

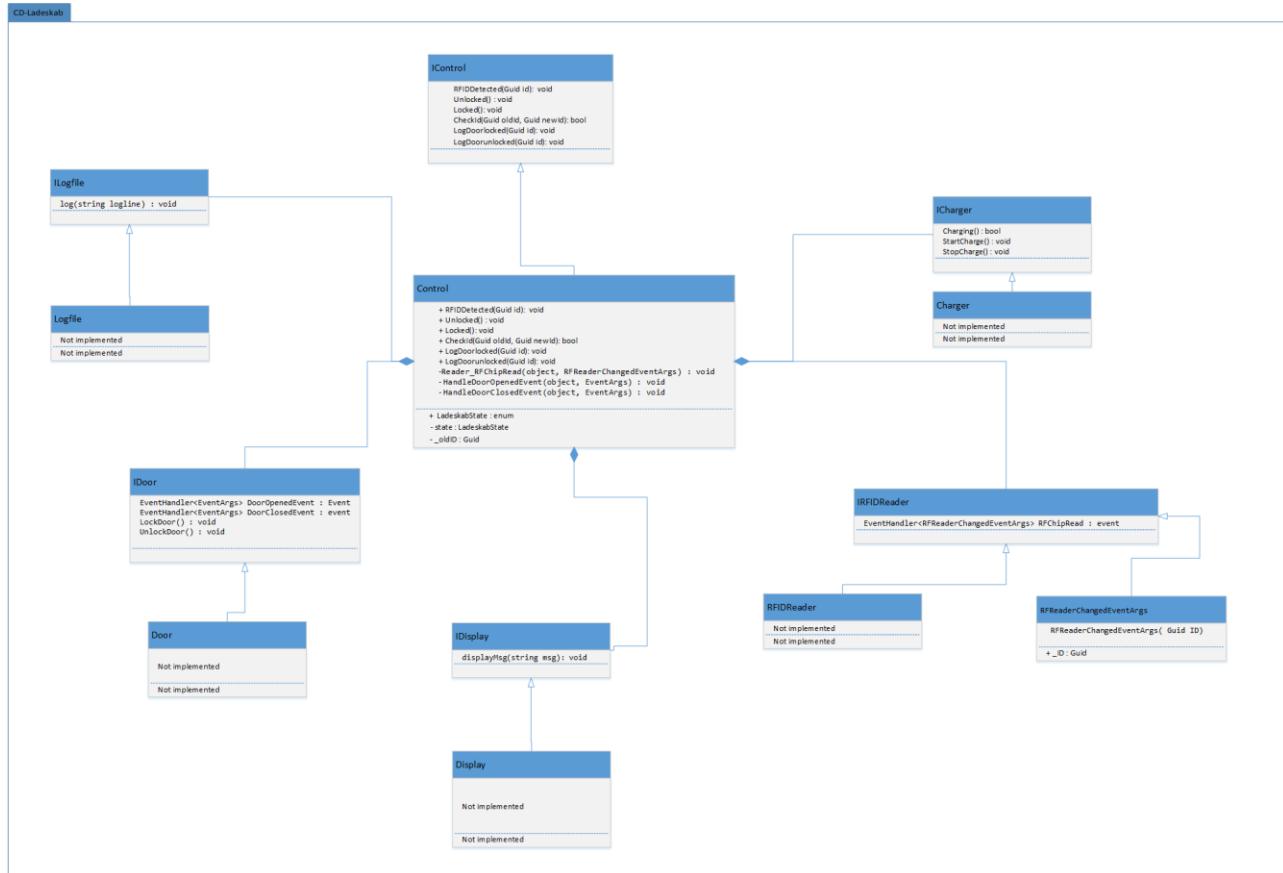
# Eksamensopgave

<b>Kursus:</b>	I4SWT/ST4SWT – Software Test - Hjemmeeksamen
<b>Eksamensdato:</b>	2019-06-13 kl. 9:00
<b>Varighed:</b>	24 timer
<b>Underviser:</b>	Frank B. Jakobsen
<b>Eksamenermin:</b>	Sommer 2019
<b>Praktiske informationer:</b>	
<b>Digital eksamen</b>	
Opgaven tilgås og afleveres gennem den digitale eksamensportal.	
Opgavebesvarelsen skal afleveres i ZIP-format	
Husk at upload og aflevere i Digital eksamen. Du vil modtage en elektronisk afleveringskvittering, straks du har afleveret.	
Husk at aflevere til tiden, da der ellers skal indsendes dispensationsansøgning.	
Husk angivelse af navn og studienummer på alle sider, samt i dokumenttitel/filnavn.	
<b>Hjælpemidler:</b>	
Alle hjælpemidler må benyttes, herunder internettet som opslagsværktøj, men opgaven er en personlig opgave.	

Studerende: Abdul-Rahman Barakeh

Studienummer: 201706954

## Delopgave 1



Figur 1 Klassediagram over ladeskabs systemet

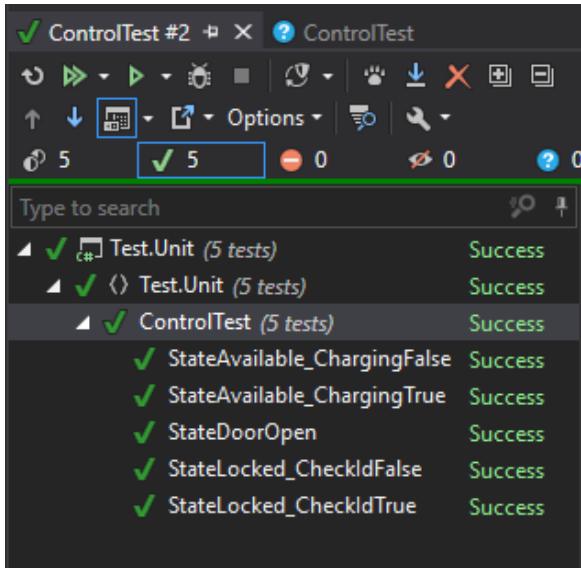
Klassediagrammet ses på figur 1. Det der gør designet testbart, er at der er lavet interfaces til hver klasse. Herudover er opgaverne delt ud på forskellige klasser, således at der undgås at Control bliver til en monolit. Der er fuldt SOLID princippet for S, *Single responsibility*.

## Delopgave 2

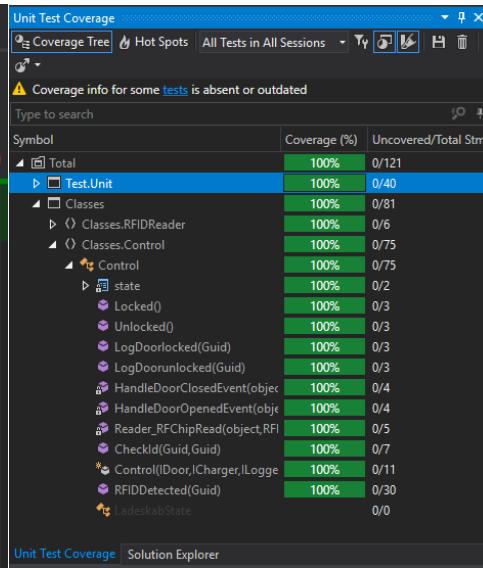
Henvises til namespace "Classes.Control" i solution "SWT\_Eksamens\_ChargeCloset".

## Delopgave 3

Efter implementeringen af unittest, hvor Control var unit under test, så kan resultatet af unittestene ses i både testsuiten på Figur 2 og coverragerapporten over unittestene. På Figur 3 ses en coverage på 100% som betyder at alle dele i Control er blevet kaldt. Om de er blevet testet på eller testet rigtig, er ikke muligt at se i en coveragerapport.



Figur 2 Testresultater

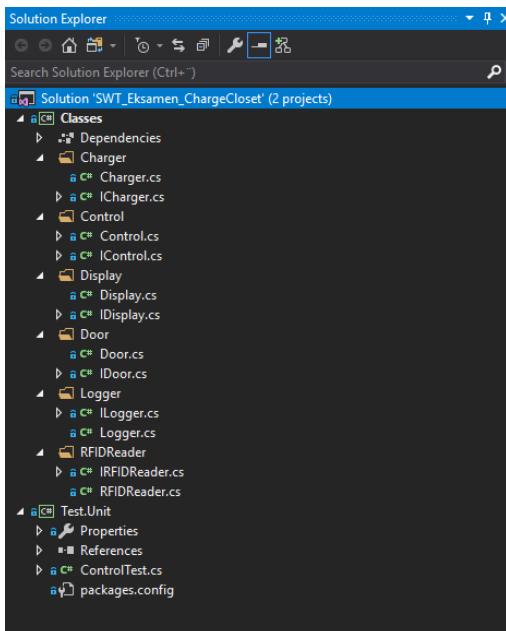


Figur 3 Coverragerapport over Control unittest

For Implementering af unittest henvises der til namespace "Test.Unit" is solution "SWT\_Eksamens\_ChargeCloset".

#### Delopgave 4

I figur 4 ses den valgte struktur over visual studioprojektet. Der er valgt at sætte det op i 2 projekter et class library projekt til opsætning af klasser og interfaces, samt et class library projekt til opsætning af unittest. I "Classes" projektet er der lavet undermapper til hver del af systemet for at holde en god struktur. I undermapperne ligger både interfaces og klasser, dog ikke implementerede klasser. Eventuelt kunne der tilføjes et tredje, console app, projekt i fremtidig iteration, hvor der her ville blive implementeret en main funktion hvor programmet ville kunne blive afprøvet i en helhed ved færdig implementering af klasserne i "Classes".



Figur 4 Solution struktur

## Delopgave 5

Et af de steder hvor designvalget om interfaces er blevet brugt er gældende i hele test projektet. Det ses på Figur 5. Ved at bruge Interfaces kan der laves fakes ved brug af et isolationsframework, ved navn NSubstitute. Idéen er at ved at fake interfacene kan der ved hjælp af NSubstitute manipuleres med koden således at der uden at implementerer de tilhørende klasser stadig kan simulere et virkende system. Dette ville ikke være muligt hvis ikke interfaces var blevet taget i brug.

```
[SetUp]
0 references | Abdul, 2 hours ago | 1 author, 1 change
public void SetUp()
{
    _door = Substitute.For<IDoor>();
    _charger = Substitute.For<ICharger>();
    _logger = Substitute.For<ILogger>();
    _reader = Substitute.For<IRFIDReader>();
    _display = Substitute.For<IDisplay>();

    _uut = new Control(_door, _charger, _logger, _reader, _display);
}

[Test]
0 references | Abdul, 2 hours ago | 1 author, 1 change
public void StateAvailable_ChargingFalse()
{
    var id = Guid.NewGuid();
    _charger.IsConnected().Returns(false);
    _reader.RFChipRead += Raise.EventWith<RFReaderChangedEventArgs>(new RFReaderChangedEventArgs(id));
    _uut.RFIDDetected(id);

    _display.Received().displayMsg("Din telefon er ikke ordentlig tilsluttet. Prøv igen.");
}
```

Figur 5 Snippet fra unittest1

Design valget om Single responsibility princippet ses i figur 6. Grundet måden koden er designet på giver muligheden for at teste på to måder. Alt kode i possibility 1 forsager samme stadieændring som possibility 2. Ved at have isoleret door eventsne, er det muligt ved raising af et event at ændre på stadiet af ladeskabets dør.

```
[Test]
0 references | Abdul, 39 minutes ago | 1 author, 2 changes
public void StateLocked_CheckIdFalse()
{
    //possibility 1
    //var id = Guid.NewGuid();
    //_charger.IsConnected().Returns(true);
    //_reader.RFChipRead += Raise.EventWith<RFReaderChangedEventArgs>(new RFReaderChangedEventArgs(id));
    //_uut.RFIDDetected(id);

    //Possibility 2
    _door.DoorClosedEvent += Raise.Event();

    _uut.RFIDDetected(Guid.NewGuid());
    _display.Received().displayMsg("Forkert RFID tag");
}
```

Figur 6 Snippet fra unittest2

## Delopgave 6

Den adfærdsbaserede test ses på figur 7. Her testes der på en fake, som i dette tilfælde er display.

Kendetegnet på en adfærdsbaseret test er at der testes på en fake og ikke en unit under test. Der er i løsningen ikke testet på en unit under test, som er kendetegnet på en tilstandsbaseret test. Der kan dog

argumenteres for at der er blevet testet på unit under test, da den inddrages i den adfærdsbaserede test, samt set på figur 3, hvor den her opnår en 100% coverage.

```
[Test]
0 references | Abdul, 2 hours ago | 1 author, 1 change
public void StateDoorOpen()
{
    _door.DoorOpenedEvent += Raise.Event();
    _uut.RFIDDetected(Guid.NewGuid());
    _display.Received().displayMsg("Nothing");
}
```

Figur 7 Adfærdsbaseret test

## Delopgave 7

Som benævnt i delopgave 5 tages der brug af et isolationsframework kaldt NSubstitute. En af de mange fordele ved et isolationsframework er egenskaben til nemmere at sikrer sig at der laves blackbox test, grundet at der ingen implementering benyttes og derfor er kun kan testes via blackbox metoden. Ulemper ved isolationsframework kan ligge i opbygningen af et isolationsframework. I det at det er muligt at der kan forekomme fejl i frameworket, kan det blive en ulempe at bruge.

## Delopgave 8

Ved at analyserer sekvensdiagrammet og tilstandsdiagrammet, bliver det tydeligt at se at det er de forskellige tilstande i systemet der skal testes og være fokus på. De implementerede unitest er baseret på de tre mulige tilstande i switch-casen, i Control klassen. I det at alle andre klasser og funktioner inddrages i switch-casen er det kun nødvendigt at uniteste switch-casens tre tilstande.

## Delopgave 9

Implementeringen af Logger ses på Figur 8, her bruges StreamWriter og TextWriter til at logge i en txt-fil. I Figur 9 ses den implementerede test, sat op i AAA metoden. "Arrange-Act-Assert". StreamReader bruges her til at læse fra den oprettede bil.

```
3 references | Abdul, 4 hours ago | 1 author, 3 changes
public class Logger : ILogger
{
    private string _logFile = Path.GetTempPath()+"logfile.txt";

    4 references | Abdul, 4 hours ago | 1 author, 3 changes
    public void Log(string logline)
    {
        using (StreamWriter writer = File.AppendText(_logFile))
        {
            loggin(logline,writer);
        }
    }

    1 reference | 0 authors, 0 changes
    public void loggin(string logginline, TextWriter writer)
    {
        writer.WriteLine(value: $"{logginline}");
    }

    //Inspiration inddraget fra: https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-open-and-append-to-a-log-file
}
```

Figur 8 Implementering af logger<sup>1</sup>

På figur 10 ses den fil sti, hvorledes txt-filen beligger i, samt ses den loggede linje som bliver skrevet og læst i testen. De overvejelser, det forekommer ved Implementeringen, er en overvejelse om inddragning af

<sup>1</sup> <https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-open-and-append-to-a-log-file>

Prøve: I4SWT/ST4SWT

Dato: 2019-06-13

jsonformattering. Her ville det være en god idé at serialiserer data'en til json format, hvor der herudover bliver forbundet brugernavne til, således at det kan ses hvem der sidst havde en specifik brik. Udover dette ville det måske være en rationelt valg at oprette en ny logfil for hverdag således at en logfil ikke indeholder 100 dages-log data.

The screenshot shows a code editor with a C# unit test. The test class is named `LoggerTest`. It contains a `Setup` method that creates a new `Logger` instance. The `LogLocked` test method arranges for a line to be logged ("Testing the logging"), acts by calling `_ut.log(lineToBeLogged)`, and asserts that the result matches the expected line.

```
namespace Test.Unit
{
    [TestFixture]
    0 references | 0 changes | 0 authors, 0 changes
    class LoggerTest
    {

        private Logger _ut;

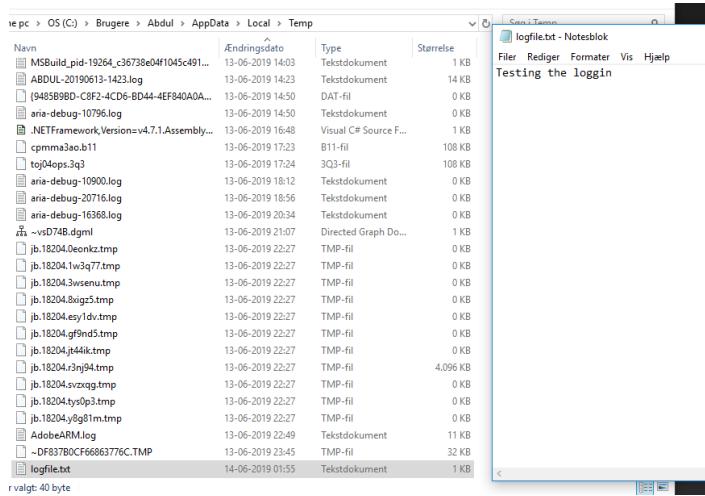
        [SetUp]
        0 references | 0 changes | 0 authors, 0 changes
        public void Setup()
        {
            _ut = new Logger();
        }

        [Test]
        0 references | 0 changes | 0 authors, 0 changes
        public void LogLocked()
        {
            //Arrange
            string lineToBeLogged = "Testing the logging";
            string result = String.Empty;

            //Act
            _ut.log(lineToBeLogged);
            using (StreamReader R = File.OpenText(path: Path.GetTempPath() + "logfile.txt"))
            {
                result = R.ReadLine();
            }

            //Assert
            Assert.That(result, Is.EqualTo(lineToBeLogged));
        }
    }
}
```

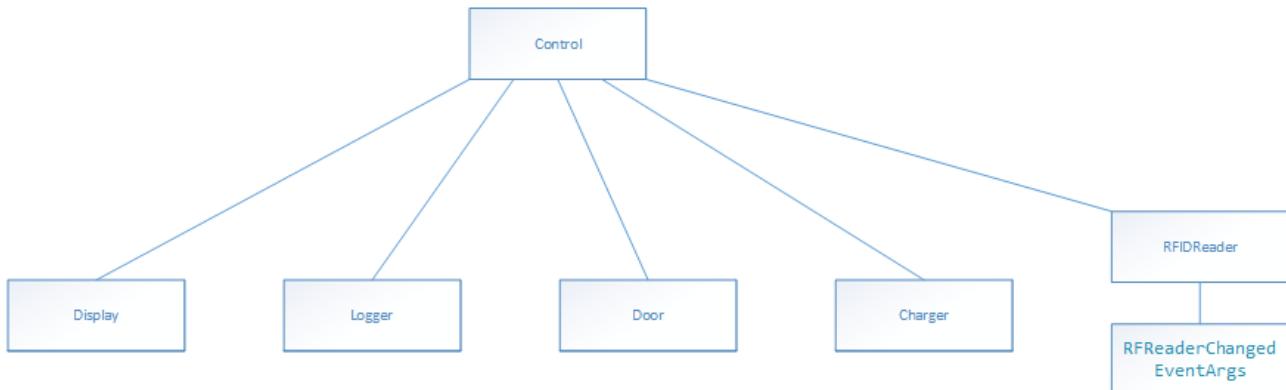
Figur 9 Unit Test af logger



Figur 10 Tempmappe indeholdende oprettede txt-fil & txt-fil

## Delopgave 10

På figur 11 ses det designede dependency-tree over systemet. Der tages her ikke højde for interfaces men delene i systemet i en helhed. De vigtige ting ved en integrationstest er at teste interaktionerne og interfacene mellem de forskellige moduler. Målet er at sikrer sig en korrekt interaktion mellem modulerne som kan være alt fra implementerede klasser, eksterne klasser i form af dll-filer, eller ekstern hardware i anledning af at der er tale om en integrationstest af et system der indeholder hardware.



Figur 11 Dependency-tree

## Delopgave 11

De pipelines der ville være fornuftige at have med, er listet i punktform forneden i en specifik rækkefølge.

- Fetch From Git
- Update NuGet packages
- Build
- Run coverage of unitests
- Publish test results
- Run integrationtest
- Publish Test results
- Publish coverage results

Det er vigtigt at opstille pipelines i en specifik rækkefølge. Eksempelvis er det ikke muligt at teste kode hvis ikke koden er blevet hentet fra GitHub, og bygget. En ting der især er vigtig, er at pipelinen for unitest skal forekomme før integrationstesten. Grunden lyder på at en af kravende for at teste interaktionerne mellem de forskellige moduler i et system, er at sikrer sig at de individuelle moduler virker i sig selv. Herudover ville der ikke kunne opnås bedst mulig coverage, hvis unitest og integrationstest fejler eller bliver testet efter coverage rapporten, og er af den grund den sidste pipeline.