WRITING TESTABLE CODES

Course Module

- Introduction
- Creating Seams in Code
- Constructing Testable Objects
- Working with Dependencies
- Managing Application State
- Maintaining Single Reponsibility
- Next Steps

Purpose

- Learn to write testable code
- Make Unit testing and TDD easier
- Make software more maintainable

Audience

- Developers writing unit tests
- Developer practicing TDD

Prerequisite

- Basic programming experience
- Unit testing or TDD experience

Why should we Test Code?

- Reduce bugs
- Reduce cost
- Improve desing
- Documentation
- Eliminate fear

Type of Tests

Some test are base on:

- **What they are testing**
  - Unit tests
  - Integration tests
  - Componenet tests
  - Service tests
  - UI tests
- **Why they are being tested**

  - Functional tests
  - Acceptance tests
  - Smoke tests
  - Explotary tests
- **How they are being tested**
  - Automated tests
  - Semi-automated tests
  - Manual tests

What is a Unit test?

A unit test is a type of automated software test that verifies the correct behaviour of a single unit of production code in isolation.

What is Testable Code?

- Testable code is code that makes testing easier, not harder.
- It's code that has been designed with testability in mind so that creating automated unit tests is relatively quick and easy.

How do we write Testable Code?

- ***Create seams in code*** that allow injecting tests into the code.
- ***Simplify construction*** of objects to separate the testing of their construction versus the testing of their behaviour.
- ***Word directly with dependencies*** rather than digging through a chain of dependencies.
- ***Decouple code*** from global state in the application in order to unit test in isolation and in parallel.
- ***Maintain single responsibility principle*** to keep tests focused and simple.
- ***Use Test-Driven Development TDD*** to drive the design of our tests.

Guidelines

- Context is important
- Use best judgement

Demonstation --- Building a simple application to create invoices for automobile parts and service. The demo code will involve various task such as calculating, printing and emailing these invoices. There will be omission of many aspects of modern software development typically found in real-world software, such as logging, security, caching, exception handling, and more.

Demo Process

- Hard to test code
- Easy to test code
- Unit test the code

Demo Technologies

.NET Framework 4.8	.NET Core 6.0

.NET Framework 4.8	.NET Core 6.0
Visual Studio 2022	Visual Studio 2022
C# .NET 6.0	C# .NET 10.0
NUnit 3.13	NUnit 3.13
Moq 4.18	Moq 4.18
Moq.AutoMock 3.4	Moq.AutoMock 3.4
Ninject 3.3	Ninject 3.3

Open-Source Demo Code

Download a copy of the demo code from the Exercise files tab

View, Dowload and modify the source code from Github repository

Creating Seam in Code

- Seams
- Problems
- Symptoms
- Solution
- **A seam** is a place in our code where we can alter the behaviour of the program without manually editing the code in that place.
- **Seams** allow us to replace the calling class with a test fixture [`TestFixture`] and replace any dependency with a test double [`TestDouble`], which is a class that stands in for the actual dependency for testing purposes.

Problems

- Cannot pull apart code
- Cannot connect test harness
- Cannot replace dependencies
- Cannot test in isolation

Systoms

- **Keyword `new` in code:** Watch out for `new` keyword in application logic. The `new` keyword is an indication that we're creating a dependency from inside of our class.
- **`Static` method calls:** Keep an eye out for classess that make calls to **`static`** methods on one or more of its dependencies. Static methods create very tight coupling between the caller of the method and the class with the static method being called.
- **Direct `Coupling`:** watch out for places in our code where we have direct coupling to third party frameworks and external resources.

Solution

- **Create `Seams`:** we need to create seams in our code to separate components that are wired directly together.
- `Decouple` **dependencies:** seams allow us to decouple our code from its dependencies.
- **Program to `Interfaces`:** program to interfaces rather than programming to implementations.
- `Inject` **Dependencies:** inject the dependencies of our classes into the classes themselves. In general, we should prefer to inject our dependencies via constructor injection.
- **Test in `Isolation`:** If we decouple from our dependencies by programming to interfaces and injecting our dependencies, this means that we can test any class in insolation.

Constructing Testable Objects

- Constructors
- Problems
- Symptoms
- Solution
- **Constructors** are methods in our class that are `used to build instances of objects` for that class.
- Inside of constructor we execute code necessary to `prepare the object for use`.

Problems

- **`Creates tight coupling:`** If our constructors create any complex dependencies, this `creates tight coupling` beween our class and its dependencies.
- **`Logic is diffiuclt to test:`** If we have logic in our constructor, it is difficult to `test this logic directly`. The only way we can test the logic in a constructor is by constructing the object in different ways and verifying the object's state, which is often hidden from direct testing.
- **`Logic is difficult to setup:`** If we have logic in our constructor, it makes our `setup process` for testing more difficult.

Systoms

- **`Keyword new:`** seeing the `new` keyword in a constructor is often a systom.
- **`Logic in constructor:`** any logic in our constructor is usually a systom of testability problems. This could be a conditional statement, or it could be a loop.
- **`Any non-assignment code:`** any code other than the assignment of values is often an indication of a potential problem.

Solution

- **`Inject dependencies:`** Inject our dependencies rather than creating them in our constructors. Withing the constructor we simply assign those dependencies to private variables inside our class.
- **`Avoid Logic in constructor:`** we want to avoid adding logic in our constructors. also there is need to be careful that logic is not moved from the constructor into an initialization method and then call

that method from the constructor.
- `Use factory, builder or IoC/DI:` use well-established design patterns that separate the construction of the objects and object graphs from the application logic. For example, use the `factory pattern`, the `builder pattern`, or `dependency injection` via and `IoC container`.
- `Don't mix construction and logic:`
- `Separate injectables vs. newables`
    - ***An Injectable*** is an object that is composed of other injectables and performs work on newable objects. injectables are generally services that implement interfaces. For example Database, Printer and InvoiceWriter classes.
    - ***A newable*** is an object at the end of your object graph. These are generally things like entities and value objects. For example Invoice, Customer, Address, and Credit car numbers.

Injectable vs Newable

- *Injectable can ask for injectable... but should not ask for newable.*
- *Newable can ask for newable... but should not ask for injectable*