Clean Code TDD

Sebastian Lindgren

Välkomna till dagens föreläsning!

Clean Code - TDD!

Dagens agenda

- Repetition design patterns
- Testdriven utveckling
- Mock objekt

Repetition av design patterns

Vad minns ni om:

- Facade
- Proxy
- Observer
- Builder

Testdriven utveckling (TDD)

I den testdrivna utvecklingsmetoden skriver du testerna först. Med väl valda tester kan du sedan skriva funktioner som du direkt kan veta att dem håller för det du vill att dem ska hålla. Det krävs övning för att kunna skriva tester som håller för många olika test cases (och för att testa edge cases) [5].

Testdriven utveckling (TDD)

Test driven utveckling funkar bäst när man inte behöver tänka på GUI. Till exempel i många ackend sammanhang. GUIs kan fortfarande bli testade på så vis att man testar saker som går att hämta programatiskt, t.ex. storlek på skärmen etc. Det kan bli problematiskt att till exempel testa saker som har med UI/UX att göra, t.ex. responser till klick eller tangentbordsklick från användare. [1] [4] [3]

Testdriven utveckling (TDD)

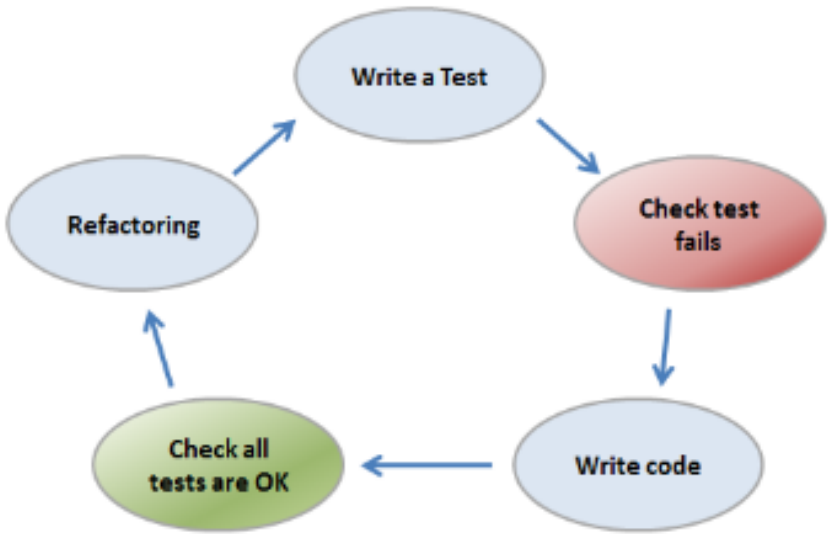
Några av fördelarna med att anvädna TDD [5]:

- Code coverage - du säkerställer att varje funktion får minst ett test
- Regression testnig - du säkerställer att det finns en suite med tester med alla gamla krav redan där, dessa kommer underlätta regression testing och det blir lättare att lägga till nya funktioner utan att vara orolig över att göra sönder något.
- Förenklad debugging - det blir lättare att lokalisera var ett problem kommer ifrån.
- Systemdokumentation - testerna kan hjälap att förklara hur funktionerna används och agerar på så vis som en slags dokumentation.

Testdriven utveckling (TDD)

- Skriv ett test
- Kör test
- fail
- Implementera funktionalitet och refaktor (förbättra)
- pass (eventuellt)
- Gå till 1

Introduction to Test Driven Development



Testing

- Testing is necessary to make sure that our program is as free from bugs as possible.
- Testing does not prove that the program is correct. Proving program correctness has been a research topic for a long time. Still, no easy-to-use solution has come forward.
- Traditionally, the testing was done at the end of the development and it's purpose was (of course) to find as many bugs as possible, and correct them.
- Today, we tend to have tests around earlier. When we improve/refactor our code we also want to run tests before and after, so we may feel comfortable in not breaking the code in the process.

Example class that needs to be tested


```
public class Account {
int Balance {get; private set;}
string Owner {get; private set;}
int AccountNumber {get; private set;}
public Account(String name) { //... }
public void Deposit (int amount) { //... }
public void Withdraw(int amount) { //... }
public String toString() { //... }
}
```

How not to test


```
public class AccountTesterMain {
public static void Main() {
Account a1 = new Account("Ulf");
Console.Writeln("Balance is " + a1.Balance);
a1.Deposit(230);
Console.Writeln("Balance is " + a1.Balance);
a1.Withdraw(50);
Console.Writeln("Balance is " + a1.Balance);
try {
a1.Withdraw(200);
} catch (Exception e) {
Console.Writeln("Caught Exception");
}
}
}
```

What is wrong with this test program?

A test tool

- If you want to run tests often:
  - Tests must be easy to set up
  - Tests must be easy to run
  - The test results must be easy/quick to analyze

Good news / bad news

- Good news: there is a simple tool that helps you with all of this: **MSTest**
- Bad news: you still have to design and program the tests and make sure that they cover “everything”.

MSTest

- MSTest is a tool that will help you to handle and manage tests with minimal effort.
- It is Microsofts own standard tool. There are some others: xUnit, nUnit
- Direct support in VS
- Demo time...

Unit testing

- MSTest is very good for “unit testing“, i.e. testing each component before you use it together with something else [2].
- A “unit” could be a single class, a couple of them working together or a bigger module.
- You should never be able to say: “I didn’t have time to run the tests”.

Refactoring

- The term “refactoring” means modifying the code to make it better – but without changing the functionality
- Refactoring should be a normal part of any development process. Otherwise you’ll eventually end up with a monster blob of code.
- Examples: renaming of a class, field or method to something more descriptive; splitting a large method; eliminating code duplication etc.
- It is very scary (read: impossible) to do this without having tests around. Tests actually inspires you to do refactoring!

MSTest

- MSTest will scan all the files that is specified and configure itself with the attributes (e.g. **[TestMethod]**) it finds.
- Tests are usually packaged in a separate VS project and needs a reference to the other project that it is testing

MSTest attributes

- The three most common MSTest attributes are:
  - [TestClass] – on a class that contains test methods
  - [TestMethod] – defines a method as a test
  - [TestInitialize] – on a method run before each test method
- There are also once only init and cleanup for the whole test class: [ClassInitialize] [ClassCleanup]

MSTest Assert methods

- The most common is Assert.AreEqual, to check if the expected value is the one you really get
- There are a bunch of others:

```
Assert.IsTrue
Assert.IsNull
Assert.AreSame (reference equality)
Assert.IsInstanceOfType
StringAssert.Contains
StringAssert.EndsWith
StringAssert.Matches (with regular expression)
```

MSTest and exceptions

- You also want to do “rain weather” tests to check whether the appropriate exception is thrown
- Test succeeds if the correct exception is thrown

```
[TestMethod]
public void TestOverdraft()
{
Assert.ThrowsException<OverdraftException>(
() => a1.Withdraw(300);
);
}
```

MSTest summary

- It is a crucial part of any project to have automated tests.
- It is one of the fundamental tools for refactoring and thus also for enabling agile development.
- Tests should be around early (or even before developed code). Code without tests is not complete.
- It must be trivial to run tests and to interpret the result.
- The hard part is still to be sure that “everything” is covered by tests. That is why we have certified testers.

More info:

- https://www.coscreen.co/blog/tdd-in-c-guide/
- https://www.youtube.com/watch?v=e_Twc6kZymo
- https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices
- https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest

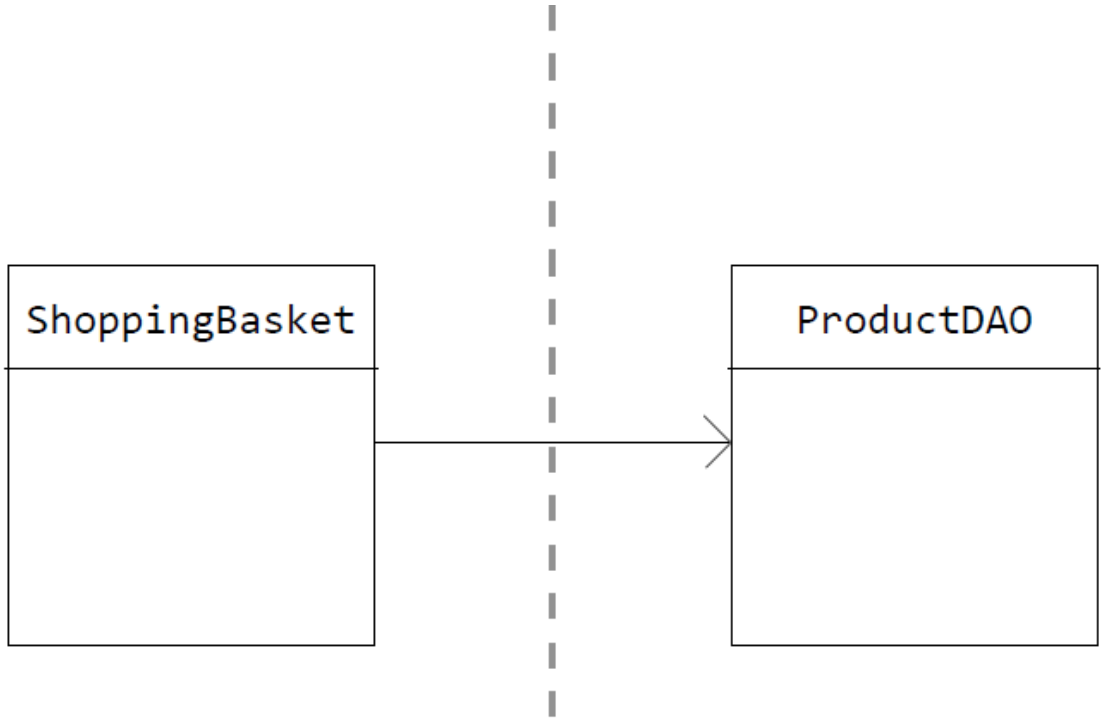
Testing with Mock Objects

Vi kommer bara att titta lite snabbt på Mocking nu och testa på det i praktiken lite senare.

Unit-testing even of a single class can be tricky

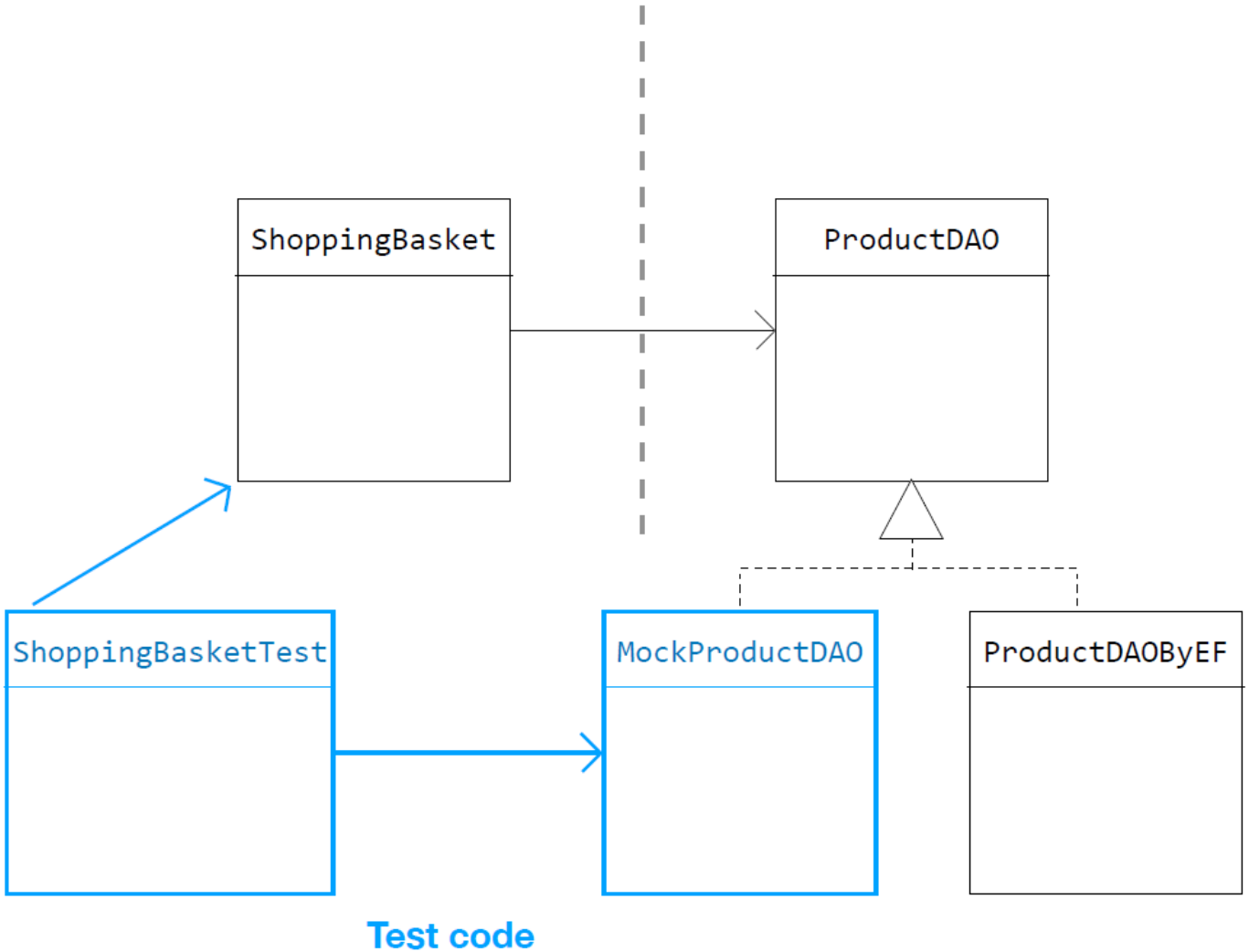
- When the class has dependency(-ies), we would like to test the object without its dependency
- But maybe the object cannot work at all without the dependency. It needs it, always.
- The tester may then supply a Mock Object that implements the dependency, but in a way that the tester controls, not the object
- This almost always requires that the object being testing gets it dependencies via Dependency Injection, so that the tester may supply the tested object with the Mock Object instead of the real production dependency.

Dependent object – example



- A ShoppingBasket needs a ProductDAO to work at all
- It cannot even be tested fully without one
- The ProductDAO normally talks to a database
- We want to test the ShoppingBasket without the database being involved

Mock the ProductDAO



Two ways to write Mock Objects

- Manual mocking
  - Manually create a class that implements the ProductDAO interface
  - All the methods must be there, but only the ones used by the ShoppingBasket during the test need meaningful code
  - Return values may me hardcoded
- Mocking framework, e.g. Moq, the most popular one for .NET
  - We do not have to create the Mock Object in a manually written class, it is created on the fly by Moq
- Both work, Moq helps a lot with the instrumentation of the Mock Object, but the framework API is a bit strange to learn

TDD

Testa på att skapa ett program helt utgående ifrån TDD. Alltså skapa ingenting utan att det finns ett test som "vill" att det finns något. Testen i sig ska komma ifrån krav.

Förslag

Testa att bygga en "TV" som beroende på vad den har för mediadevice inkopplad kommer att returnera "576p", "720p" eller "1080p".

Lägg in några tester till labben

Låt oss utföra en uppgift med dubbel nytta. Lägg in några unit tests till er labbuppgift. Läs sedan på lite mer om edge-cases inom unit testing och försök lägga in unit tests som testar för dessa!

https://en.wikipedia.org/wiki/Edge_case

Edge case

Ett edge case är när man försöker lägga in "extrempunkterna" för olika argument.

[1] Paul Hamill. *Unit Test Frameworks by*. url=https://www.oreilly.com/library/view/unit-test-frameworks/0596006896/ch05.html. 2021.

[2] Mikejo5000. *Unit Testing Fundamentals - Visual Studio (windows)*. URL: https://learn.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2022.

[3] Ian Sommerville. *Further reflections on test-driven development*. url=https://iansommerville.com/systems-software-and-technology/static/2016/03/21/further-reflections-on-test-driven-development/. 2016.

[4] Ian Sommerville. *Giving up on test-first development*. url=https://iansommerville.com/systems-software-and-technology/static/2016/03/17/giving-up-on-test-first-development/. 2016.

[5] Ian Sommerville. *Software Engineering*. Tenth. Pearson, 2016.