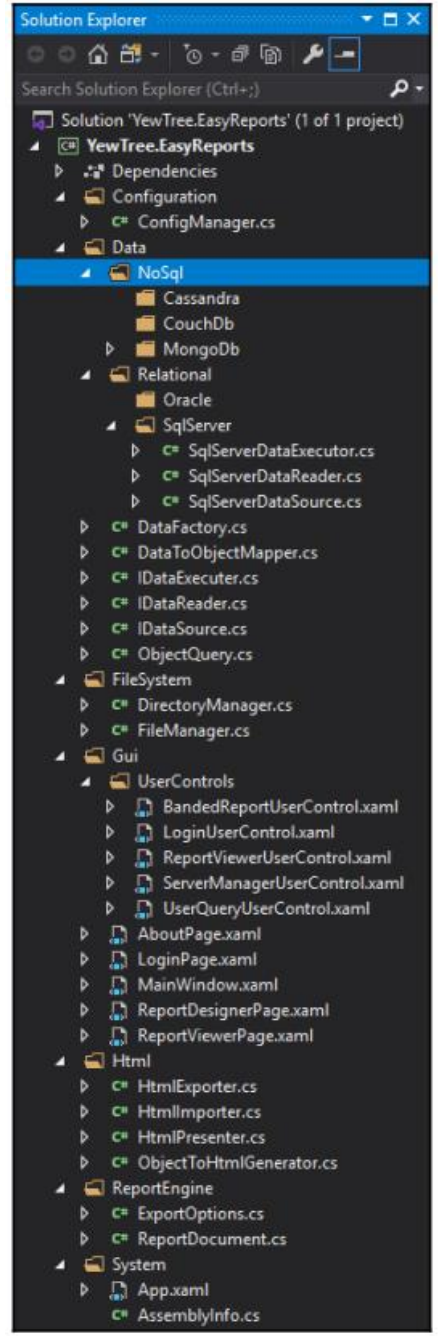
For critical feedback, you must take it seriously and act upon it as this feedback is imperative for the very success of the project. It is very important that you handle critical feedback in a polite and professional manner. Don't allow yourself to be offended by any comments from your reviewer; they are not meant to be personal. This is especially important for new programmers, and programmers who lack confidence.

As soon as you receive your reviewer's feedback, act upon it, and make sure that you discuss it with them as necessary.

Questions

1. What are the two roles involved in a peer code review?
2. Who agrees on the people that will be involved in the peer code review?
3. How can you save your reviewer time and effort prior to requesting a peer code review?
4. When reviewing code, what kinds of things must you look out for?
5. What are the three categories of feedback?

Classes, Objects, and Data Structures


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'YewTree.EasyReports' (1 of 1 project)
YewTree.EasyReports
Dependencies
Configuration
ConfigManager.cs
Data
NoSql
Cassandra
CouchDb
MongoDb
Relational
Oracle
SqlServer
SqlServerDataExecutor.cs
SqlServerDataReader.cs
SqlServerDataSource.cs
DataFactory.cs
DataToObjectMapper.cs
IDataExecuter.cs
IDataReader.cs
IDataSource.cs
ObjectQuery.cs
FileSystem
DirectoryManager.cs
FileManager.cs
Gui
UserControls
BandedReportUserControl.xaml
LoginUserControl.xaml
ReportViewerUserControl.xaml
ServerManagerUserControl.xaml
UserQueryUserControl.xaml
AboutPage.xaml
LoginPage.xaml
MainWindow.xaml
ReportDesignerPage.xaml
ReportViewerPage.xaml
Html
HtmlExporter.cs
HtmlImporter.cs
HtmlPresenter.cs
ObjectToHtmlGenerator.cs
ReportEngine
ExportOptions.cs
ReportDocument.cs
System
App.xaml
AssemblyInfo.cs

A class should have only one responsibility

Responsibility is the work that has been assigned to the class. In the SOLID set of principles, the S stands for **Single Responsibility Principle** (**SRP**). When applied to a class, SRP states that the class must only work on a single aspect of the feature being implemented. The responsibility of that single aspect should be fully encapsulated within the class. Therefore, you should never apply more than one responsibility to a class.

Cohesion and coupling

In a well-designed C# assembly, code will be correctly grouped together. This is known as **high cohesion**. **Low cohesion** is when you have code grouped together that does not belong together.

You want related classes to be as independent as possible. The more dependent one class is on another class, the higher the coupling. This is known as **tight coupling**. The more independent classes are of one another, the lower the cohesion. This is known as low cohesion.

So, in a well-defined class, you want high cohesion and low coupling. We'll now look at examples of tight coupling followed by low coupling.

Design for change

When designing for change, you should change the *what* to the *how*.

The *what* is the requirement of the business. As any seasoned person involved in a role within software development will tell you that requirements frequently change. As such, the software has to be adaptable to meet those changes. The business is not interested in *how* the requirements are implemented by the software and infrastructure teams, only that the requirements are met precisely on time and on budget.

On the other hand, the software and infrastructure teams are more focused on *how* those business requirements are to be met. Regardless of the technology and processes that are adopted for the project to implement the requirements, the software and target environment must be adaptable to changing requirements.

Interface-oriented programming

Interface-Oriented Programming (**IOP**) helps us to program polymorphic code. Polymorphism in OOP is defined as different classes having their own implementations of the same interface. And so, by using interfaces, we can morph our software to meet the needs of the business.

Immutable objects and data structures

Immutable types are normally thought of as just value types. With value types, it makes sense that when they are set, you don't want them to change. But you can also have immutable object types and immutable data structure types. Immutable types are a type whose internal state does not change once they have been initialized.

The behavior of immutable types does not astonish or surprise fellow programmers and so conforms to the **principle of least astonishment** (**POLA**). The POLA conformity of immutable types adheres to any contracts made between clients, and because it is predictable, programmers will find it easy to reason about its behavior.

Since immutable types are predictable and do not change, you are not going to be in for any nasty surprises. So you don't have to worry about any undesirable effects due to them being altered in some way. This makes immutable types ideal for sharing between threads as they are thread-safe and there is no need for defensive programming.

An example of an immutable type

We are now going to look at an immutable object. The `Person` object in the following code has three private member variables. The only time these can be set is during the creation time in the constructor. Once set, they are unable to be modified for the rest of the object's lifetime. Each variable is only readable via read-only properties:

```
namespace CH3.ImmutableObjectsAndDataStructures
{
    public class Person
    {
        private readonly int _id;
        private readonly string _firstName;
        private readonly string _lastName;

        public int Id => _id;
        public string FirstName => _firstName;
        public string LastName => _lastName;
        public string FullName => $"{_firstName} {_lastName}";
        public string FullNameReversed => $"{_lastName}, {_firstName}";

        public Person(int id, string firstName, string lastName)
        {
            _id = id;
            _firstName = firstName;
            _lastName = lastName;
        }
    }
}
```

Now we have seen how easy it is to write immutable objects and data structures, we will look at data and methods in objects.

Objects should hide data and expose methods

The state of your object is stored in member variables. These member variables are data. Data should not be directly accessible. You should only provide access to data via exposed methods and properties.

Why should you hide your data and expose your methods?

Hiding data and exposing methods is known in the OOP world as encapsulation. Encapsulation hides the inner workings of a class from the outside world. This makes it easy to be able to change value types without breaking existing implementations that rely on the class. Data can be made read/writable, writable, or read-only providing more flexibility to you regarding data access and usage. You can also validate input and so prevent data from receiving invalid values. Encapsulating also makes testing your classes much easier, and you can make your classes more reusable and extendable.

Let's look at an example.

An example of encapsulation

The following code example shows an encapsulated class. The `Car` object is mutable. It has properties that get and set the data values once they have been initialized by the constructor. The constructor and the set properties perform the validation of the parameter arguments. If the value is invalid, an invalid argument exception is thrown, otherwise the value is passed back and the data value is set:

```
using System;

namespace CH3.Encapsulation
{
    public class Car
    {
        private string _make;
        private string _model;
        private int _year;

        public Car(string make, string model, int year)
        {
            _make = ValidateMake(make);
            _model = ValidateModel(model);
            _year = ValidateYear(year);
        }
```

```
        private string ValidateMake(string make)
        {
            if (make.Length >= 3)
                return make;
            throw new ArgumentException("Make must be three
             characters or more.");
        }

        public string Make
        {
            get { return _make; }
            set { _make = ValidateMake(value); }
        }

        // Other methods and properties omitted for brevity.
    }
}
```

Data structures should expose data and have no methods

Structures differ from classes in that they use value equality in place of reference equality. Other than that, there is not much difference between a struct and a class.

There is a debate as to whether a data structure should make the variables public or hide them behind get and set properties. It is purely down to you which you choose, but personally I always think it best to hide data even in structs and only provide access via properties and methods. There is one caveat in terms of having clean data structures that are safe, and that is that once created, structs should not allow themselves to be mutated by methods and get properties. The reason for this is that changes to temporary data structures will be discarded.

Let's now look at a simple data structure example.

An example of data structure

The following code is a simple data structure:

```
namespace CH3.Encapsulation
{
    public struct Person
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public Person(int id, string firstName, string lastName)
        {
            Id = id;
            FirstName = firstName;
            LastName = lastName;
        }
    }
}
```

Writing Clean Functions

Understanding functional programming

The only thing that sets functional programming aside from other methods of programming is that functions do not modify data or state. You will use functional programming in scenarios such as deep learning, machine learning, and artificial intelligence when it is necessary to perform different sets of operations on the same set of data.

The *LINQ syntax* within .NET Framework is an example of functional programming. So, if you are wondering what functional programming looks like, and if you have used LINQ before, then you have been subjected to functional programming and should know what it looks like.

Since functional programming is a deep subject and many books, courses, and videos exist on this topic, we will only touch on the topic briefly in this chapter by looking at pure functions and immutable data.

A pure function is restricted to only operating on the data that is passed into it. As a result, the method is predictable and avoids producing side effects. This benefits programmers because such methods are easier to reason about and test.

```
public struct Product
{
    public string Vendor { get; }
    public string ProductName { get; }
    public Product(string vendor, string productName)
    {
        Vendor = vendor;
        ProductName = productName;
    }
}
```

Now that we have our struct, we will add some sample data inside the `GetProducts()` method:

```
public static List<Product> GetProducts()
{
    return new List<Products>
    {
        new Product("Microsoft", "Microsoft Office"),
        new Product("Oracle", "Oracle Database"),
        new Product("IBM", "IBM DB2 Express"),
        new Product("IBM", "IBM DB2 Express"),
        new Product("Microsoft", "SQL Server 2017 Express"),
        new Product("Microsoft", "Visual Studio 2019 Community Edition"),
        new Product("Oracle", "Oracle JDeveloper"),
        new Product("Microsoft", "Azure"),
        new Product("Microsoft", "Azure"),
        new Product("Microsoft", "Azure Stack"),
        new Product("Google", "Google Cloud Platform"),
        new Product("Amazon", "Amazon Web Services")
```

Finally, we can start to use LINQ on our list. In the preceding example, we will get a distinct list of products, ordered by the vendor's names, and print out the results:

```
class Program
{
    static void Main(string[] args)
    {
        var vendors = (from p in GetProducts()
                       select p.Vendor)
                       .Distinct()
                       .OrderBy(x => x);
        foreach(var vendor in vendors)
            Console.WriteLine(vendor);
        Console.ReadKey();
    }
}
```

Keeping methods small

While programming clean and readable code, it is important to keep the methods small. Preferably, in the C# world, it is best to keep methods *under 10 lines* long. The perfect length is no more than *4 lines*. A good way to keep methods small is to consider if you should be trapping for errors or bubbling them further up the call stack. With defensive programming, you can become a little too defensive, and this can add to the amount of code you find yourself writing. Besides, methods that trap errors will be longer than methods that don't.

Indenting code

A very long method is hard to read and follow at the best of times, especially when you have to scroll through the method many times to get to the bottom of it. But having to do that with methods that are not properly formatted with the correct levels of indentation can be a real nightmare.

Avoiding duplication

Code can be either **DRY** or **WET**. WET code stands for **Write Every Time** and is the opposite of DRY, which stands for **Don't Repeat Yourself**. The problem with WET code is that it is the perfect candidate for *bugs*. Let's say your test team or a customer finds a bug and reports it to you. You fix the bug and pass it on, only for it to come back and bite you as many times as that code is encountered within your computer program.

Avoiding multiple parameters

Niladic methods are the ideal type of methods in C#. Such methods have no parameters (also known as *arguments*). Monadic methods only have one parameter. Dyadic methods have two parameters. Triadic methods have three parameters. Methods that have more than three parameters are known as polyadic methods. You should aim to keep the number of parameters to a minimum (preferably less than three).

Implementing SRP

All objects and methods that you write should, at most, have one responsibility and no more. Objects can have multiple methods, but those methods, when combined, should all work toward the single purpose of the object they belong to. Methods can call multiple methods, where each does different things. But the method itself should only do one thing.

Exception Handling

In the previous chapter, we looked at functions. Despite the best efforts of programmers to write robust code, functions will, at some point, generate exceptions. This could be for a number of reasons, such as a missing file or folder, an empty or null value, the location can't be written to, or the user is denied access. So, with that in mind, in this chapter, you will learn about appropriate ways to use exception handling to produce clean C# code. First, we will start by looking at checked and unchecked exceptions with regards to arithmetic `OverflowExceptions`. We will look at what they are, why they are used, and some examples of them being used in code.

In this chapter, we will cover the following topics:

- Checked and unchecked exceptions
- Avoiding `NullPointerExceptions`
- Business rule exceptions
- Exceptions should provide meaningful information
- Building your own custom exceptions

Checked and unchecked exceptions

In unchecked mode, an arithmetic overflow is *ignored*. In this situation, the high-order bits that cannot be assigned to the destination type are discarded from the result.

By default, C# operates in the unchecked context while performing non-constant expressions at runtime. But compile-time constant expressions are *always* checked by default. When an arithmetic overflow is encountered in checked mode, an `OverflowException` is raised. One reason why unchecked exceptions are used is to increase performance. Checked exceptions can decrease the performance of methods by a small amount.

The rule of thumb is to make sure that you perform arithmetic operations in the checked context. Any arithmetic overflow exceptions will be picked up as compile-time errors, and you can then fix them before you release your code. That is much better than releasing your code and then having to fix customer runtime errors.

Running code in unchecked mode is dangerous as you are making assumptions about the code. Assumptions are not facts and they can lead to exceptions being raised at runtime. Runtime exceptions lead to poor customer satisfaction and can produce serious follow-on exceptions that negatively impact a customer in some way.

Allowing an application to continue running that has experienced an overflow exception is very dangerous from a business perspective. The reason for this is that data can end up in a non-reversible invalid state. If the data is critical customer data, then this can be considerably costly to the business, and you don't want that on your shoulders.

Consider the following code. This code demonstrates how bad an unchecked overflow can be in the world of customer banking:

```
private static void UncheckedBankAccountException()
{
    var currentBalance = int.MaxValue;
    Console.WriteLine($"Current Balance: {currentBalance}");
```

```
        currentBalance = unchecked(currentBalance + 1);
        Console.WriteLine($"Current Balance + 1 = {currentBalance}");
        Console.ReadKey();
    }
```

Imagine the horror on this customer's face when they see that adding £1 to their bank balance of £2,147,483,647 causes them to be in debt by -£2,147,483,648!

Now, it's time to demonstrate checked and unchecked exceptions with some code examples. First, start a new **console application** and declare some variables:

```
static byte y, z;
```

The preceding code declares two bytes that we will use in our arithmetic code examples. Now, add the `CheckedAdd()` method. This method will raise a checked `OverflowException` if an arithmetic overflow is encountered when adding two numbers that result in a number that is too big to be stored as a byte:

```
private static void CheckedAdd()
{
    try
    {
        Console.WriteLine("### Checked Add ###");
        Console.WriteLine($"x = {y} + {z}");
        Console.WriteLine($"x = {checked((byte)(y + z))}");
    }
    catch (OverflowException oex)
    {
        Console.WriteLine($"CheckedAdd: {oex.Message}");
    }
}
```

Then, write the `CheckedMultiplication()` method. Again, a checked `OverflowException` will be raised if an arithmetic overflow is detected during the multiplication, which results in a number that is larger than a byte:

```
private static void CheckedMultiplication()
{
    try
    {
```

```
            Console.WriteLine("### Checked Multiplication ###");
            Console.WriteLine($"x = {y} x {z}");
            Console.WriteLine($"x = {checked((byte)(y * z))}");
        }
        catch (OverflowException oex)
        {
            Console.WriteLine($"CheckedMultiplication: {oex.Message}");
        }
    }
```

Next, we add the `UncheckedAdd()` method. This method will ignore any overflow that happens as a result of an addition, and so an `OverflowException` will not be raised. The result of this overflow will be stored as a byte, but the value will be incorrect:

```
    private static void UncheckedAdd()
    {
        try
        {
             Console.WriteLine("### Unchecked Add ###");
             Console.WriteLine($"x = {y} + {z}");
             Console.WriteLine($"x = {unchecked((byte)(y + z))}");
        }
        catch (OverflowException oex)
        {
             Console.WriteLine($"CheckedAdd: {oex.Message}");
        }
    }
```

And now, we add the `UncheckedMultiplication()` method. This method will not throw an `OverflowException` when an overflow is encountered as the result of this multiplication. The exception will simply be ignored. This will result in an incorrect number being stored as a byte:

And now, we add the `UncheckedMultiplication()` method. This method will not throw an `OverflowException` when an overflow is encountered as the result of this multiplication. The exception will simply be ignored. This will result in an incorrect number being stored as a byte:

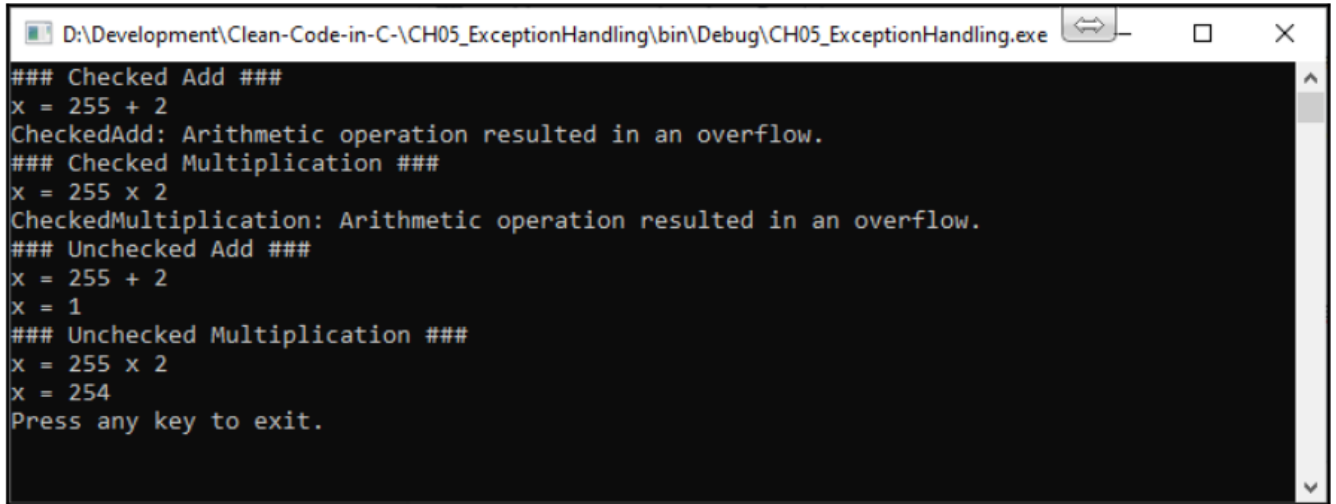
```
private static void UncheckedMultiplication()
{
    try
    {
        Console.WriteLine("### Unchecked Multiplication ###");
        Console.WriteLine($"x = {y} x {z}");
        Console.WriteLine($"x = {unchecked((byte)(y * z))}");
    }
    catch (OverflowException oex)
    {
        Console.WriteLine($"CheckedMultiplication: {oex.Message}");
    }
}
```

Finally, it is time to modify our `Main(string[] args)` method so that we can initialize the variables and execute the methods. Here, we add the maximum value for a byte to the `y` variable and `2` to the `z` variable. Then, we run the `CheckedAdd()` and `CheckedMultiplication()` methods, which will both generate `OverflowException()`. This is thrown because the `y` variable contains the maximum value for a byte.

So, by adding or multiplying by `2`, you are exceeding the address space needed to store the variable. Next, we will run the `UncheckedAdd()` and `UncheckedMultiplication()` methods. Both these methods ignore overflow exceptions, assign the result to the `x` variable, and disregard any bits that overflow. Finally, we print a message to the screen and then exit when the user presses any key:

```
static void Main(string[] args)
{
    y = byte.MaxValue;
    z = 2;
    CheckedAdd();
    CheckedMultiplication();
    UncheckedAdd();
    UncheckedMultiplication();
    Console.WriteLine("Press any key to exit.");
    Console.ReadLine();
}
```

When we run the preceding code, we end up with the following output:



Avoiding NullPointerExceptions

`NullReferenceException` is a common exception that has been experienced by most programmers. It is thrown when an attempt is made to access a property or method on a `null` object.

To defend against computer program crashes, the common course of action among fellow programmers is to use `try{...}catch (NullReferenceExceptionre){...}` blocks. This is a part of defensive programming. But the problem is that, a lot of the time, the error is simply *logged* and *rethrown*. Besides this, a lot of wasted computations are performed that could have been avoided.

A much better way of handling `ArgumentNullExceptions` is to implement `ArgumentNullValidator`. The parameters of a method are usually the source of a `null` object. It makes sense to test the parameters of a method before they are used and, if they are found to be invalid for any reason, to throw an appropriate `Exception`. In the case of `ArgumentNullValidator`, you would place this validator at the top of the method and then test each parameter. If any parameter was found to be `null`, then `NullReferenceException` would be thrown. This would save computations and remove the need to wrap your method's code in a `try...catch` block.

To make things clear, we will write `ArgumentNullValidator` and use it in a method to test the method's arguments:

```
public class Person
{
    public string Name { get; }
    public Person(string name)
    {
        Name = name;
    }
}
```

In the preceding code, we have created the `Person` class with a single read-only property called `Name`. This will be the object that we will use to pass into the example methods to cause `NullReferenceException`. Next, we will create our `Attribute` for the validator called `ValidatedNotNullAttribibute`:

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple =
true)]
internal sealed class ValidatedNotNullAttribute : Attribute { }
```

Now that we have our `Attribute`, it's time to write the validator:

```
internal static class ArgumentNullValidator
{
    public static void NotNull(string name,
     [ValidatedNotNull] object value)
    {
        if (value == null)
        {
            throw new ArgumentNullException(name);
        }
    }
}
```

`ArgumentNullValidator` takes two arguments:

- The name of the object
- The object itself

The object is checked to see if it is `null`. If it is `null`, `ArgumentNullException` is thrown, passing in the name of the object.

The following method is our `try/catch` example method. Notice that we log a message and throw the exception. However, we don't use the declared exception parameter, and so by rights, this should be removed. You will see this quite often in code. It is unnecessary and should be removed to tidy the code up:

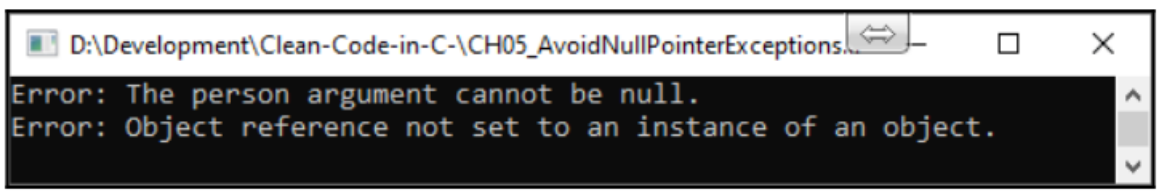
```
private void TryCatchExample(Person person)
{
    try
    {
        Console.WriteLine($"Person's Name: {person.Name}");
    }
    catch (NullReferenceException nre)
    {
        Console.WriteLine("Error: The person argument cannot be null.");
        throw;
```

Next, we will write our example method that will use `ArgumentNullValidator`. We will call it `ArgumentNullValidatorExample`:

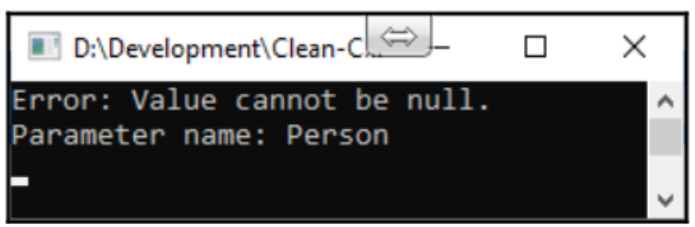
```
private void ArgumentNullValidatorExample(Person person)
{
    ArgumentNullValidator.NotNull("Person", person);
    Console.WriteLine($"Person's Name: {person.Name}");
    Console.ReadKey();
}
```

Notice that we have gone from nine lines, including braces, to only two lines. We also don't attempt to use the value before it has been validated. All we need to do now is modify our `Main` method to run the methods. Test each method by commenting out one of the methods and running the program. When you do this, it is best to step through your code to see what's going on.

The following is the output of running the `TryCatchExample` method:



The following is the output of running `ArgumentNullValidatorExample`:



Business rule exceptions

Technical exceptions are exceptions that are thrown by a computer program as a result of programmer mistakes and/or environmental issues such as there not being enough disk space.

But business rule exceptions are different. Business rule exceptions imply that such behavior is expected and is used to control program flow, when in fact, exceptions should be an exception to the normal flow of the program and not the expected output of a method.

For example, picture a person at an ATM drawing out £100 from their account that has £0 in it and does not have the ability to go overdrawn. The ATM accepts the user request to draw £100 out, and so it issues the `Withdraw(100);` command. The `Withdraw` method checks the balance, discovers that the account has insufficient funds, and so throws `InsufficientFundsException()`.

You may think that having such exceptions is a good idea as they are explicit and help identify issues so that you can carry out a very specific action upon receiving such exceptions – but no! This is not a good idea.

In such a scenario, when the user submits the request, the amount requested should be checked to see if it can be withdrawn. If it can, then the transaction should go ahead, as requested by the user. But if the validation check identifies that the transaction is unable to go ahead, then the program should follow normal program flow to cancel the transaction and inform the user who issued the request without raising an exception.

The withdrawal scenario we've just looked at shows that the programmer has correctly pondered upon the normal flow of the program and the different outcomes. The program flow has been appropriately coded using Boolean checks to allow for the successful withdrawal transactions and to prevent disallowed withdrawal transactions.