

Software Test Eksamen

Banksystem

Mathias Schjødt-Bavngaard



Figur 1 - <https://www.vecteezy.com/vector-art/2126283-bank-system-banner-with-icons-and-customers>

Indhold

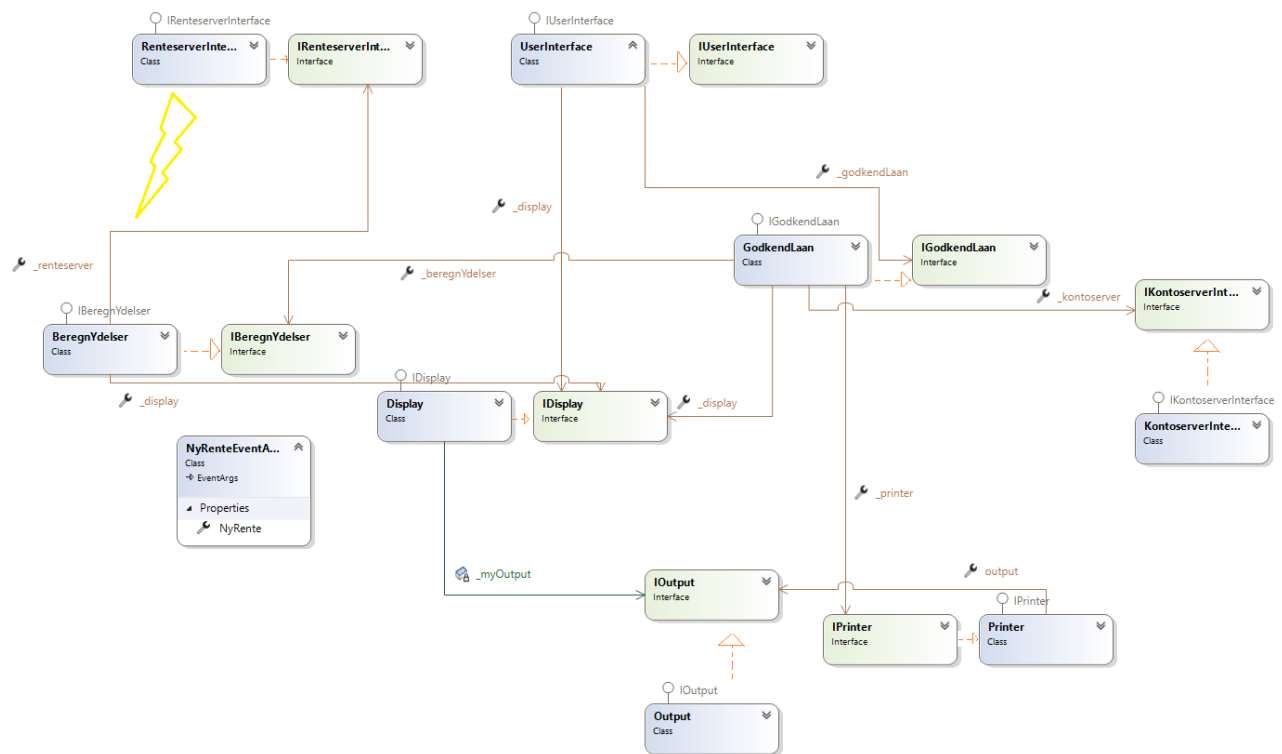
1	Design System.....	3
1.1	Class Overview.....	4
1.2	Implementeringstanker	5
2	UnitTest – Delopgave 3,4 og 6.....	5
2.1	BlackBox Testing.....	6
2.2	WhiteBox Testing.....	7
2.2.1	Event Testing.	7
3	Coverage.....	8
4	SWT Facts to talk about.....	9
4.1	Unit Test vs IntergrationsTest	9
4.2	Dynamisk analyse af Software.....	9
4.3	Projekt WorkFlow – CI yamI	10

1 DESIGN SYSTEM

Ud fra figur 3, 4 og figur 5, samt beskrivelserne her til udlagt i opgaven, er et Klasse diagram udarbejdet (Figur 2).

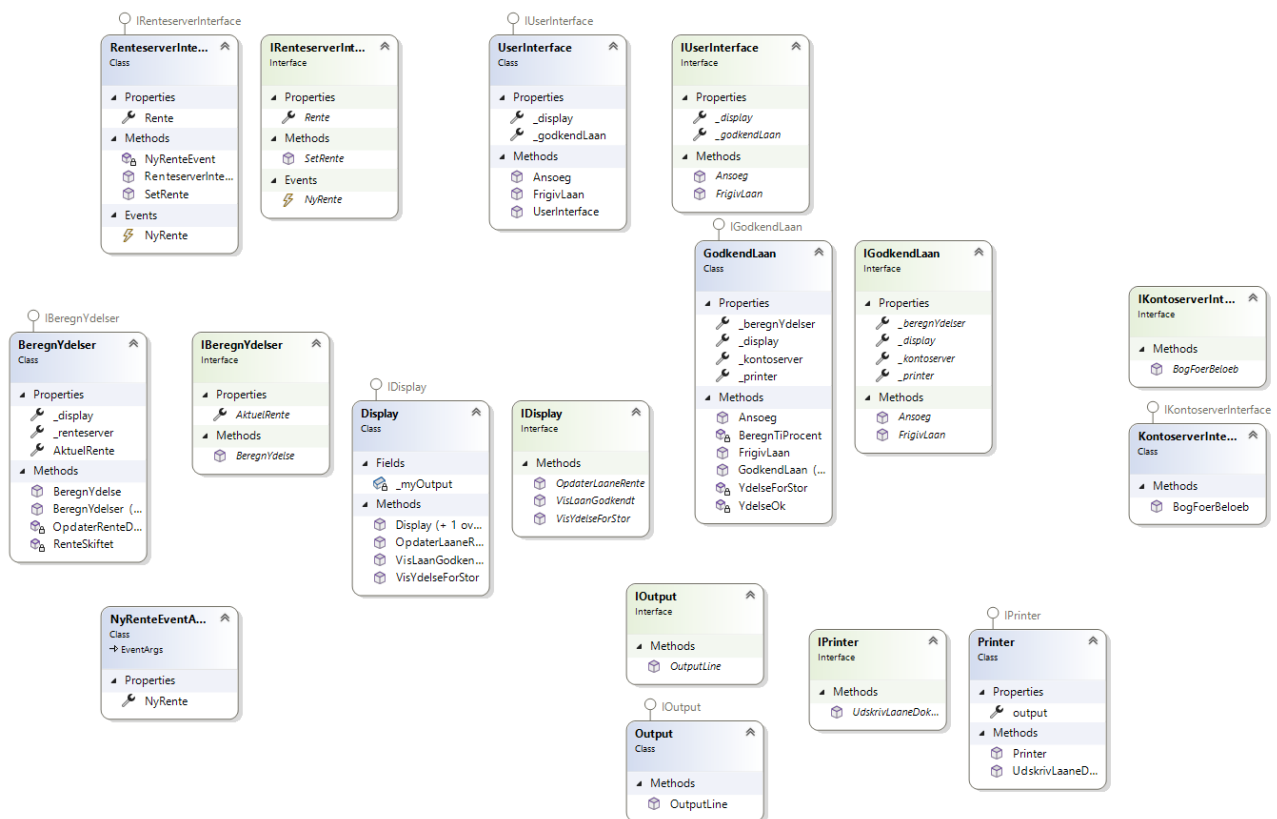
Ved at Seperarer de konkrete klasser med Interfaces opnås et testbart design.

Alle klasser implementere et interface og klasserne følger SingleResponserbility hvilket gør at det er nemt at teste Boundery values for de forskellige funktionaloteter.Figur 2



Figur 2 Class Diagram Overview With Assosiations

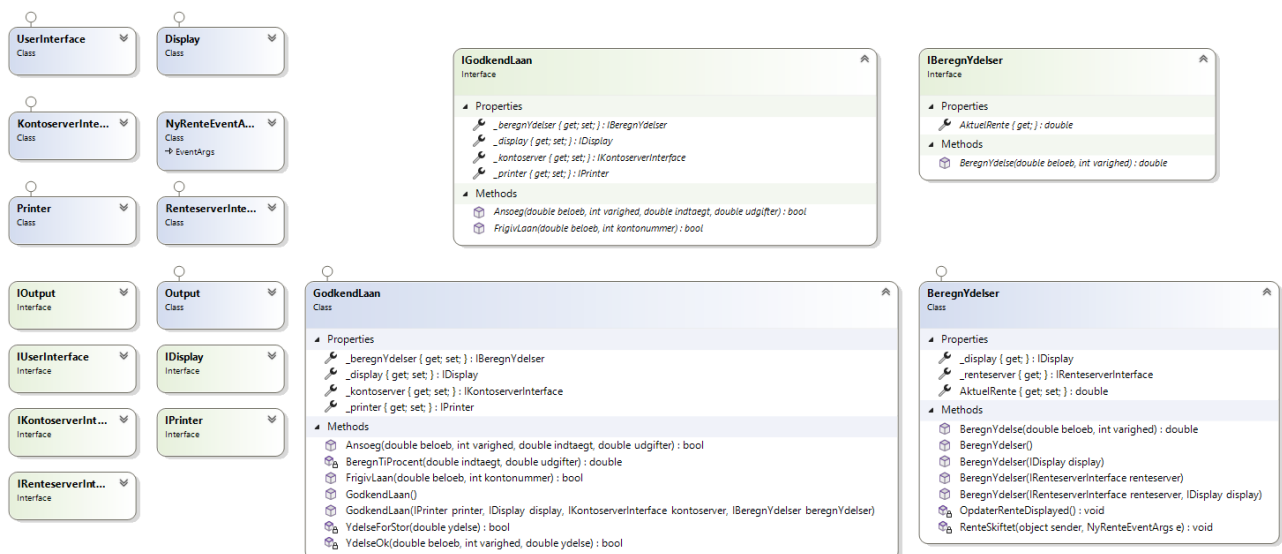
Som det kan ses på (Figur 3) har Nogle af de konkrete klasser Flere funktioner end interfaces, dette er for at give mere Separation of Concern.



Figur 3 Class Diagram with All Functions and Properties and none Assosiation Arrows

1.1 CLASS OVERVIEW

For at gøre Testning nemmere er der lavet Propertie injektion. På to forskellige måder: Først ved at lave Default constructors. Og så have public Properties til Dependencies.



Figur 4 Class Diagram With the two most important Klasses Fully Detailed-

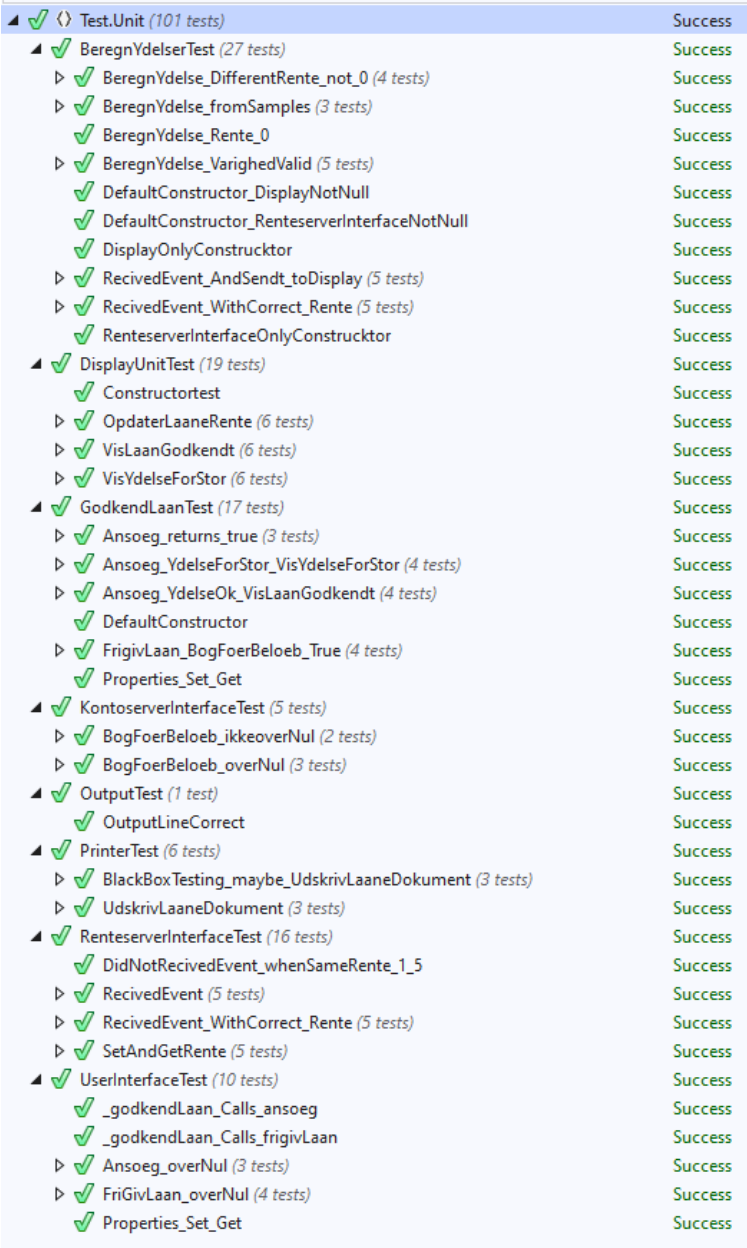
Der er efterfølgende lavet Constructors med parametre til alle dependencies. Dette kan fx ses på (Figur 4) Hvor de to Hovedklasser er vist i form af klasse digram med fuld signatur detalie.

1.2 IMPLEMENTERINGSTANKER

For at implementere klassen GodkendLån og færdiggøre BeregnYdelser med udgangspunkt i mit testbare design, satte jeg klasserne op til at have interfaces. De klasser som er med til at udfører nogle af de beskrevne funktioner er både Defineret som interfaces men som en lille Bonus er de også alle implementeret til at håndtere de krævede funktionalitet. Der er også unit Test til Alle disse klasser. Se (Figur 5) for Test overblik. I forhold til Sekvens Diagrammet er der ingen ændringer.

2 UNITTEST – DELOPGAVE 3,4 OG 6

Da der er lavet 8 klasser giver det mening at have en UnitTestsuite for hver af disse klasser. Figur 5 viser



Test.Unit (101 tests)	Success
BeregnYdelserTest (27 tests)	Success
BeregnYdelse_DifferentRente_not_0 (4 tests)	Success
BeregnYdelse_fromSamples (3 tests)	Success
BeregnYdelse_Rente_0	Success
BeregnYdelse_VarighedValid (5 tests)	Success
DefaultConstructor_DisplayNotNull	Success
DefaultConstructor_RenteserverInterfaceNotNull	Success
DisplayOnlyConstructor	Success
RecivedEvent_AndSendt_toDisplay (5 tests)	Success
RecivedEvent_WithCorrect_Rente (5 tests)	Success
RenteserverInterfaceOnlyConstructor	Success
DisplayUnitTest (19 tests)	Success
Constructortest	Success
OpdaterLaaneRente (6 tests)	Success
VisLaanGodkendt (6 tests)	Success
VisYdelseForStor (6 tests)	Success
GodkendLaanTest (17 tests)	Success
Ansoeg_returns_true (3 tests)	Success
Ansoeg_YdelseForStor_VisYdelseForStor (4 tests)	Success
Ansoeg_YdelseOk_VisLaanGodkendt (4 tests)	Success
DefaultConstructor	Success
FrigivLaan_BogFoerBeloeb_True (4 tests)	Success
Properties_Set_Get	Success
KontoserverInterfaceTest (5 tests)	Success
BogFoerBeloeb_ikkeoverNul (2 tests)	Success
BogFoerBeloeb_overNul (3 tests)	Success
OutputTest (1 test)	Success
OutputLineCorrect	Success
PrinterTest (6 tests)	Success
BlackBoxTesting_maybe_UdskrivLaaneDokument (3 tests)	Success
UdskrivLaaneDokument (3 tests)	Success
RenteserverInterfaceTest (16 tests)	Success
DidNotRecivedEvent_whenSameRente_1_5	Success
RecivedEvent (5 tests)	Success
RecivedEvent_WithCorrect_Rente (5 tests)	Success
SetAndGetRente (5 tests)	Success
UserInterfaceTest (10 tests)	Success
_godkendLaan_Calls_ansoeg	Success
_godkendLaan_Calls_frigivLaan	Success
Ansoeg_overNul (3 tests)	Success
FrigivLaan_overNul (4 tests)	Success
Properties_Set_Get	Success

Figur 5 Unit Test Suites Resultater.

Resultaterne af alle TestSuites og de er alle Succesfulde.

Der er blevet testet for at Funktioner returnere det forventede resultat på forskellige inputs, Her nærmer vi os Black box Testing, fordi vi kun kigger på input og output.

2.1 BLACKBOX TESTING

Fx ved test af KontoServerInterface's bogførbeløb (Figur 6). Her sender vi et input ind uden at se på andet end outputtet.

```
[TestCase(10000)]
[TestCase(50000)]
[TestCase(100000)]
• | 0 references | MathiasSchjoedt-Bavngaard, 7 hours ago | 1 author, 1 change
public void BogFoerBeloeb_overNul(double beloeb)
{
    Assert.That(_uut.BogFoerBeloeb(beloeb, kontoNummer: 1), Is.EqualTo(true));
}

[TestCase(0)]
[TestCase(-10000)]
• | 0 references | MathiasSchjoedt-Bavngaard, 7 hours ago | 1 author, 1 change
public void BogFoerBeloeb_ikkeoverNul(double beloeb)
{
    Assert.That(_uut.BogFoerBeloeb(beloeb, kontoNummer: 1), Is.EqualTo(false));
}
```

Figur 6 BlackBox Testing of KontoServerInterface

I GodkendLån testen nærmer vi os også en Blackboxtest da vi kigger på Ansøg med værdier vi ved kommer

```
[TestCase(beloeb: 10000, rente: 0.015, varighed: 60, ydelse: 253.93)]
[TestCase(beloeb: 50000, rente: 0.0025, varighed: 120, ydelse: 482.80)]
[TestCase(beloeb: 100000, rente: 0.005, varighed: 120, ydelse: 1110.21)]
• | 0 references | 0 changes | 0 authors, 0 changes
public void Ansog_returns_true(double beloeb, double rente, int varighed, double ydelse)
{
    _uut._beregnydelser.AktuelRente.Returns(rente);
    _uut._beregnydelser.Beregnydelse(beloeb, varighed).Returns(ydelse);
    Assert.That(_uut.Ansog(beloeb, varighed, indtaegt: 50000, udgifter: 35400), Is.True);
}
```

Figur 7 Ansøg returns true on valid request

til at være rigtige og kigger på at der returneres True.

Det er ikke helt en Blackbox test da vi stadig bliver nødt til at kigge ind under kølerhjelen for at sætte de rigtige Nsubstitut Returns...

En anden Test er det program der er skrevet i Program.cs fra bilaget.

Her bruges kun Userinterface til at bruge klasserne og Programmet. Så bruges Consollen til at verificere om Programmet virker.

2.2 WHITEBOX TESTING

De fleste af Unit Testne er WhiteBox Test. Der er både test som går ind og kigger på Behavior ved hjælp af Nsubstitut og de interfaces de er bygget op på. Og så er der test som kun kigger på resultatater af funktioner.

I de tilfælde hvor der også er flere konstruktors, testes de forskellige tilfælde også. Bla Propertie injektion som ses på (Figur 8)

```
[Test]
public void Properties_Set_Get()
{
    _ uut = new GodkendLaan();
    _ uut._beregnydelser = _beregnydelser;
    _ uut._display = _display;
    _ uut._kontoserver = _kontoserver;
    _ uut._printer = _printer;

    Assert.That(_ uut._beregnydelser, Is.EqualTo(_beregnydelser));
    Assert.That(_ uut._display, Is.EqualTo(_display));
    Assert.That(_ uut._kontoserver, Is.EqualTo(_kontoserver));
    Assert.That(_ uut._printer, Is.EqualTo(_printer));
}
```

Figur 8 Property set and get for GodkendLån

2.2.1 Event Testing.

For at Teste det observer pattern som er implementeret mellem RenteseverInterface og BeregnYdelser er der lavet test i begge Klassers UnitTest Suite. På (Figur 9) kan testen for "Observeren" BeregnYdelser ses. Her bruges Nsubstitute til at Raise event og der Assertes så på om De rigtige værdier kommer ind og om display bliver ændret.

```
[TestCase(0.05)] [TestCase(0.10)] [TestCase(0.005)] [TestCase(0)] [TestCase(-0.05)]
public void RecivedEvent_WithCorrect_Rente(double rente)
{
    _ renteseverInterface.NyRente += Raise.EventWith(new NyRenteEventArgs { NyRente = rente });
    Assert.That(_ uut.AktuelRente, Is.EqualTo(rente));
}

[TestCase(0.05)] [TestCase(0.10)] [TestCase(0.005)] [TestCase(0)] [TestCase(-0.05)]
public void RecivedEvent_AndSendt_toDisplay(double rente)
{
    _ renteseverInterface.NyRente += Raise.EventWith(new NyRenteEventArgs { NyRente = rente });
    _ uut._display.Received(1).OpdaterLaaneRente(rente);
}
```

Figur 9 RecivedEvent in BeregnYdelser

På (Figur 10) kan De Event test ses som er i "Subjekt" RenteServerInterface. Da der er tale om et push pattern tjekker vi også for om den rigtige værdi er sendt med i eventet-.

```
[SetUp]
public void Setup()
{
    _receivedNyRenteEvent = null;
    _uut = new RenteserverInterface();
    _uut.NyRente +=
        (O:object?, args) =>
        {
            _receivedNyRenteEvent = args;
        };
}

[TestCase(0.05)] [TestCase(0.10)] [TestCase(0.005)] [TestCase(0)] [TestCase(-0.05)]
public void RecivedEvent(double rente)
{
    _uut.SetRente(rente);
    Assert.That(_receivedNyRenteEvent, Is.Not.Null);
}

[TestCase(0.05)] [TestCase(0.10)] [TestCase(0.005)] [TestCase(0)] [TestCase(-0.05)]
public void RecivedEvent_WithCorrect_Rente(double rente)
{
    _uut.SetRente(rente);
    Assert.That(_receivedNyRenteEvent.NyRente, Is.EqualTo(rente));
}

[Test]
public void DidNotRecivedEvent_whenSameRente_1_5()
{
    _uut.SetRente(0.015);
    Assert.That(_receivedNyRenteEvent, Is.Null);
}
```

Figur 10 Send Event From RenteServerInterface.

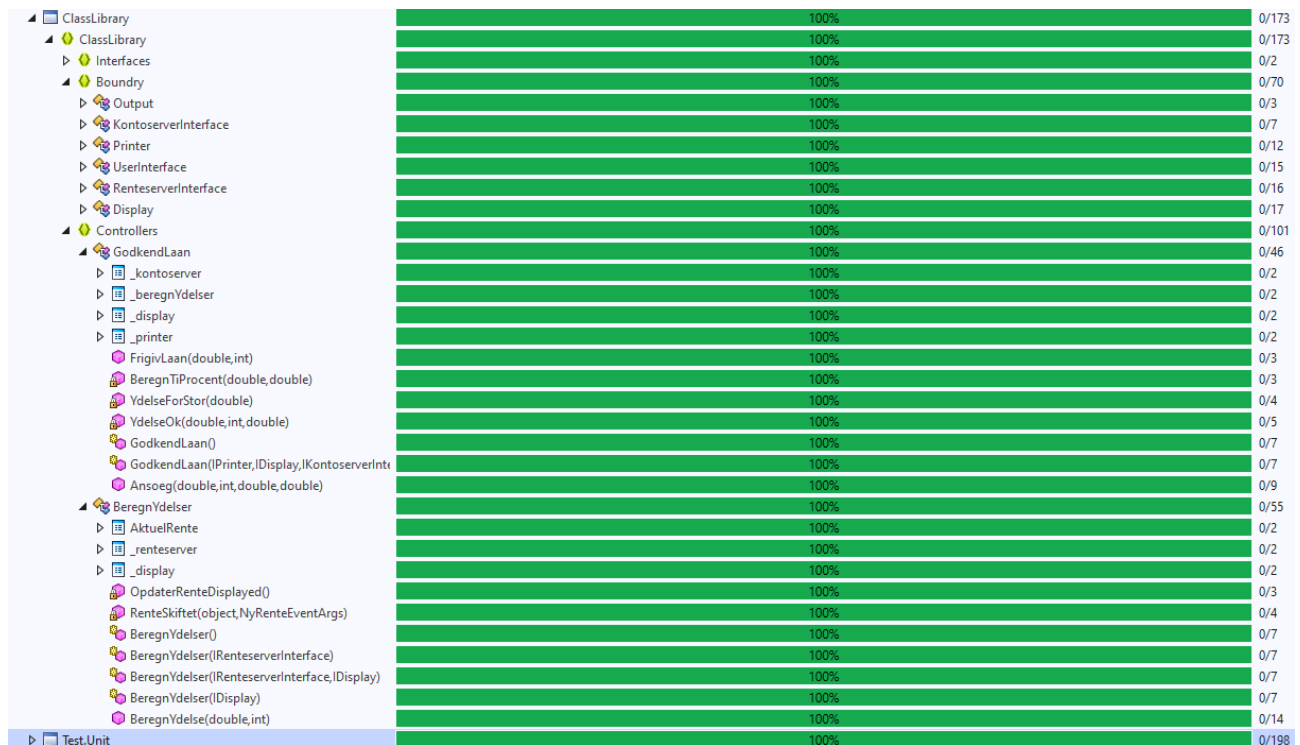
3 COVERAGE

Når vi arbejder med Unit Test er Line – og Branch coverage en god måde at se på om man har nok Test.

Ved at bruge ReSharper og Visual Studie Enterprise kan man nemt se hvor mange linjer der har coverage. Og hvilke klasser der har. Et andet værktøj er reportGenerator som kan lave nogle HTML eller Markdown filer til at vise coverage som rapport. Der er lagt den Seneste Report i billag under CoverageReport. Der skal man bare åbne Index.html. Den kommer fra det CI script som kan findes i [.github/workflows/dotnet.yml](#). Det Script bruges til at få en privat CI Runner til at teste at programmet kan bygges og at testene består.

For at sikre at vi har nok test går vi ind og kigger på Boundry values. Ved at lave test Cases omkring vores Boundrys kan vi sikre at vi får testet "alle" senarier uden at skulle lave Uendeligt mange test.

Linecoverage siger meget simpelt hvor meget mange linjer af Classernes kode er kørt igennem under test. Dette giver ikke et enerådigt resultat for om det er godt nok testet, bare at koden er løbet igennem.



Figur 11 Line Coverage Visualisation

I forhold til branch coverage kigger man på om Testene går i alle retninger ved if's det kan give et indblik i om "alle" situationer bliver testet.

4 SWT FACTS TO TALK ABOUT

4.1 UNIT TEST VS INTERGRATIONS TEST

UnitTest og integrationstest er to forskellige typer softwaretest. UnitTest fokuserer på at teste individuelle "units" eller komponenter i en softwareapplikation, såsom metoder eller funktioner, isoleret fra resten af systemet. Dette giver udviklere mulighed for at fange fejl tidligt i udviklingsprocessen. Integrationstest fokuserer på den anden side på at teste interaktionerne og sammenspillet mellem forskellige komponenter eller systemer. Dette gøres for at sikre, at softwaren fungerer efter hensigten, når alle dens komponenter er kombineret og koblet sammen. Integrationstest udføres normalt efter UnitTest, og det udføres normalt af et QA Team eller testere. Generelt er UnitTestning mere granuleret og fokuserer på små, isolerede stykker kode, mens integrationstest ser på, hvordan de forskellige dele af systemet arbejder sammen.

4.2 DYNAMISK ANALYSE AF SOFTWARE

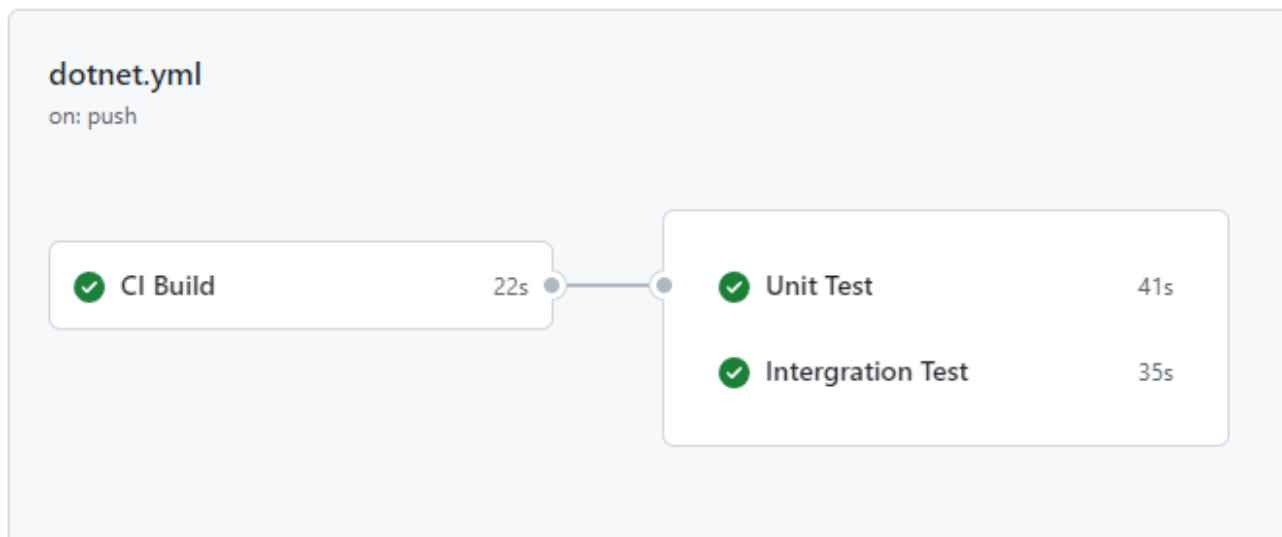
Dynamisk analyse er en metode til at evaluere opførslen af et system eller software ved kører det og observere dets runtime-adfærd. Dette kan omfatte overvågning af input og output, sporing af hukommelsesforbrug og analyse af ydeevne. Det bruges ofte i softwaretest og fejlfinding, men også i sådan noget som i sikkerhedsundersøgelse for at identificere sårbarheder systemet. Man bruge Functional testing, Performance testing, Stress testing, Automated testing og debug til at lave sine dynamiske analyse.

4.3 PROJEKT WORKFLOW – CI YAML

Når man er flere om at arbejde sammen i en projekt er de nogle almindelige udfordringer. Hvordan skal man holde styr på historikken af en kode? Hvordan skal man samle kode fra forskellige udviklere? Og hvordan skal man hjælpe hinanden med at sikre god kode?

Hvis man bruger Git til at hjælpe med det her er første problem løst. Hvis man så bruger Branching kan man lave sin egen udvikling når man arbejder på en feature og så Branche ind i den originale gren igen når man er færdig. For så at sikre at koden man lægger ind igen er god. Så bør man lave en Pull Request / Merge Request som så skal Reviewes enten af ens med Udviklere eller af nogle andre som er gode til at tjekke for fejl.

Når man så arbejder på Git så giver det god mening at arbejde med Commits. Og disse commits bør være isolerede ændringer så man kan arbejde til bare til succesfulde udgaver af systemet hvis det en dag skulle fejle. Så bør man først Pushe til Det fælles Repository når man har kørt alle test igennem og set at de virker. For at gøre sådan noget nemmere kan man sætte en CI (continues intergration) op. Dette gør det muligt at få en computer til at lave en automatiseret del af det at teste systemet når det bliver pushed til en online GIT. Det er også lavet i denne opgave, som jeg rigtig Gerne vil snakke mere om da det er ved brug af Github Actions som slet ikke har været en del af pensum, men er en udvidelse til test.



Figur 12 CI Run Overview from dotnet.yml