# Chapter 7: Validating the Code

Now that you are working with code, tracking changes against work items, and building the code, you need to squeeze our defects. In chapter 4, we discussed how to configure Azure Boards to shed light on every type of work that must be done for a work item to make its way from an idea to the customer. In this way, you are baking defect detection into every part of your process. You can certainly have code that performs perfectly while doing the wrong thing because of poor design or poor analysis. But this chapter is about ensuring that the code is working properly. Since the code is what the software is built from, you want to ensure that your DevOps process and infrastructure is set up to be able to validate it all comprehensively and quickly. You will likely accumulate a volume of code that is impossible to keep in your head. Significant software systems have so many code files that the only way the code can be validated in a manageable way is to automate most of it and create a process for manual review of just the recent changes. This chapter will span steps that will be automated through the continuous integration build, the first deployed environment, and the pull request.

## Strategy for Defect Detection

From the research that our industry has available, and summarized by Capers Jones, “the cost of finding and fixing bugs or defects is the largest single expense element in the history of software”[[1]](#footnote-1). Mr. Jones goes on to report that for the expected 25 year life span of a 500,000 line-of-code .Net system (estimated at 52 LOC C# to 1 Function Point[[2]](#footnote-2)), almost $0.50 of our of every dollar will be spent on finding and fixing bugs. A review of the available quality research would be beneficial to anyone looking to put together a high-performing DevOps environment.

To summarize, defect removal efficiency (DRE) is a metric that has a basic in industry research. Among all of the methods and techniques that are available for maximizing DRE, three emerge as a good balance of investment while together having a track record of achieving the range of 85%-95% DRE. This should be considered the minimal starting point. Excluding any of these techniques will almost certainly yield poor quality given that other techniques are not shown to make up for the lack of these. Use these as an essential starting point and evaluate what your standard should be. The three essential defect removal techniques are:

* Static analysis
* Testing
* Inspections

Discussing 85% isn’t worthwhile without knowing how many defects we should expect to be generated in a given software project. Only then would we know how many defects would have to be caught and fixed in order to arrive at the 85% DRE level. And after this, what number of defects are shipped to production if 15% of them escape? Capers Jones summarizes this research as well in his 2016 article “Exceeding 99% in Defect Removal Efficiency (DRE) for Software”[[3]](#footnote-3). The following table shows the average defects potentials by phase of work. These are the average rate of defects generated by each type of work from software projects studies through 2016

|  |  |
| --- | --- |
| Phase of work | Defects per 100 lines of C# (1 FP = ~52 C#) |
| Requirements | 1.35 |
| Architecture | 0.19 |
| Design | 1.83 |
| Code | 2.21 |
| Security code flaws | 0.48 |
| Documents | 0.87 |
| Bad fixes | 1.25 |

Table: Defects that should be expected by phase of work per 100 lines of resulting C# code.

The research community uses Function Points to normalize projects and make them comparable. We can convert averages into comparable lines of code by using a technique called backfiring. This is where we take that the average function point of software functionality can be implemented in 52 lines of C#. We use this conversation ratio to determine what range of defect potentials might be relevant for our own software system. If our system is 10,000 lines of C# (HTML, VB, SQL all have very similar conversion ratios), we should expect a ballpark defect potential in the neighborhood of 800 defects, from all sources. A minimum quality bar, 85% DRE would catch 680 defects before releasing to the customer and would release to production 120. Research shows that around 25% of these released defects can be caught and fixed each year after release. This is why for systems that have been in production usage for many years can become quite stable – and why new changes tend to break things in a visible way, especially when users have been used to stability.

If our system is much larger, say 500,000 lines of code, we should expect around 41,000 defects to be generated from all phases of work. These number can become quite scary. If we achieve 95% DRE, we are still releasing over 6,000 defects to our customers. 99% DRE would bring the number of defects released to customers to around 400. These numbers are sobering. It is tempting to think that even with industry averages like this, certainly your team is above average. One would hope so, and one should be able to articulate why. If you were to speculate to beat the averages by a factor of 2, feel free to cut these numbers in half. Even there, we can see the importance of a clear defect detection and defect removal strategy if we are to have any hope of producing a quality software system. A highly automated DevOps environment is an enabler of quality and speed, but it must be a rich pipeline, full of quality controls.

Consider the analogy of a water treatment system for a town. We can think of this as a pipeline where water from available sources comes in to the pipeline. Through a series of treatment steps, water that is prone to cause disease and sickness if cleaned, filtered, and treated so that good drinking water is produced as an output. The drinking water is not perfect, but it is good enough for the community. This series of treatments and filters in this water pipeline is what we must create in our DevOps pipeline. The raw ideas that come from business initiatives are not suitable for working software. We translate the ideas into requirements (features), and then we break those down into units that can be implemented (user stories). We translate these into code, then into a deployable release candidate, then into a deployed environment, and then into a working production system. Each step of the way, the work in process coming from the left, as visualized by our swim lanes in Azure Boards, has more hidden defects that we want promoted to stages to the right of our project board. It is up to us to ensure that every time the work moves from one swim lane to the next that there is a filter or a “treatment” that find the defects that are hiding at that point in time and removes them. For the rest of this chapter, we’ll focus on the quality control techniques that are the minimum bar for detecting and removing defects that are produced in the code that our teams write.

### Strategy and Execution of Defect Detection

While this chapter, or book, could not comprehensively cover all of the defect detection techniques you may want to implement, it will cover the three essential techniques. Omitting any of these could be considered malpractices given the documented effectiveness and affordability of each of the three.

Pair Programming as Defect Detection

Pair programming does have a good track record for defect detection. Read the texts cited in this chapter in order to dive into the actual numbers. Pair programming is the act of having two developers create and change the code together, each trading off at the keyboard and swapping roles of coder and navigator. Those who partake in these exercises report anecdotally what the research shows: for tough problems, it helps push through quicker, but for normal to easy code, it creates overhead. The reason this technique isn’t one of the first you should reach for is the high cost as a defect detection method. The rate of return is not as high as static analysis, testing, and inspections because it does double the cost of labor for the same scope of software created. The software is of very high quality, but the ROI does not translate into an economic advantage. This technique is best used for the smaller number of more risky or difficult software features.

Let’s briefly define each of three essential defect removal methods.

**Static Analysis**

Static analysis is the automated examination of a source file in order to predict defects. More broadly, static analysis can be used as a technique against documents and other artifacts as well as source code. The spelling and grammar check in Microsoft Word is a very valuable static analyzer, without which this book might sound very unprofessional, indeed. While the copy editor performs testing on each chapter by reading every word, and the chapter layout proofer inspects images, tables, and margins, the static analyzer in Microsoft Word is run many times, often after every change to the document. Because it is automated, it can be run essentially for free frequently. For our source code, we will implement a number of static analysis tools. These will run automatically as part of our DevOps pipeline. These tools will emit warnings and errors. We may choose to fail a step in our pipeline when errors occur – or choose to fail on new warnings.

**Testing**

Since the dawn of software, testing has been part of the workflow. A programmer has always run the written code to see if it works as intended. In 2002, Kent Beck published a very influential book that has shifted the testing methods of scores of teams. This book is “Test Driven Development: By Example”[[4]](#footnote-4). James Newkirk, co-author of NUnit 2 and XUnit testing frameworks, illustrated TDD for .Net in his 2004 book, “Test-Driven Development in Microsoft .NET”[[5]](#footnote-5). The technique of test-driven development shifts the developer from either manual desk checking or custom test harnesses to a standard pattern for creating executable tests. This standard format, and the method of creation, allows for test suites that continually grow as the software grows. In many cases, the best format for Scrum’s acceptance criteria for a backlog item is a written down test scenario whose steps are coded into an automated test that exercises the system in that fashion.

**Inspections**

Anything that is built is inspected. We value home inspectors that can use a formal checklist to inspect a house or apartment before purchase or move-in. These inspectors are experts. They know what to look for, and they are equipped with a checklist to ensure they don’t forget to inspect all of the necessary items. Lay people cannot be inspectors. They lack the training or knowledge of what to inspect. The author would likewise be unqualified to inspect a house being purchased. In software, one can craft an inspection at several stages in the value chain. The DevOps process includes more than just the pipeline and begins once an idea has been crafted and placed on the project board. Take care to evaluate what steps should include a formal inspection, who should perform it, and what the checklist should be.

### Code Validation in the DevOps Pipeline

We have seen that work moves through our process according to our swim lane progression.



Figure: Standard swim lanes for a measurable DevOps process

For the purposes of this chapter, we will focus on just the following:

* Test Design
* Development
* Functional Validation

These three phases of work surround the code and produce a release candidate that can be further evaluated. So our scope of focus is narrowed to just these three columns.



Figure: Validating the code focuses on these three swim lanes in our process.

For simplicity, here is the part of our automated DevOps pipeline that will be impacted by the implemented of our defect removal methods.



Figure: Validating the code starts a few steps before coding and includes some critical steps after.

This figure is a snapshot of the DevOps process surrounding making code changes. Static analysis, testing, and inspections go in specific places in this process. Each method integrates well into Azure DevOps Services, Visual Studio, and .Net. Let’s take them one at a time.

### Static Analysis

Once you have decided what static analysis tools you should use, you will configure them in the continuous integration build. It is often unnecessary to have them run every time as part of the private build, but developers may run them frequently on their own. Any static analysis tool will be able to be run locally on demand, but you will want to make it an automated part of your pipeline. Placing it before release candidate packaging is important. If the revision doesn’t pass static analysis checks, there may be little point in archiving the packages from the build given that the revision has no chance of every becoming a release candidate.

In Visual Studio, FxCop has long been an available static analysis tool for .Net. It fully supports .Net Framework. With recent changes in the C# compiler, Roslyn-based analyzers have been replacing FxCop and are the preferred method. These analyzers become part of the Visual Studio solution and can run both in the IDE as well as command line. This chapter will not duplicate the documentation, which can be found online[[6]](#footnote-6). Other popular static analysis tools include:

* ReSharper command line – for code style conventions
* Ndepend – for code metrics, warnings, and high-level quality gradings
* SonarQube – for code metrics, warnings, and high-level quality gradings
* TSLint – for readability, maintainability, and functionality errors.
* WAVE – Web Accessibility Evaluation Tool for statically analyzing web pages for screen reader compatibility errors

This is not meant to be a comprehensive list of static analysis tools. There are many, many more. Static analysis is a method for which there are many implementations. Evaluate your software and include as many as you can.

### Testing

Manual testing will always occur. For some validation, only a human eye can uncover a defect that may affect customers. Certainly if there was a defect in colors or a CSS spreadsheet that made all text white on a white background, your software may function just fine, but few customers would be able to use it. The majority of system functionality can be covered by forms of automated testing, and this section will focus on that. By applying levels of automated testing, we minimize the load needed on manual testers and ensure that people performing usability testing do not encounter functional defects. Further, those performing exploratory testing will be able to focus on that task rather than using time to report functional defects preventable by automated testing.

When we consider automated testing, one can group them into four categories.

* Unit tests
* Integration tests
* Full-system tests
* Specialized tests

Rather than create an exhaustive listing, specialized tests include types of testing that do not have a short enough cycle time to reliably include in an automated DevOps pipeline in any comprehensive fashion. Load testing and security testing fall into this category. While you may include some spot checks of these types of tests in your full-system tests, these types of specialized test cases often require special environments and human assistance in order to run. They are valuable, but they will be outside the scope of this chapter. For the first three types of tests, Microsoft provides some documentation[[7]](#footnote-7) and guidance on how they separate these test types within their Azure DevOps product team. They correlate tests in four different categories, L0-L3 and match nicely the list above. All of these tests can be run with popular testing frameworks like NUnit and XUnit.

**Unit tests (L0)**

These tests are very fast. The call stack stays in memory. The average execution time for these tests should hover around 50-70ms. Because of these, code that includes out-of-process dependencies are disqualified. Any of these would make the tests too slow. These tests can test a single method or many classes together, but they should be testing some logical unit of software logic. The watchwords for these tests are small and fast. These tests should be able to be run on each developers workstation as well as on the build server. These tests should be included in the Visual Studio solution with the production code. Some antipatterns for unit tests are:

* Use of global or threading resources like Mutexes, file I/O, registry, etc
* Any dependencies between a test and another
* High consumption of CPU or memory for a single test
* Including code that calls out of the current process

**Integration tests (L1)**

Microsoft’s guidance is that L1 tests should run under 2 seconds. The vast majority of these tests should run within 1 second. Consider 2 seconds to be an upper bound. When the code is covered with unit tests, we are left with a code base where the individual classes do the right thing, but we have not yet proven that all of the modules or layers work together. The best example of this is the database schema, the data layer, and the domain model entities. Entity Framework Core is a very good choice for working with relational data in .Net Core, but without executing tests that round-trip from the domain model entities to the database and back, we cannot know that those components will work when integrated together on a downstream environment. Unit tests will not test this capability because any call to the database is an out-of-process call. Integration tests are run with the continuous integration build as well as within the private build script on the developers’ workstation. These tests should be included in the Visual Studio solution with the production code. Some antipatterns for integration tests are:

* Requirement for large amounts of data setup
* Any functional dependency on any other test
* Validating more than one logical behavior between layers (being too large)
* Requiring external test state or data setup – every test is responsible for its own setup

**Full-system tests (L2)**

These tests are a superset of the designed test scenarios for each developed feature and defect fix proofs created when the root cause of a defect is identified. Full-system tests require a fully deployed environment in order to execute. They often will execute through the same interfaces as other interactors of the software. If a web page, Selenium may be used to type in text boxes and push buttons. All layers of the application or service are online as these tests execute. They are responsible for their own setup, and are often responsible for reliably running in any order even as other tests continually change the state of the system. These tests should assume the context of an identity and execute the full application just as a normal user would. These tests should be included in the Visual Studio solution with the production code. Some antipatterns for integration tests are:

* Unnecessarily slow: while these tests will be a few orders of magnitude slower than unit tests, the aggregate of them will determine the cycle time of a release branch.
* Modify global state
* The use of shared resources that prevent parallelization
* Requirement of 3rd party services that are outside of the team’s control. i.e. Office 365 login, PayPal, etc.

For these three types of automated testing, you will see a decline in the numbers of each. Let’s consider a code base that is 300,000 lines of code. Some averages this author has seen (not backed at all by research) is around a unit test for every 50 lines of code. For covered code, the average should be lower, but some production code will not be covered, especially code that is on the edge, hopelessly coupled to 3rd party libraries and frameworks and wrapped in isolation layers. Beyond this, drop an order of magnitude for your expectation of integration tests. This would be an integration test for every 500 lines of code. Then for full-system tests, one of these for every 5000 lines of code. Giving concrete numbers like these is fraught with peril because inadequate research exists for one to give any numbers at all. Given the uselessness of the “it depends” answer, anecdotal experience has seen ratios such as 100:10:1 when looking at unit tests to integration tests to full-system tests. Don’t expect the drop in order of magnitude to be exact, but do expect each smaller scope of tests to include a greater number. Your ratio is certain to vary, but take alarm if you end up with more full-system tests than integration tests and more integration tests than unit tests. Take alarm if the numbers are similar. You should see a significant different in numbers. For example, full-system tests are testing user scenarios with the fully deployed system online. Each branch of business logic can be tested as a unit test and each branch of database or queue behavior can be tested with an integration test. So take care that you pick the type of test with the smallest scope when determining how to test an aspect of code behavior.

Acceptance Test-Driven Development

Before coding, we have a swim lane called Test Design. In this column, test scenarios are to be added to the work item definition. Scrum calls for clear acceptance criteria to be added to backlog items. Scripted test scenarios are an implementation of Scrum’s acceptance test concept that creates a test name and a set of test steps that can be programmed into an executable full-system test. In this fashion, acceptance criteria is added to an executable regression tests suite so that all accumulated acceptance criteria is validated with every successive build of the software. This puts the product owner, or other leader, in control of this aspect of verifiable quality.

### Inspections

Inspections are a manual process. But it is different than manual testing. An inspection is a consistent process whereby a human checks some work product using the same checklist and criteria as every other work product. In software, we can implement inspections in several places across the broader process. For example, a good pre-code inspection would be after all four elements of design are complete and before the feature or user story is cleared for development. The checklist for this type of inspection might have high level items to verify completeness:

* Feature includes conceptual definition and vision description along with objectives.
* Feature includes detailed user experience design such as wireframes, screen mockups, etc.
* Feature includes changes to architecture layers, new libraries needed, and other key technology decisions
* Feature includes written test scenarios complete with test steps suitable for manual execution as well as test automation

Without this inspection, it would likely be common for features to make it to developers and lack a critical part of the design. Faced with an incomplete design, developers will have to stop developing and backtrack with the right conversations in order to complete the work that was left incomplete upstream in the process. Without catching this design rework, it may appear that the development phase of work is dropping in productivity (throughput) when the developers are actually finding upstream design defects and working to fix them before continuing with coding.

For the purposes of finding coding defects, a good implementation of an inspection would be integrated with the pull request process. If every feature/user story is developed using a feature branch, a pull request can govern and document the process of accepting the changes on the branch back into the master branch. In Azure Repos or GitHub, the pull request experience is rich enough to accommodate a formal, documented inspection. When the pull request fails inspection, which is not to be feared given that this would indicate a defect being found, the branch can continue to be worked in order to resolve the defect. Once the defect is fixed, the branch can be inspected again, and upon passing inspection, the pull request can then be approved, and the branch merged into master. In this example, we would use an expert inspector – another member of the engineering team. For this type of inspection, a power user or product owner would not be a qualified inspector because the target of the inspection is source code. But a product owner/product manager would likely be very interested in the results of the inspection, reports that they are happening, and the number of defects that are found and fixed through executing inspections.

Along with other items, a pull request code inspection might have steps from the following list:

* The application works after a git pull and private build
* The changes conform to the approved architecture of system
* The changes implement the design decisions called out in the feature
* The changes conform to existing norms of the code base
* No unapproved packages or libraries were introduced to the code base
* The code is accompanied by right balance of tests
* All test scenarios in acceptance criteria of the feature have been implemented as full system L2 tests
* Logging is implemented properly and of sufficient detail
* Performance considerations – application specific
* Security considerations – application specific and conforming to organizational standards
* Readability considerations - code is scannable – factored and named so that it is self-documenting and quickly reveals what it does.

When the inspector (pull request approver) approves the pull request, that individual is affirming that they have faithfully inspected the changes on the branch according to the inspection checklist and that in their professional opinion, the branch meets all the demands of the inspection and does not contain any defects that can be seen or suspected at this time. With this large responsibility, code reviews become a thing of the past. Quick glances at the code and subjective comments in the pull request record cease. In its place, we gain a rigorous and comprehensive inspection of each set of changes as branches are created and merged back into master. Using trunk-based development, branches are very short-lived, so inspections remain quick to perform. And because the standards for passing inspection are well-known, developers understand exactly what is expected and submit pull requests fully expecting to pass inspection.

## Implementing Defect Detection

Armed with these defect removal methods and where they should reside in the process, let’s look at how each one of them look in .Net and implementing using Azure DevOps Services.

### Static Analysis

Microsoft provides very good documentation on FxCop analyzers for Visual Studio, and those instructions can be found in the footnotes[[8]](#footnote-8). After adding FxCop analyzers to a .Net Framework application, we can customize the built-in Microsoft rulesets right from within Visual Studio.

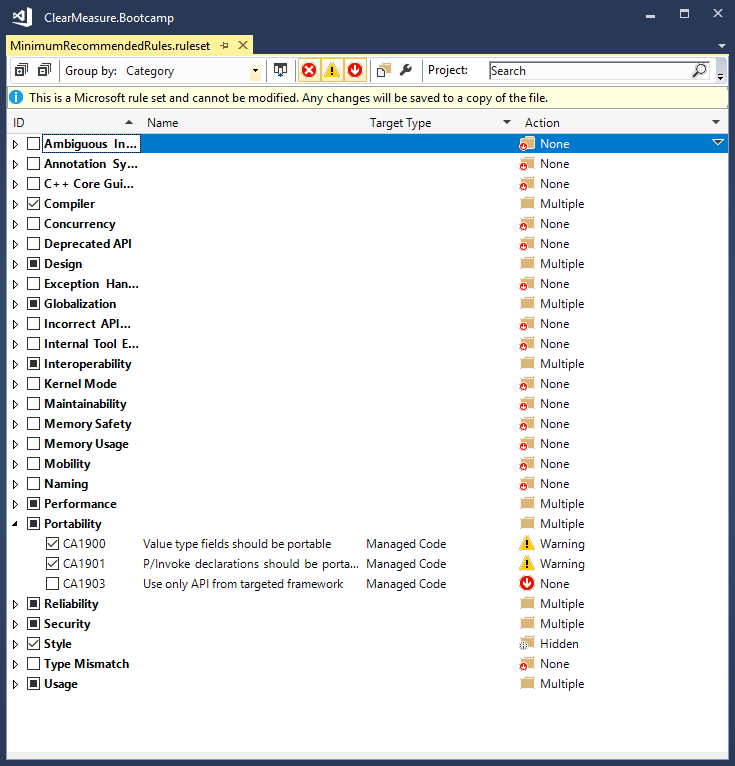


Figure: Visual Studio will save a project-specific ruleset file if you modify any of the settings of the Microsoft ruleset.

In your build script, you can add the following command line arguments so that the analyzers are run when you want them run. Make sure to fail the build on a rule failure.

msbuild.exe

/t:Clean`;Rebuild /v:m /maxcpucount:1 /nologo /p:RunCodeAnalysis=true

/p:ActiveRulesets=MinimumRecommendedRules.ruleset /p:Configuration=Release

src\MySolution.sln

When you add the Nuget package

Microsoft.CodeAnalysis.FxCopAnalyzers

To your project in .Net Core, you’ll see the analyzers appear in your solution explorer, and warnings will start to show when you build your code inside Visual Studio.

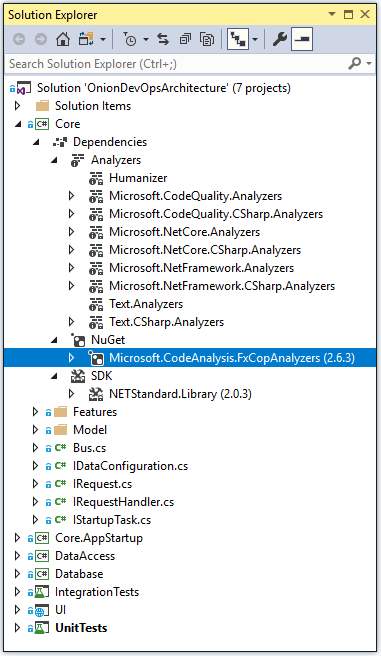


Figure: Code Analyzers are added to a .Net Core project through Nuget

There is no need to add a command line argument to your call to dotnet.exe in your build script. When analyzers are added to your project, they will automatically run and generate the appropriate warnings or errors.

Each static analysis product has its own instructions for integrating it with your code, but in order to keep your Azure Pipelines build configuration simple, make sure to add your static analysis tools to your build script so that the configuration is stored in your Git repository. If you convert your Azure Pipelines build to YAML, you’ll be storing more build logic in Git.

### Testing

Implementing automated tests could fill a volume of its own, and there are plenty of books on the topic. If you are new to test automation, you would spend some time well reading James Newkirks’ book mentioned above. For brevity, here are some examples of the various types of tests that are mentioned in this chapter.

**Unit tests**

In our example application, we have an entity which serves as an an aggregate root, in domain-driven design terms. It has a number of properties and methods. The code for this short class is as follows.

using System;

namespace ClearMeasure.OnionDevOpsArchitecture.Core.Model

{

public class ExpenseReport

{

public Guid Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public ExpenseReportStatus Status { get; set; }

public string Number { get; set; }

public ExpenseReport()

{

Status = ExpenseReportStatus.Draft;

Description = "";

Title = "";

}

public string FriendlyStatus

{

get { return GetTextForStatus(); }

}

protected string GetTextForStatus()

{

return Status.ToString();

}

public override string ToString()

{

return "ExpenseReport " + Number;

}

protected bool Equals(ExpenseReport other)

{

return Id.Equals(other.Id);

}

public override bool Equals(object obj)

{

if (ReferenceEquals(null, obj)) return false;

if (ReferenceEquals(this, obj)) return true;

if (obj.GetType() != this.GetType()) return false;

return Equals((ExpenseReport) obj);

}

public override int GetHashCode()

{

return Id.GetHashCode();

}

}

}

There is quite a bit of logic here that could fail. This logic can be tested inside a single memory space without needed to call out of process to any application dependencies; therefore, we can write some unit tests. In a code base where entities are placed into collections, sorted, and compared, some methods are used by the Base Class Library (BCL) and show a diminished return on investment for explicit unit tests. These methods are Equals() and GetHashCode(). Any entity in a domain model that doesn’t implement these will force other logic to know what property represents its identity in order to see if two objects represent the same record. Most of these objects have data that is pulled from a database of some sort. Full coverage on Equals() and GetHashCode() normally happens automatically as tests of business logic are written. And some tools such as JetBrains ReSharper will generate these methods automatically, so the likelihood of defects is low unless you hand-write them.

A unit test class for ExpenseReport is shown here.

using System;

using ClearMeasure.OnionDevOpsArchitecture.Core.Model;

using NUnit.Framework;

namespace ClearMeasure.OnionDevOpsArchitecture.UnitTests

{

public class ExpenseReportTester

{

[Test]

public void PropertiesShouldInitializeToProperDefaults()

{

var report = new ExpenseReport();

Assert.That(report.Id, Is.EqualTo(Guid.Empty));

Assert.That(report.Title, Is.EqualTo(string.Empty));

Assert.That(report.Description, Is.EqualTo(string.Empty));

Assert.That(report.Status, Is.EqualTo(ExpenseReportStatus.Draft));

Assert.That(report.Number, Is.EqualTo(null));

}

[Test]

public void ToStringShouldReturnNumber()

{

var report = new ExpenseReport();

report.Number = "456";

Assert.That(report.ToString(), Is.EqualTo("ExpenseReport 456"));

}

[Test]

public void PropertiesShouldGetAndSetValuesProperly()

{

var report = new ExpenseReport();

Guid guid = Guid.NewGuid();

report.Id = guid;

report.Title = "Title";

report.Description = "Description";

report.Status = ExpenseReportStatus.Approved;

report.Number = "Number";

Assert.That(report.Id, Is.EqualTo(guid));

Assert.That(report.Title, Is.EqualTo("Title"));

Assert.That(report.Description, Is.EqualTo("Description"));

Assert.That(report.Status, Is.EqualTo(ExpenseReportStatus.Approved));

Assert.That(report.Number, Is.EqualTo("Number"));

}

[Test]

public void ShouldShowFriendlyStatusValuesAsStrings()

{

var report = new ExpenseReport();

report.Status = ExpenseReportStatus.Submitted;

Assert.That(report.FriendlyStatus, Is.EqualTo("Submitted"));

}

}

}

As you read this code file, you see that each test validates that a piece of logic works correctly while keeping all the executing code in-process. Unit tests written in this fashion run very fast, and thousands of them can execute in seconds.

**Integration Tests**

Our ExpenseReport object is persisted, through Entity Framework Core, to a SQL Server database. In order to validate that the Expense Report class can be hydrated from data in SQL Server, we need a test that puts several layers together.

* The domain model itself, containing the Expense Report class
* The Entity Framework Core mapping configuration
* The data access logic, specifying the query to run
* The SQL Server schema, which contains the DDL (Data Definition Language) for the ExpenseReport table

In most cases, these tests are easy to write, but they are very important. Without them, you will encounter defects, and you will spend valuable time debugging through these four layers in order to find the problem. If all of your database-backed classes are equipped with persistence-level integration tests, you will seldom find yourself in a debugging session for a problem in this area.

We have seen the Expense Report class. The next class to examine is the Entity Framework Core mapping configuration, which is comprised of the data context class and a mapping class. The data context class is as follows.

using ClearMeasure.OnionDevOpsArchitecture.Core;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Diagnostics;

namespace ClearMeasure.OnionDevOpsArchitecture.DataAccess.Mappings

{

public class DataContext : DbContext

{

private readonly IDataConfiguration \_config;

public DataContext(IDataConfiguration config)

{

\_config = config;

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.EnableSensitiveDataLogging();

var connectionString = \_config.GetConnectionString();

optionsBuilder

.UseSqlServer(connectionString)

.ConfigureWarnings(warnings =>

warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));

base.OnConfiguring(optionsBuilder);

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

new ExpenseReportMap().Map(modelBuilder);

}

}

}

In our example application, we have one aggregate root, so in our OnModelCreating class, we include one “Map” class. We use this pattern so that as we accumulate hundreds of mapped entities, each have their own class rather than bloating the single DataContext class.

using System;

using ClearMeasure.OnionDevOpsArchitecture.Core.Model;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

using Microsoft.EntityFrameworkCore.ValueGeneration;

namespace ClearMeasure.OnionDevOpsArchitecture.DataAccess.Mappings

{

public class ExpenseReportMap : IEntityFrameworkMapping

{

public EntityTypeBuilder Map(ModelBuilder modelBuilder)

{

var mapping = modelBuilder.Entity<ExpenseReport>();

mapping.UsePropertyAccessMode(PropertyAccessMode.Field);

mapping.HasKey(x => x.Id);

mapping.Property(x => x.Id).IsRequired()

.HasValueGenerator<SequentialGuidValueGenerator>()

.ValueGeneratedOnAdd()

.HasDefaultValue(Guid.Empty);

mapping.Property(x => x.Number).IsRequired().HasMaxLength(10);

mapping.Property(x => x.Title).HasMaxLength(200);

mapping.Property(x => x.Description).HasMaxLength(4000);

mapping.Property(x => x.Status).HasMaxLength(3)

.HasConversion(status => status.Code

, s => ExpenseReportStatus.FromCode(s));

return mapping;

}

}

}

Rather than rely on defaults, which tend to change, our map class specifies how to map each property. Choosing to be explicit in this fashion also lowers the bar for developers understanding what is going on. Each developer will have a different level of memorization for what Entity Framework Core’s default behavior is. Our ExpenseReport table looks like the following.

CREATE TABLE [dbo].[ExpenseReport] (

[Id] UNIQUEIDENTIFIER NOT NULL,

[Number] NVARCHAR (10) NOT NULL,

[Title] NVARCHAR (200) NULL,

[Description] NVARCHAR (4000) NULL,

[Status] NCHAR (3) NOT NULL

);

With four different layers of code running across two different processes, most of the time across a network on different servers, you should see the importance of an automated test ensuring the stability of the integration of these layers. Our integration test to validate persistence logic is here.

using ClearMeasure.OnionDevOpsArchitecture.Core.Model;

using NUnit.Framework;

using Shouldly;

namespace ClearMeasure.OnionDevOpsArchitecture.IntegrationTests.DataAccess.Mappings

{

public class ExpenseReportMappingTester

{

[Test]

public void ShouldPersist()

{

new DatabaseTester().Clean();

var report = new ExpenseReport

{

Title = "TestExpense",

Description = "This is an expense",

Number = "123",

Status = ExpenseReportStatus.Cancelled

};

using (var context = new StubbedDataContextFactory().GetContext())

{

context.Add(report);

context.SaveChanges();

}

ExpenseReport rehydratedExpenseReport;

using (var context = new StubbedDataContextFactory().GetContext())

{

rehydratedExpenseReport = context.Find<ExpenseReport>(report.Id);

}

rehydratedExpenseReport.Title.ShouldBe(report.Title);

rehydratedExpenseReport.Description.ShouldBe(report.Description);

rehydratedExpenseReport.Number.ShouldBe(report.Number);

rehydratedExpenseReport.Status.ShouldBe(report.Status);

}

}

}

This pattern for an integration test can be repeated across all classes that must be persisted to a database through an object-relational mapper. The base case is to send an object through the ORM to the database, clear memory, and then query again to build up the object. We have our first test helper illustrated in this case. The call to DatabaseTester.Clean() represents a helper that can remove all records from all tables in the database in the order of foreign key dependencies. It contains a bit too much code than can be printed in this book. If you are interested in it, clone the Git repository that accompanies this book. In integration tests involving a database, each test is responsible for putting the database in a known state. In many cases, it can be appropriate to run a test starting with no records in the database. Certainly, this case works that way. In other cases, you may want a small known set of data to be loaded into the database before the test suite executes. Maintaining a data test set for build purposes can become time-consuming, so don’t make that your first solution.

**Full-system tests**

Full-system tests, implementing acceptance criteria should begin at external interfaces of the application. If the feature in question is a web service, then the test should perform setup and call the web service. If the interface is a user interface screen, the test should navigate to the screen and use it. If the interface is file ingestion of a custom Excel file for data import, the test should build up an Excel file and place it in the right file path to be process. You get the pattern.

Since web applications are so popular, you will definitely have Selenium tests running in your .Net DevOps pipeline. You can see how to implement Selenium tests in Microsoft’s docs[[9]](#footnote-9). For a simple form-based login screen, a Selenium test might look similar to the following.

[Test]

public void ShouldLoginAndLogOut()

{

Driver.Navigate().GoToUrl(AppUrl);

var login = Driver.FindElement(By.XPath("//button[contains(text(), 'Log In')]"));

login.Click();

Driver.Title.ShouldStartWith("Home Page");

var logout = Driver.FindElement(By.LinkText("Logout"));

logout.Click();

Driver.Title.ShouldStartWith("Login");

}

In this case, Driver is a property that is the Selenium Driver class that wraps a model of the web page being viewed by the browser. These tests can execute from any machine where the executing identity can actually start up and control and instance of a web browser. And since full-system tests are run against a fully deployed environment, it is important that the CI build process packages up the test suite and deploys it along with the application components in the TDD environment. We will cover more about packaging and deploying in later chapters.

### Inspections

A pull request in Azure Repos or GitHub is the perfect place to facilitate a code inspection. Here is a flow in Azure Repos. We start with a feature branch ready for merging. The developer creates the pull request. By policy, the developer initializes the description with a markdown task list that includes all the steps of the inspection. This can be pulled from a wiki or markdown file stored with the application.

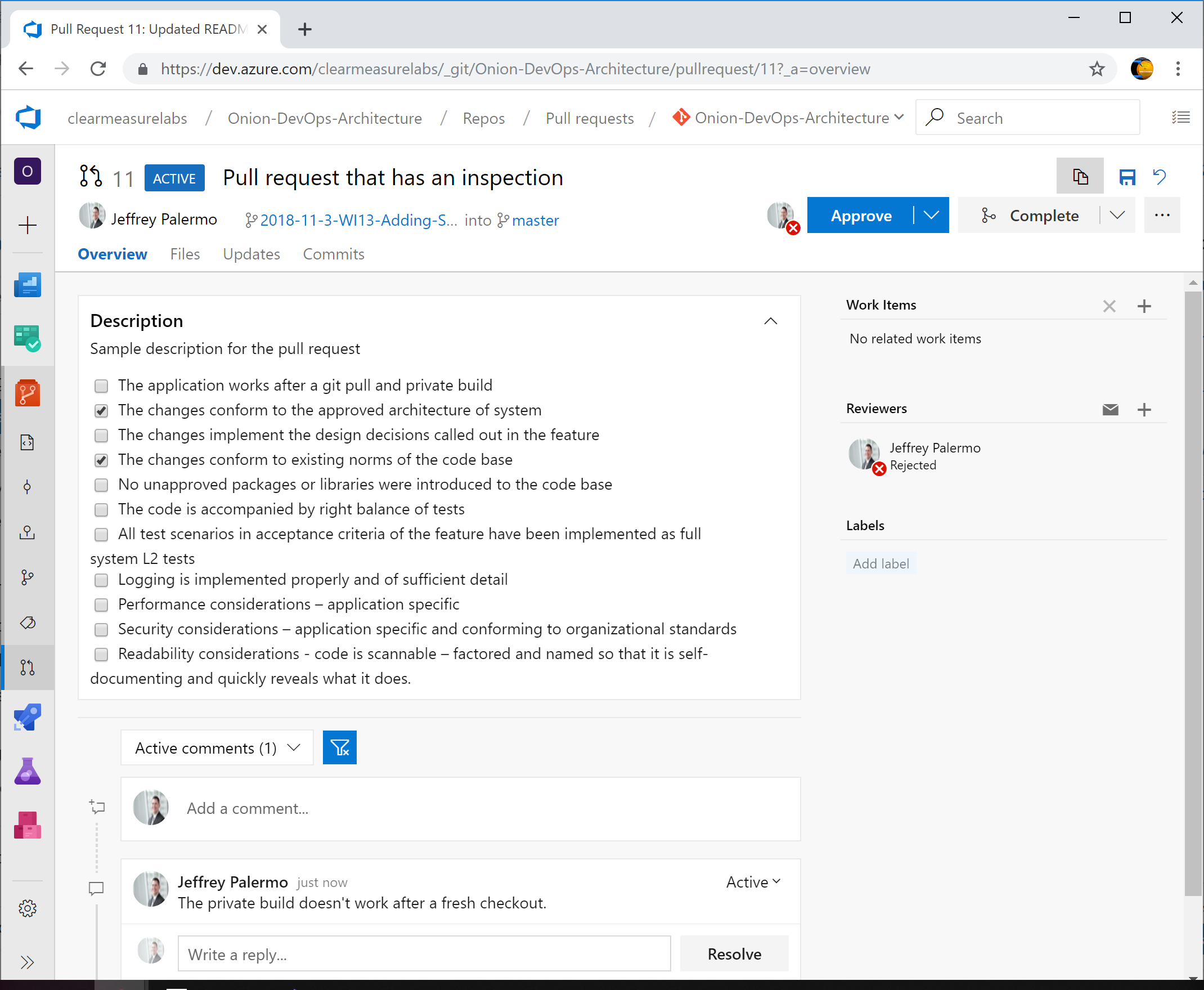


Figure: Pull request that executes a multi-step inspection

The approver can check off the items as they are inspected. When an item fails, the comments can be used and the pull request rejected. More commits can be added to the branch to fix the issue. Then, using the comments in the pull request, the submitter can request that the inspector have another look. Once the branch meets all criteria in the inspect, the inspector approves the pull request and merges the branch. The checklist, and the complete dialog used to resolve any issues is fully documented in Azure Repos.

## Wrap up

In this chapter, you’ve learned how to use some available research to predict how many defects to expect for your application. You’ve also learned three of the critical defect removal methods available in the industry. We’ve covered static analysis, multiple levels of testing, and the concept and implementation of inspections. Armed with these defect removal methods, your teams will quickly remove defects even within development rather than promoting them to downstream phases.

# Bibliography

Beck, K. (2002). *Test Driven Development: By Example.* Addison-Wesley Professional.

*Install FxCop analyzers in Visual Studio*. (2018, 8 2). Retrieved from Visual Studio Docs: https://docs.microsoft.com/en-us/visualstudio/code-quality/install-fxcop-analyzers?view=vs-2017

Jones, C. (2012). Retrieved from SOFTWARE DEFECT ORIGINS AND REMOVAL METHODS: https://www.ifpug.org/Documents/Jones-SoftwareDefectOriginsAndRemovalMethodsDraft5.pdf

jones, C. (2016). *Exceeding 99% in Defect Removal Efficiency (DRE) for Software.* Retrieved from http://www.ifpug.org/Documents/Toppin99percentDRE2016.pdf

Jones, C. (2017). *Software Economics and Function Point Metrics: Thirty years of IFPUG Progress.* Retrieved from http://www.ifpug.org/wp-content/uploads/2017/04/IYSM.-Thirty-years-of-IFPUG.-Software-Economics-and-Function-Point-Metrics-Capers-Jones.pdf

Microsoft. (n.d.). *Get started with Roslyn analyzers*. Retrieved from Visual Studio Docs: https://docs.microsoft.com/en-us/visualstudio/extensibility/getting-started-with-roslyn-analyzers?view=vs-2017

Microsoft/Azure. (n.d.). *Shift Left to Make Testing Fast and Reliable*. Retrieved from Azure DevOps Docs: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/shift-left-make-testing-fast-reliable#test-taxonomy

Newkirk, J. W., & Vorontsov, A. A. (2004). *Test-Driven Development in Microsoft .NET.* Microsoft Press.

*UI test with Selenium*. (n.d.). Retrieved from https://docs.microsoft.com/en-us/azure/devops/pipelines/test/continuous-test-selenium?view=azure-devops

1. (Jones, SOFTWARE DEFECT ORIGINS AND REMOVAL METHODS, 2012) [↑](#footnote-ref-1)
2. (Jones, Software Economics and Function Point Metrics: Thirty years of IFPUG Progress, 2017) [↑](#footnote-ref-2)
3. (jones, 2016) [↑](#footnote-ref-3)
4. (Beck, 2002) [↑](#footnote-ref-4)
5. (Newkirk & Vorontsov, 2004) [↑](#footnote-ref-5)
6. (Microsoft, n.d.) [↑](#footnote-ref-6)
7. (Microsoft/Azure, n.d.) [↑](#footnote-ref-7)
8. (Install FxCop analyzers in Visual Studio, 2018) [↑](#footnote-ref-8)
9. (UI test with Selenium, n.d.) [↑](#footnote-ref-9)