Unit-testing

Thomas Bøgholm

Overblik

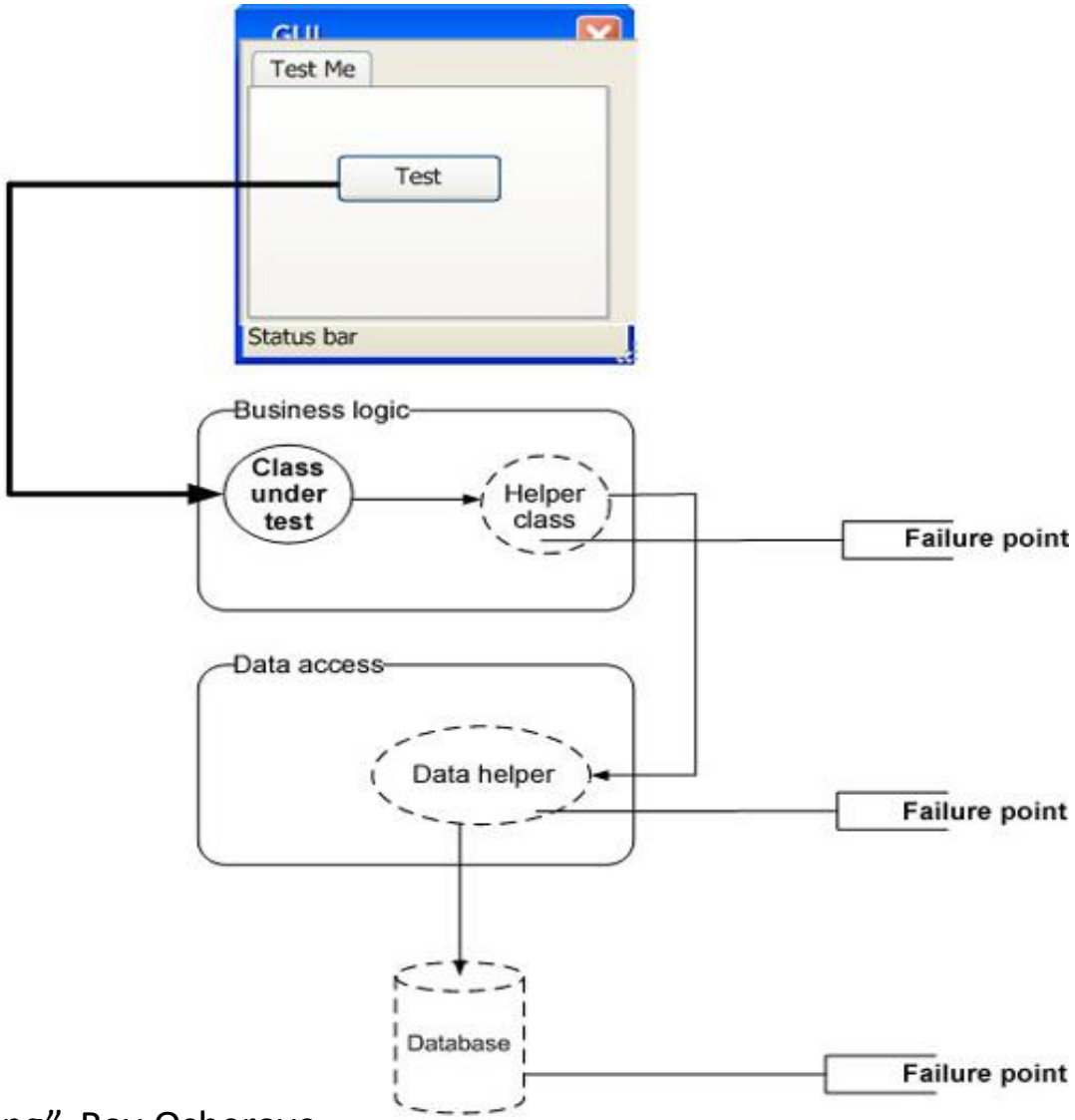
- Hvad er en god test?
- Unit Testing
- Black Box Testing
- White Box testing
- Tests med Isolations-Frameworks
- Generelle Test-Guidelines
- Test Driven Development (TDD)

Hvad er test og hvorfor teste?

- Programtest: Udførelse af et program med henblik på at finde fejl.
  - Formålet med at teste er, at sikre at et program opfører sig som *forventet*, og ikke indeholder fejl.
- En god test har høj sandsynlighed for at afdække en fejl.
  - Aftestning er dyrt – men fejl er dyrere
- Jo før vi finder fejl jo bedre. Jo senere fejl opdages, jo dyrere er de at rette.
  - Worst case: Fejl opdages efter produkt er sendt ud (tilbagekaldelse, kompensation, dårlig pr, etc.)
  - Next-worst case: Fejl kan vise sig at kræve et re-design

- *”GM recalling 4 million vehicles worldwide for software defect linked to 1 death”*

Ad-hoc testing



Fra ”The art of Unit Testing”, Roy Osherove

Systematiske tests

- Forskellige former for tests bruges på forskellige tidspunkter til forskellige ting:
  - **Unit Test**: Aftestning af programmets mindste dele i isolation.
  - **Integration Test**:
    - Delene er testet med unit tests
    - Nu testes interaktionen mellem komponenter
  - **System Test**:
    - Tester systemet som en helhed (performance, sikkerhed, GUI, etc)
  - **Acceptance Test**:
    - Aftestning via slutbruger – afviger systemet fra kundens krav?

➢ Dækkende unit tests er byggeblokken for efterfølgende tests.

http://en.wikipedia.org/wiki/System_testing

Egenskaber ved en god test

- **It should be automated and repeatable.***
- **It should be easy to implement.***
- **Once it's written, it should remain for future use.***
- **Anyone should be able to run it.***
- **It should run at the push of a button.***
- **It should run quickly.***

- Hvis nej til nogle af ovenstående, så bliver aftestning for bøvlet/dyrt

➢ Så bliver aftestning ikke gjort

➢ Så er kode ikke behørigt gennemtestet.

Unit Tests – i kombination med et unit test framework – opfylder kravene til en god test.

*The Art Of Unit Test. Roy Osherove

Hvad er unit tests?

- En unit test består af en metode der kalder en anden metode (*kode under test*) og tjekker korrektheden af en bestemt antagelse.
  - Hvis antagelsen er forkert, har unit testen fejlet
    - Fejlen rettes, og test(s) gentages

Hvad er unit tests?

- En unit test kan opdeles i tre dele:
  - **Arrange**: Data der skal bruges som input til testen klargøres.
  - **Act**: Udførelse af koden under test med data fra Arrange skridtet.
  - **Assert**: Korrektheden af antagelser for kode under test verificeres.

Eksempel på Arrange-Act-Assert

```
[TestMethod]
public void TestReverserExample()
{
    //Arrange (gør klar til test)
    string input    = "abc";
    string expected = "cba";
    Reverser r = new Reverser();

    //Act (foretag test)
    string result = r.Reverse(input);

    //Assert (tjek antagelse/krav er overholdt)
    Assert.AreEqual(expected, result);
}
```

Pass/Fail?

Unit Test frameworks

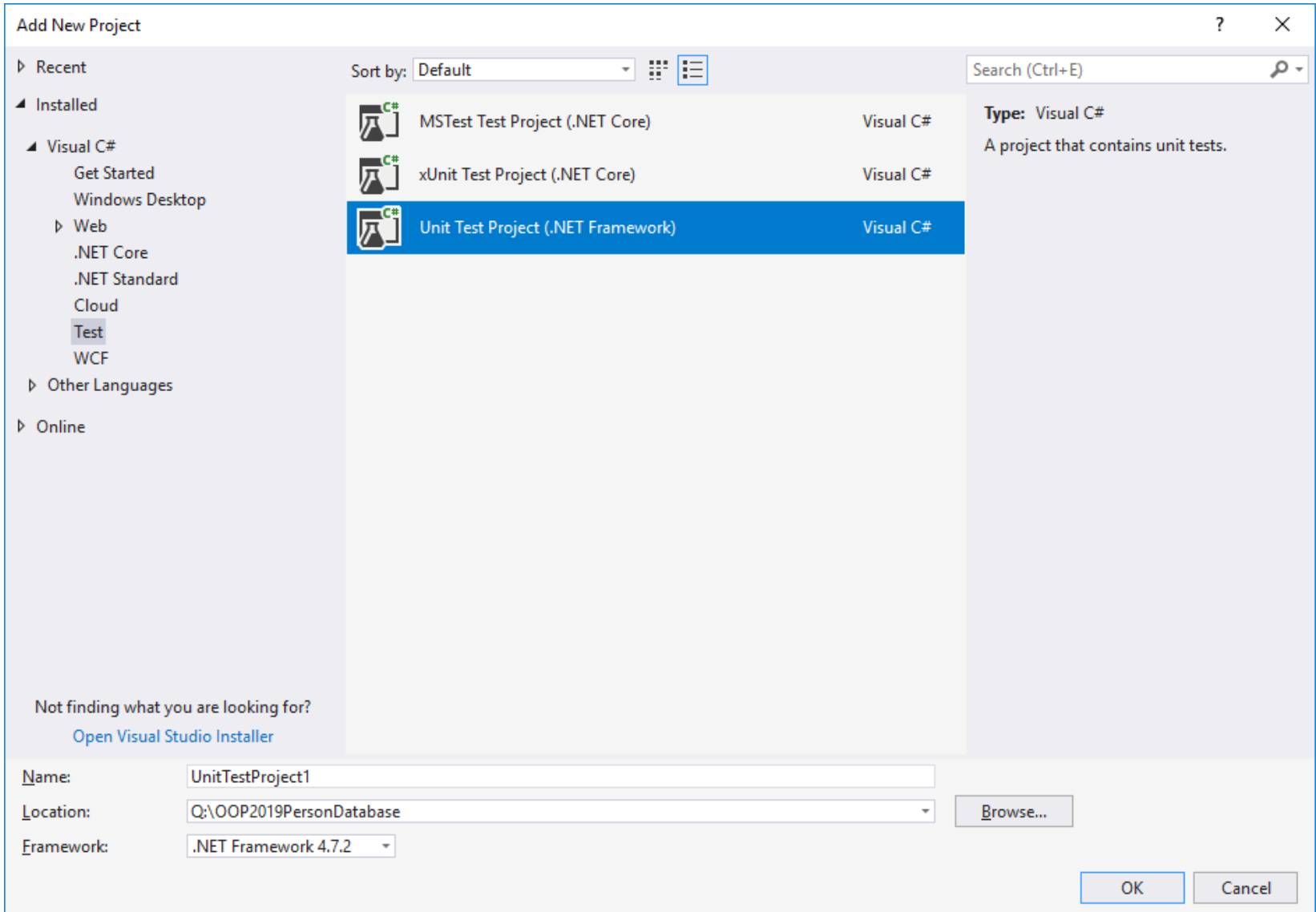
- Unit tests foretages ved at bruge et (eksisterende) test framework.
  - I princippet *kunne* vi lave vores eget test framework, men der er ingen grund til at genopfinde hjulet
- Unit test frameworks ***automatiserer*** tests.
  - Indeholder runners der kan udføre én, flere, eller alle tests på én gang.
  - Indeholder *assertions* til at validere forventet opførsel

➢Unit testing frameworks letter processen med at skrive tests.

- NUnit
- xUnit
- MSTest

Unit test framework i VS

```
namespace OOP2020.UtilApps
{
    public class Calculator
    {
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

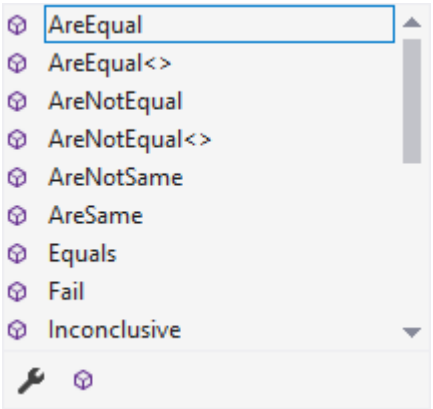


Testklasse/metoder

```
namespace OOP2020.UtilApps.Tests
{
    [TestClass]
    public class CalculatorTests
    {
        [TestMethod]
        public void Add_ReturnsCorrectSum()
        {
            Calculator calculator = new Calculator();
            int inputa = 1;
            int inputb = 10;
            int expected = 11;

            int result = calculator.Add(inputa, inputb);

            Assert.AreEqual(expected, result);
        }
    }
}
```

Testklasse/metoder

```
namespace OOP2020.UtilApps.Tests
{
    [TestClass]
    public class CalculatorTests
    {
        [TestMethod]
        public void Add_ReturnsCorrectSum()
        {
            Calculator calculator = new Calculator();
            int inputa = 1;
            int inputb = 10;
            int expected = 11;

            int result = calculator.Add(inputa, inputb);

            Assert.AreEqual(expected, result);
        }
    }
}
```



Exceptions

```
[TestClass]
public class PersonTests
{
    [TestMethod]
    [ExpectedException(typeof(ArgumentException), "No exception thrown on null values")]
    public void Constructor_WhenCalledWithNullFirstname_ThrowsInvalidArgumentExceptions()
    {
        //arrange
        string firstnameparam = null;
        string lastnameparam = "Bøgholm";

        //act
        Person p = new Person(firstnameparam, lastnameparam);
    }
}
```

”Regler” for test cases(1)

UNDGÅ fejl i testkode!

- En test case skal validere én ting
  - Hvis én assert fejler i en metode udføres resten ikke.
  - Flere asserts betyder flere tests
- En test skal være simpel og læsbar
  - Undgå at introducere bugs i testen.
  - Betinget logik er ”forbudt” -> split op i flere tests
- Test cases skal udføres i isolation
  - Samme tilstand før hver udførelse af test.
  - Test cases skal kunne udføres i vilkårlig rækkefølge

”Regler” for test cases(2)

- Test kun offentlige medlemmer
  - Tests bliver mindre skrøbelige
  - Private medlemmer testes indirekte gennem offentlige medlemmer
- Don’t Repeat Yourself (DRY)
  - Placer fælles Arrange i metoder (InitializeXXX)

Opbygning af testcases

Brug en navnekonvention

Target	**Test**
Projekt	Lav projekt kaldet [Projekt].Tests. - MySuperProject -> MySuperProject.Tests
Klasse	Lav klasse kaldet [Klasse]Tests. - Car -> CarTests
Metode	Lav (minimum én) metode kaldet [Metode]_[Betingelse]_[Forventning]. Fx. isStrongPassword_strongPass_ReturnsTrue IsStrongPassword_weakPass_ReturnsFalse

Hvad skal der testes efter?

- Forventet opførsel ved forskelligt input.
  - Hvordan håndteres lovligt input?
  - Hvordan håndteres ulovligt input

➢Flere test cases per metode under test

- Kræver en specifikation.

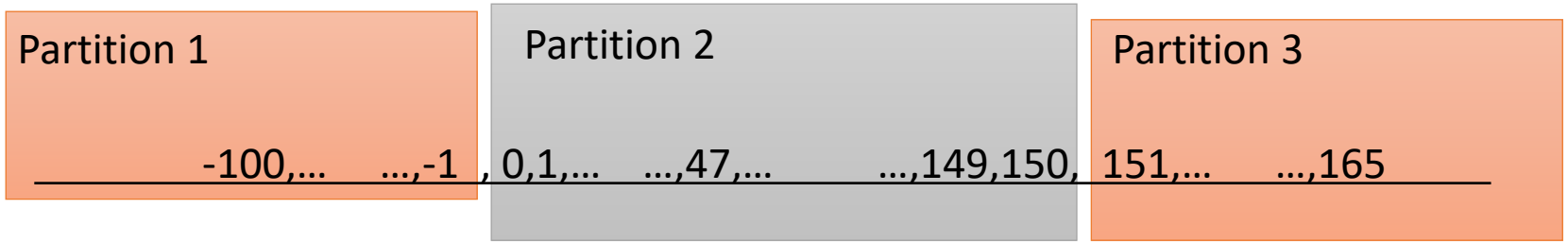
Black Box Testing

- **Black Box Testing**: Drevet af funktionelle krav. Givet bestemt input skal systemet give bestemt output.
- Udtømmende kombinationer af input værdier er ikke muligt (2^32 forskellige ints, et ubegrænset antal strings, osv.)
- **Mål**: En test skal maksimere antallet af fejl der findes med et endeligt(minimum) antal test cases
- Hvordan finder vi vores endelige antal test cases?
  - Analyse af input/output domænet

Input/Output analyse

- Baseret på *ækvivalens partitionering*: Input deles ind i et antal klasser (partitioner) hvor følgende antages:
  - Programmet opfører sig ens for input fra samme klasse (all fail – all pass)
  - En test med en værdi fra hver klasse er tilstrækkeligt
- Ækvivalens-klasser kan afgøres fra specifikation
  - F.eks. ”alder skal være mellem 0 og 150”
- Mere effektivt er, at inkludere værdier *på*, *lige over*, og *lige under* grænserne ->

Alder skal være mellem 0 og 150



	Ækvivalens klasse	Værdi	Bemærkning
1	#1 (x < 0)	-100	Tilfældigt tal under nedre grænse
2	#1 (x < 0)	-1	Lige under nedre grænse
3	#2 (0 <= x <=150)	0	På nedre grænse
4	#2 (0 <= x <=150)	1	Lige over nedre grænse
5	#2 (0 <= x <=150)	47	Tilfældig tal imellem
6	#2 (0 <= x <=150)	149	Lige under øvre grænse
7	#2 (0 <= x <=150)	150	På øvre grænse
8	#3 (x >150)	151	Lige over øvre grænse
9	#3 (x > 150)	165	Tilfældig tal over øvre grænse

Fordele ved unit testing

- Unit tests kan bruge til at angive *funktionelle krav*: Givet at input overholder en *precondition*, så tjekkes at output overholder en *postcondition*.
- Kræver en specifikation:
  - Hvad udgør lovligt input for en metode? Hvad er forventet opførsel?
  - Hvad udgør ulovligt input for en metode? Hvad er forventet opførsel?
- Unit tests kan bruges til at *dokumentere* korrekt program-opførsel.
- En test-suite med høj kode-dækning og mange assertions indikerer kode af god kvalitet.
  - Kort feedback loop – fejl opdages og rettes hurtigt.
- Unit tests fungerer som *regressions-tests*. Når vi ændrer i koden, fungerer det så stadig?

Ulemper ved unit testing

- Kan være trivielt og tidskrævende
- Kvaliteten af unit tests afhænger af kvaliteten af unit tests (resultat af den investerede tid samt testernes erfaring)
  - **Kvalitet afhænger af kode dækning og antal assertions** (kode-dækning er ikke i sig selv en garanti, men lav kode-dækning indikerer en risiko)
  - Unit tests kan indeholde bugs + være svære at vedligeholde.

Fjernelse af eksterne afhængigheder med Stubs

- En ekstern afhængighed er noget uden for vores kontrol
  - database interaktion, fil-operationer, web service kald, ikke-implementeret kode, …)
- Vi vil gerne fjerne eksterne afhængigheder, så vores tests kan køre i isolation.
- Til formålet kan man bruge en **stub**: En kontrollérbar erstatning for en ekstern afhængighed.
- Fremgangsmåde:
  1. Lav interface for den eksterne afhængighed
  2. Substituér ekstern afhængighed for stub vha. dependency injection.
- To muligheder for stubs
  - Lav den i hånden
  - Brug et Isolation/ Mocking framework (f.eks. NSubstitute – bruges til følgende eksempel)

Eksempel: Persistenslag

- Det er god skik at indkapsle persistens operationer i et **persistenslag** (aka **Data Access Layer (DAL)**).
- Herved undgår man afhængighed af én bestemt måde at persistere data på.

- Lav et DAL interface der angiver mulige persistens operationer (CRUD) for data i modellaget.
- Lav derefter konkrete DAL klasser der indeholder kode til at gemme/hente fra specifikke underliggende data kilder (DB, XML, Web Service, …).
- Til aftestning: Lav stub der implementerer DAL interfacet

Ovenstående definition af et DAL refererer til det såkaldte Repository mønster (http://msdn.microsoft.com/en-us/library/ff649690.aspx)

#1: Lav DAL interface

```
interface IKundeDAL
{
    bool Create(Kunde nyKunde);
    bool Update(Kunde eksisterendeKunde);
    bool Delete(Kunde eksisterendeKunde);
    //Hent alle kunder
    IEnumerable<Kunde> ReadAll();
    //Hent kunder der opfylder prædikat
    IEnumerable<Kunde> Read(Func<Kunde, bool> filter);
    //Hent den enkelte kunde der opfylder prædikatet
    Kunde ReadSingle(Func<Kunde, bool> filter);
}
```

Her kunne man f.eks. have en konkret Entity Framework implementation: **KundeEfDAL**

Kode med ekstern afhængighed

- En klasse der interagerer med DAL

```
public class KundeAdministrationController
{
    IKundeDAL _kundeDAL; //Data Access Layer - vores eksterne afhængighed
    List<Kunde> _lokaleKunder; //Lokalt indlæste kunder

    public KundeAdministrationController()
        : this(new KundeEfDAL()) { }

    public KundeAdministrationController(IKundeDAL kundeDAL) {
        this._kundeDAL = kundeDAL;
        _lokaleKunder = new List<Kunde>(_kundeDAL.ReadAll());
    }


    //Returnerer hvorvidt kunden blev oprettet
    public bool OpretKunde(Kunde k) {}
    //Flere metoder…
}
```

Dependency injection for at kunne Subsistuere normal DAL med stub

Vi vil gerne teste denne metode af i isolation (uden at have en konkret database på plads)

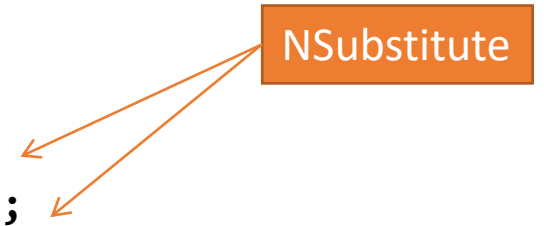
Kode under test – med ekstern afhængighed

```
//Returnerer hvorvidt kunden blev oprettet
public bool OpretKunde(Kunde k)
{
    bool oprettet = false;
    if (k != null && !_lokaleKunder.Contains(k))
    {
        try
        {
            //Create() kan finde på at smide KundeAlreadyExistsException
            if (_kundeDAL.Create(k))
            {
                _lokaleKunder.Add(k);
                oprettet = true;
            }
        }
        catch (KundeAlreadyExistsException) { }
    }

    return oprettet;
}
```

Test #1

- Her bruger vi blot stub til at komme i gang - undgå at lave instans af EF + kald til DAL.ReadAll()
- Selve koden under test interagerer ikke med ekstern afhængighed

```
[TestFixture]
public class KundeAdministrationControllerTests
{
    [Test]
    public void OpretKunde_KundeErNull_ReturnsFalse()
    {
        //Arrange
        IKundeDAL kundeDAL = Substitute.For<IKundeDAL>();
        kundeDAL.ReadAll().Returns(new List<Kunde>());
        KundeAdministrationController controller =
                new KundeAdministrationController(kundeDAL);

        //Act
        bool kundeOprettet = controller.OpretKunde(null);
        //Assert
        Assert.IsFalse(kundeOprettet);
    }
    //...
```



Test #2: Opret kunde. Kunde findes ikke lokalt, men er oprettet i databasen andetsteds i mellemtiden

```
[Test]
public void OpretKunde_EksisterendeKunde_ReturnsFalse()
{
    //Arrange
    Kunde eksisterendeKunde =
         new Kunde() { KundeNummer = "1234", Navn = "Ida" };

    //Setup af mocking framework
    IKundeDAL kundeDAL = Substitute.For<IKundeDAL>();
    kundeDAL.ReadAll().Returns(new List<Kunde>()); //Ikke vigtigt
    kundeDAL.Create(eksisterendeKunde).Returns(
        k => { throw new KundeAlreadyExistsException(); });

    //Setup af kode under test
    KundeAdministrationController controller =
        new KundeAdministrationController(kundeDAL);

    //Act
    bool kundeOprettet = controller.OpretKunde(eksisterendeKunde);
    //Assert
    Assert.IsFalse(kundeOprettet);
}
```

stub-setup

Test #3: Test succesful oprettelse

```
[Test]
public void OpretKunde_NyKunde_ReturnsTrue()
{
    //Arrange
    //Setup af mocking framework
    IKundeDAL kundeDAL = Substitute.For<IKundeDAL>();
    kundeDAL.ReadAll().Returns(new List<Kunde>());
    kundeDAL.Create(Arg.Any<Kunde>()).Returns(k => true );

    //Setup af kode under test
    KundeAdministrationController controller =
        new KundeAdministrationController(kundeDAL);
    Kunde nyKunde = new Kunde() { KundeNummer = "1234", Navn = "Ida" };

    //Act
    bool kundeOprettet = controller.OpretKunde(nyKunde);
    //Assert
    Assert.IsTrue(kundeOprettet);
}
```

Interaktionstests med mocks

- Når vi er interesseret i at teste selve *interaktionen* med en ekstern afhængighed, så kan vi bruge en **mock**.
  - Eksempel: Vi vil sikre os at vi fik kaldt en bestemt metode med nogle bestemte parametre på den eksterne afhængighed.
- Forskel på stub og mock:
  - Ved stubs: Tester på kendt kendt(fake) afhængighed
  - Ved mocks: Tester forventet interaktion med fake
    - Fake-afhængighed logger interaktion
- Brug kun interaktionstests hvis det er bydende nødvendigt.

Eksempel på mocking:

```
[Test]
public void OpretKunde_NyKunde_KalderDAL_Create()
{
    //Arrange
    //Setup af mocking framework
    IKundeDAL kundeDAL = Substitute.For<IKundeDAL>();
    kundeDAL.ReadAll().Returns(new List<Kunde>()); //Ikke vigtigt
    kundeDAL.Create(Arg.Any<Kunde>()).Returns(k => true);

    //Setup af kode under test
    KundeAdministrationController controller = new KundeAdministrationController(kundeDAL);
    Kunde nyKunde = new Kunde() { KundeNummer = "1234", Navn = "Ida" };

    //Act
    bool kundeOprettet = controller.OpretKunde(nyKunde);
    //Assert
    //I forhold til test #3 (med stub), så laver vi nu assert på den eksterne
    //afhængighed – og ikke på tilstanden koden under test.
    kundeDAL.Received().Create(nyKunde);
}
```

Fordele ved unit testing

- Unit tests kan bruge til at angive *funktionelle krav*: Givet at input overholder en *precondition*, så tjekkes at output overholder en *postcondition*.
- Kræver en specifikation:
  - Hvad udgør lovligt input for en metode? Hvad er forventet opførsel?
  - Hvad udgør ulovligt input for en metode? Hvad er forventet opførsel?
- Unit tests kan bruges til at *dokumentere* korrekt program-opførsel.
- En test-suite med høj kode-dækning og mange assertions indikerer kode af god kvalitet.
  - Kort feedback loop – fejl opdages og rettes hurtigt.
- Unit tests fungerer som *regressions-tests*. Når vi ændrer i koden, fungerer det så stadig?

Ulemper ved unit testing

- Kan være trivielt og tidskrævende
- Kvaliteten af unit tests afhænger af kvaliteten af unit tests (resultat af den investerede tid samt testernes erfaring)
  - **Kvalitet afhænger af kode dækning og antal assertions** (kode-dækning er ikke i sig selv en garanti, men lav kode-dækning indikerer en risiko)
  - Unit tests kan indeholde bugs + være svære at vedligeholde.

Code coverage

- Selv hvis alle tests består, er det en ringe garanti for fravær af fejl, hvis vi kun har testet halvdelen af koden
- Kode-dækning finder områder i koden der ikke er dækket af tests
- Kode-dækning er indirekte et kvalitetskriterie
  - manglende dækning indikerer lav troværdighed
- Kode-dækning bygger på antagelsen om at mange fejl er relateret til control flow
- Kan udtrykkes med linier, blokke, metoder …
- Whiteboxtilgang

Test-Driven Development

En udviklingsmetode mere end en testmetode

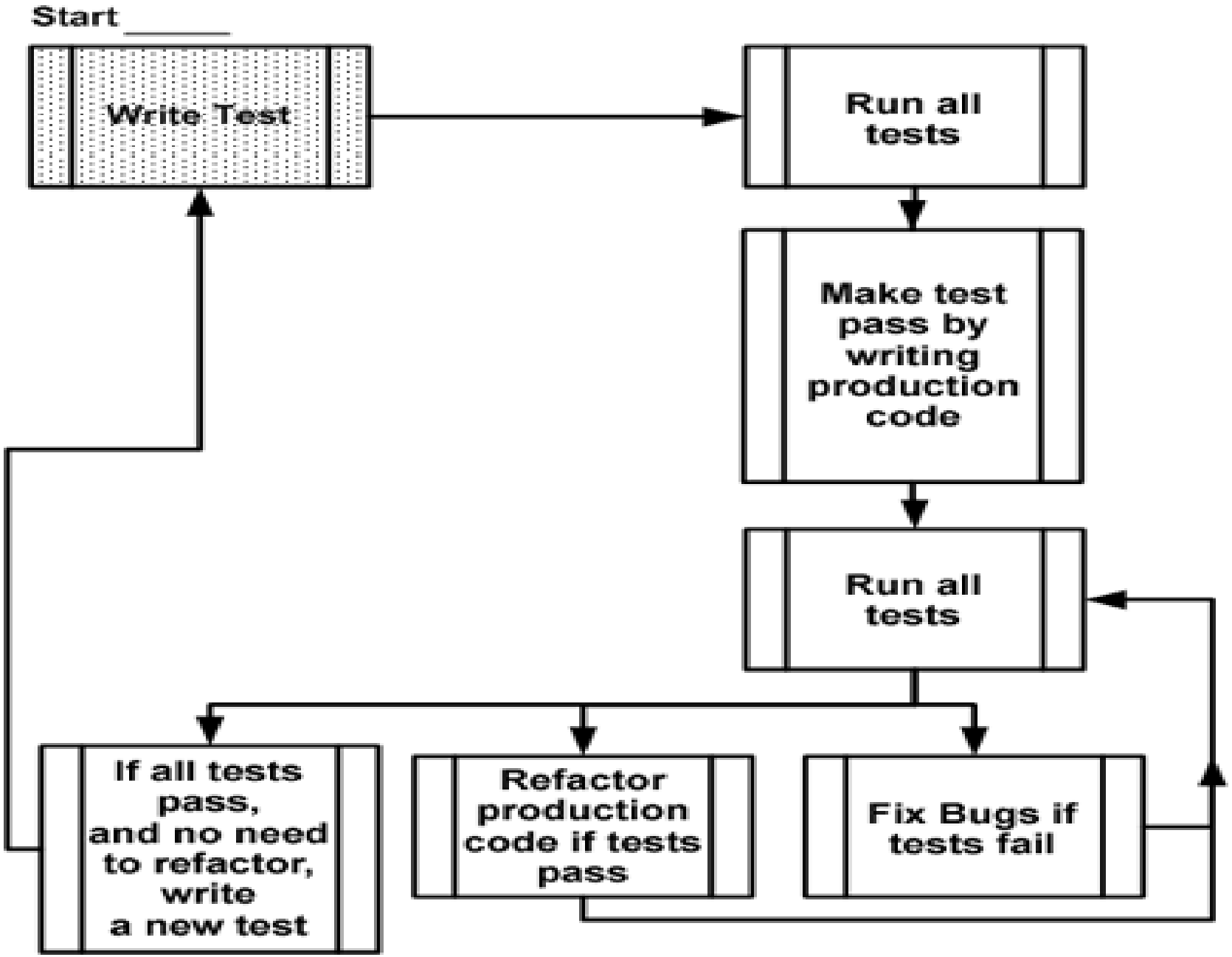
Brug unit-tests til at drive implementation

Bonus: unit-tests er gratis!

TDD - fremgangsmåde

- Tilføj en test
- Kør alle tests og se rødt (fail)
- Lav en ændring
- Kør alle tests og se grønt (pass)
- Refaktorér koden
- Kør alle tests

Test-Driven Development (TDD)



The Three Laws of TDD

- Robert C Martin

First Law: You may not write production code until you have written a failing unit test.

Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Third Law: You may not write more production code than is sufficient to pass the currently failing test.

Robert C. Martin: SOLID, Clean code, Clean architecture, … -- http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

Opsummering

- Unit Testing
  - Formål: Teste kodens mindste bestanddele inden vi begynder med integrationstest.
  - Separat testprojekt der tester production kode (CUT)
  - Arrange-Act-Assert
- Black Box Testing
  - Generation af tests ud fra funktionel specifikation
  - Analyse af input værdier
- White Box testing
  - Inspektion af programmets interne struktur til at generere yderligere tests
  - Sikrer at vi har høj kodedækning
- Test Driven Development (TDD)
  - Udviklingsmetode med udgangspunkt i unit tests.