# Enhetstestning

## En implementation i .NET

Simon Långsjö

Examensarbete för ingenjörs (YH)-examen

Utbildningsprogrammet för informationsteknik

Vasa 2016

**EXAMENSARBETE**


| | |
|---|---|
| Författare: | Simon Långsjö |
| Utbildningsprogram och ort: | Informationsteknik, Vasa |
| Handledare: | Mikael Jakas & Kaj Wikman |


Titel: *Enhetstestning – En implementation i .NET*

---


Datum: 31.3.2016 Sidantal: 24


---

## Abstrakt


Detta examensarbete utfördes åt Oy Abilita Ab i Jakobstad. Examensarbetet behandlar vad enhetstestning i .NET innebär och hur man implementerar och använder det med Microsoft Visual Studio. Uppgiften var att lära mig och implementera enhetstester i Abilitas eArkiv. eArkivet är ett tillägg till Abilita Hälsovård som gör det möjligt att koppla upp programmet mot det nationella patientdataarkivet.

Resultatet av examensarbetet blev en serie enhetstester som kan köras för att kontrollera blankettobjekt och vid ändringar av programkoden, för att kontrollera att funktioner fungerar som tidigare. Informationsblanketten och Envelope blev två mera avancerade kollektioner av tester som både skapar ett blankettobjekt och sedan kör enhetstester på dem.


---


Språk: svenska Nyckelord: enhetstest, Unit test, .NET, C#, IT


---

**OPINNÄYTETYÖ**


| | |
|---|---|
| Tekijä: | Simon Långsjö |
| Koulutusohjelma ja paikkakunta: | Tietotekniikka, Vaasa |
| Ohjaajat: | Mikael Jakas & Kaj Wikman |


Nimike: *Yksikkötestaus – Täytäntöönpanon .NETissä*

---


Päivämäärä: 31.3.2016 Sivumäärä: 24


---

**Tiivistelmä**


Tämä opinnäytetyö tehtiin Oy Abilita Ab:lle Pietarsaareessa. Opinnäytetyö käsittelee mitä yksikkötestaus .NETissä merkitsee ja miten toteutetaan ja käytetään sitä Microsoft Visual Studiolla. Tehtävä oli oppia yksikkötestausta ja toteuttaa yksikkötestejä Abilitan eArkistossa. eArkisto on Abilita Terveydenhuollon lisäosa, joka mahdollistaa kytkemisen kansalliseen potilastietoarkistoon.

Tuloksena on sarja yksikkötestejä, joita voidaan ajaa tarkistamaan lomakeolioita ja kun koodia muutetaan, vahvistavat, että kaikki toimii niin kun ennen. Informaatiolomakkeesta ja Envelopesta tuli kehittyneempiä testikokoelmia, jotka sekä luovat lomakeolion että ajavat yksikkötestejä.


---


Kieli: ruotsi Avainsanat: yksikkötesti, Unit test, .NET, C#, IT


---

**BACHELOR'S THESIS**


| | |
|---|---|
| Author: | Simon Långsjö |
| Degree Programme and location: | Information Technology, Vaasa |
| Supervisor: | Mikael Jakas & Kaj Wikman |


Title: *Unit Testing – An Implementation in .NET*

---


Date: 31.3.2016 Number of pages: 24


---


**Abstract**

This thesis was made at the request of Oy Abilita Ab in Jakobstad. The thesis explains what a unit test in .NET implies and how to implement and use it with Microsoft Visual Studio. The task was to learn unit testing and implement unit tests in Abilita's eArchive. The eArchive is an addition to Abilita Healthcare which will make it possible for the application to connect to the national Patient Data Repository.

The result of the thesis is a series of unit tests that can be run to validate form objects and when the code is changed to validate that functions are still working as intended. The information form and the Envelope resulted in two more advanced collections of tests that both create a form object and run unit tests on it.


---


Language: Swedish Key words: unit test, .NET, C#, IT


---

# Innehållsförteckning

Förklaringar

| | |
|---|---|
| Array | En datastruktur som består av en samling element. |
| enum | Enum eller enumerated type (uppräkningstyp) är en datatyp vars värden består av konstanter. |
| ERP | Förkortning för Enterprise Resource Planning. ERP eller affärssystem är ett paket med integrerade program som sköter ett företags informationshantering och underlättar ett företags styrning och administration. |
| Hårdkodning | Hårdkodning är en programmeringsterm som betyder att ett eller flera värden inte kan ändras. |
| Konstruktor | Den del av programkoden som alltid körs när man skapar ett objekt. |
| Metadata | Metadata är data som beskriver annan data d.v.s. definition eller beskrivning av något. |
| Serialisering | Data sparas till ett format som sedan kan lagras. |
| SOAP | Förkortning för Simple Object Access Protocol och är ett protokoll för utbyte av information. SOAP är XML-baserat. |
| Query | En begäran om information från t.ex. en databas. |
| XML | Förkortning för Extensive Markup Language. XML är ett utbyggbart och universellt märkspråk som definerar regler för kodningen av dokument i ett format som är både läsbart för människor och maskiner. |

# 1 Inledning

I detta kapitel presenteras arbetsgivaren för examensarbetet och även själva uppdraget i korthet.

## 1.1 Arbetsgivare

Oy Abilita Ab är ett It-företag grundat 1982 och har strax över 30 anställda. Företaget har sitt huvudkontor i Jakobstad och en mindre enhet i Korsholm. Som verkställande direktör fungerar Tommy Sjöholm. Det ägs till största delen av personalen. Till de viktigaste produkterna hör:

- Kommunernas ERP-lösning som är ett samarbete med Unit4s Business World, ett affärssystem med integrerade moduler för bl.a. ekonomi, löner, projektadministration och logistik.
- Abilita Hälso- och sjukvård som används av sjukhus och hälsostationer. Systemet är ett paket av integrerade moduler och kan skräddarsys efter kundens önskemål.
- Abilita Dagvård som är ett system som hjälper daghem med bl.a. hantering av avtal, registrering av närvaro och fakturering.
- Abilita Bibliotek som är ett heltäckande bibliotekssystem. /1/

## 1.2 Uppdrag

Under min andra sommarpraktik på Abilita blev jag involverad i ett projekt som kallas eArkivet. eArkivet är ett tillägg till Abilitas Hälsovård som möjliggör att dokument från systemet skickas till nationella patientdataarkivet var de långtidsarkiveras. FPA har krav på att dokumenten som skickas in ska vara i CDA-format (se kap 2.5) som är en XML-baserad standard på sjukvårdsdokument. För att uppnå detta måste blanketterna serialiseras i programmet innan de skickas.

I det här skedet kom behovet av testning. Jag fick uppdraget att lära mig och implementera enhetstester i programmet så att de kan köras för att kontrollera att blanketterna serialiseras rätt och att logiken i programmet fungerar korrekt.

# 2 Tekniker

Programlogiken i eArkivet är programmerat i C# med Visual Studio, vilket gjorde att programmeringsspråket och programmet som använts var självklart. Dessutom har Visual Studio bra stöd för enhetstestning inbyggt. LINQ (se kap 2.4) användes för att det är en kraftfull uppsättning funktioner och lätt att använda för ändamålet.

## 2.1 C#

I januari 1999 bildade Anders Hejlsberg ett team för att utveckla ett nytt programmeringsspråk. Språket kallads först för Cool som stod för "C-like Object Oriented Language" men när .NET-projektet tillkännagavs år 2000 hade det döpts om til C#. Det är gjort för att vara ett lättanvänt, modernt, objektorienterat programmeringsspråk för både serverapplikationer och integrerade system. Syntaxen är relativt lik C och C++, så om man kan dem så är det lätt att lära sig C#. Språket och implementationer av det är typsäkert, har storlekskontroll av arrays, märker ifall du försöker använda variabler som inte blivit initialiserade samt har automatisk skräpsamling. Nyaste versionen i skrivande stund är C# 5.0 som släpptes i augusti 2012 i och med lanseringen av .NET Framework 4.5. I version 5.0 kom nya funktioner såsom asynkrona funktioner och attribut m.h.a. vilka man kan få information om vad som kallar på en funktion. Kodexempel på ett klassiskt s.k. "Hello World"-program i *Kodexempel 1* nedan. /2/

*Kodexempel 1. "Hello World-program" i C#.*

```
using System;
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, world!");
    }
}
```

## 2.2 Microsoft Visual Studio

Microsoft Visual Studio är en integrerad utvecklingsmiljö skapad av Microsoft. Visual Studio används främst för att utveckla program för Microsoft Windows men kan även användas till att skapa webbsidor, webbapplikationer och webbtjänster. Det använder sig av Microsofts programvaruplattformer Windows API, Windows Forms, Windows Presentation Foundation (WPF), Windows Store samt Microsoft Silverlight. Visual Studio kan producera både maskinkod och hanterad kod. För tillfället stöder verktyget programmeringsspråken C, C++, Visual Basic, C# och F# men via separata installationer kan man även få stöd för M, Python och Ruby. Förutom dessa stöder det även XML/XLST, HTML/XHTML, JavaScript och CSS vilka används främst till webbsidor.

Första versionen av Visual Studio kom 1995. Microsoft hade flera utvecklingsverktyg men det var det första som kombinerade flera verktyg i samma program. Microsoft använder sig av platser som kodnamn för versionerna, t.ex. Visual Studio 97 kallades för Boston. Visual Studio kommer i fyra olika utgåvor: Express, Professional, Premium och Ultimate. Express är en nedbantad gratisversion medan Ultimate är den dyraste och enda som stöder IntelliTrace, belastningstest och modellering. Dock kan Premium visa modellerna men inte själv modellera. /3/

## 2.3 .NET Framework

.NET Framework är en systemkomponent som körs främst på Microsoft Windows. Det innehåller ett stort bibliotek och har stöd för programkompabilitet mellan flera olika programmeringsspråk vilket betyder att alla språk kan använda kod skriven i ett annat språk. Program kodade i .NET Framework körs i en programmiljö som kallas CLR, Common Language Runtime, vilket är en virtualmaskin för mjukvara. CLR tillhandahåller tjänster såsom minneshantering, säkerhet och undantagshantering. CLR och klassbiblioteket bildar tillsammans .NET Framework.

Microsoft började i slutet av 1990-talet utveckla .NET Framework och i slutet av 2000 släpptes första beta versionen av .NET 1.0. Version 3.0 inkluderades i Windows Server 2008 och Windows Vista. Version 3.5 i Windows 7 och Windows Server 2008 R2 men kan även installeras på Windows XP och Windows Server 2003. I samband

med Visual Studio 2010 lanserades version 4 medan 4.5 kommer med Windows 8 och Windows Server 2012. Nyaste versionen för tillfället är 4.5.1 som kommer med Windows 8.1 och Windows Server 2012 R2. /4/

## 2.4 Language-Integrated Query

Language-Integrated Query (LINQ) som först introducerades i Visual Studio 2008 är en uppsättning verktyg vilka vidare utvidgar de starka query möjligheterna som finns i C# och Visual Basic. LINQ använder sig av standardiserade och relativt lätt kod för att hämta och uppdatera data och kan användas till de flesta typer av data. Exempel på datalagringar som LINQ kan användas mot är .NET Framework kollektioner, SQL Server databaser, ADO.NET dataset och XML-dokument. Kodexempel på LINQ till kollektion i *Kodexempel 2* nedan visar exempel på hur man med hjälp av LINQ kan få ut alla kunder vars namn är Kalle i en lista med kunder.

*Kodexempel 2. LINQ mot en lista i C#.*

```
List<Customer> resultList = (from customers in customerList
                             where customers.Name == "Kalle"
                             select customers).ToList();
```

Som kan ses i figuren liknar query-syntaxen väldigt mycket SQL, bara lite omvänt så att *select* kommer sist istället för först. När man skriver en query på detta sätt returnerar den en IEnumerable och därför används ToList() metoden i slutet som konverterar kollektionen till en generisk List<>. /5/

Det som användes mest av LINQ i detta examensarbete var dess operatorer. Operatorerna påminner mycket om de som finns i SQL som t.ex. *Sum*, *Min*, *Max* och *Average*. En operator som användes mycket i examensarbetet var *FirstOrDefault* vilken används för att få det första elementet i t.ex. en kollektion. Det som skiljer *First* från *FirstOrDefault* är det att *First* kastar ett undantag ifall inget värde matchar medan *FirstOrDefault* ger tillbaka standarvärdet. En annan är *ElementAt* som returnerar värdet som finns på den position man lägger som inparameter. Inparametern ska

vara i integerformat. *Any* är en bra operator ifall man inte behöver värdet utan bara kontrollera att det stämmer överens med värdet man testar mot. *Any* är ännu kraftfullare ifall man använder sig av lambdauttryck vilket behövs om man har en lista med objekt som innehåller flera olika egenskaper och man vill kolla att åtminstone ett objekt i listan har rätt värde på en specifik egenskap. Eftersom detta är svårt att förklara i text visas detta i *Kodexempel 3.* /11/

*Kodexempel 3. Hur Any används med lambdauttryck.*

```
public class Program
{
      public static void Main(string[] args)
      {
            var person = new Person();
            person.Items.Any(i => i.Phone == "Nexus");
      }
}

public class Person
{
      public string Name { get; set; }
      public PersonItems[] Items { get; set; }
}
public class PersonItems
{
      public string Phone { get; set; }
      public string Wallet { get; set; }
}
```

Här kommer förstås *Any* returnera false eftersom Phone inte tilldelades något värde. I examensarbetet användes *Any* i kombination med *Assert.*

## 2.5 Clinical Document Architecture

CDA är en internationell standard på patientjournaler från USA. Standarden utvecklas av företaget HL7 (Health Level Seven) som är en icke-vinstdrivande organisation stationerad i Ann Arbor, Michigan. Ett CDA-dokument kan innehålla alla typer av patientdata. Till typiska CDA-dokument hör utskrivningssammanfattningar, röntgenrapporter, inskrivningar, hälsokontroller och patologirapporter. Det mest populära användningsområdet är utbyte av information mellan företag. /13/

CDA definierar en patientjournal med sex egenskaper:

- Uthållighet – En journal fortsätter att existera i ett oförändrat skick.
- Förvaltning – En journal upprätthålls av en person eller organisation som blivit anförtrodd att ta hand om den.
- Potential för verifiering – En journal är en samling information som är avsedd att vara juridiskt autentiserad.
- Sammanhang – En journal ska innehålla information som gör att man vet vad den handlar om. Exempel på sådan information är vilken patient den beskriver och hurudan vård patienten fått.
- Helhet – Autentiseringen av journaler gäller hela dokumentet och gäller inte för delar av dokumentet om inte hela innehållet finns med.
- Mänsklig läsbarhet – En journal ska vara läsbar. /14/

Finland har valt att använda standarden och man har lokaliserat standarden enligt finländska normer. FPA upprätthåller dokumentationen. I Finland används version CDAR2 av standarden. Största skillnaden mellan R2 och gamla versioner är att R2 har både fritext och strukturerad data. CDA-dokumentet består av en header och en bodydel. I headern finns metadata till dokumentet, information om vården som patienten fått, info om serviceleverantören och information om patienten. Bodydelen innehåller själva innehållet i dokumentet i strukturerad form med olika datatyper. /15/

# 3 Enhetstestning

I det här kapitlet beskrivs vad ett enhetstest egentligen är. Klasserna som används och lite åsikter om varför det är värt att sätta tid på det presenteras också.

## 3.1 Definition

Det primära målet med ett enhetstest är att ta den minsta delen av som går att testa av ett program, isolera den från resten av koden och kontrollera att den uppför sig som man förväntat sig. Enhetstester har visat sig hjälpa till att identifiera defekter i program när de används. /7/

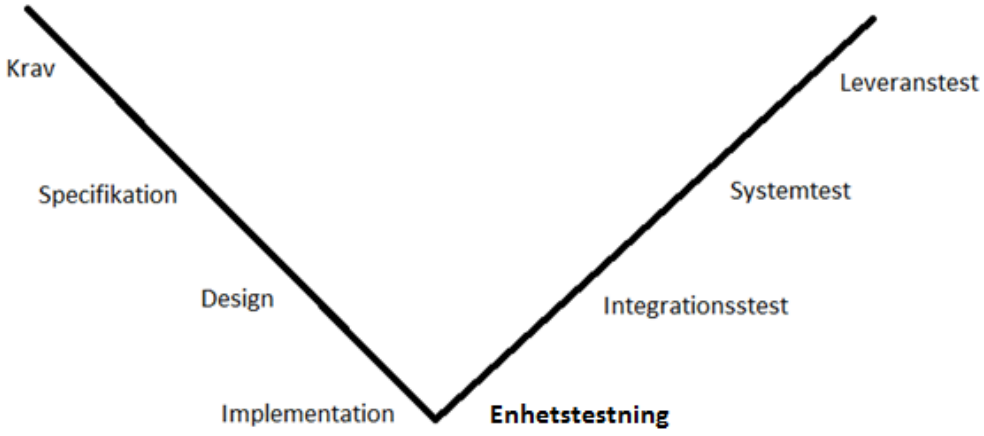
Exempel på områden som är typiska att enhetstesta är:

- Korrektheten av kalkyleringar och hanteringar gjorda av en *enhet*.
- Prestandaproblem såsom prestandaflaskhalsar observerade under upprepade anrop av en *enhet* och dess funktionalitet.
- Tillförlitlighetsproblem såsom minnesläckor observerade under förlängda upprepade anrop av en *enhet* och dess funktionalitet.
- Fönsterinnehåll och navigering i ett grafiskt användargränssnitt.
- Rapportinnehåll, layouter och kalkyleringar.
- Skapande, uppdatering och borttagning av filer och poster.
- Kommunikation mellan interagerande *enheter.*

Hur en *enhet* definieras beror på vilken tillvägagångssätt man använt sig av i utvecklingen av programvaran, till exempel:

- I ett procedurellt programmeringsspråk kan en *enhet* representeras av en procedur, funktion eller en närbesläktad grupp av procedurer och funktioner.
- I ett program utvecklat med ett objektorienterat programmeringsspråk kan en *enhet* representeras av en klass, en instans av en klass eller funktionaliteten implementerad av en metod.
- En *enhet* i en visuell programmiljö eller ett grafiskt användargränssnitt kan representeras av ett fönster eller en kollektion av element till ett fönster som t.ex. en gruppbox.
- En *enhet* i en komponentbaserad utvecklingsmiljö kan vara en fördefinierad återanvändningsbar komponent.

Beträffande *V-modellen* motsvarar enhetstestning implementationsfasen i livscykeln för programutveckling *(Figur 1).* V-modellen är en programutvecklings- och testmodell vilken hjälper till att markera behovet att tidigt i utvecklingsprocessen förbereda för testning. Vänstra sidan av ”V” representerar de traditionella utvecklingsfaserna medan den högra sidan representerar de motsvarande testningsfaserna. /9/




*Figur 1. V-modellen.*

Ett enhetstest ska endast köras i arbetsminnet, det ska vara snabbt och repeterbart. Med repeterbart menas att det inte ska t.ex. misslyckas varannan gång man kör det utan om det fungerar ska det fungera varje gång. Enhetstester ska inte använda några externa resurser såsom databaser eller filsystemet eftersom det då blir svårare att isolera och återställa så att det kan köras igen samt att andra test kan störa om de använder samma miljö. Om man testar mot en databas testar man även eventuell tredjepartskod som t.ex. databastriggers som kan störa testresultatet. Filsystem kan innehålla störande faktorer som också kan störa. /8/

Enhetstestning kan jämföras med integrationstestning som är nästa steg i testprocessen och är en logisk extension till enhetstestning. Integrationsstestning i dess simplaste form är att man tar två *enheter* som redan blivit testade och kombinerar dem till en komponent och sedan testar gränssnittet mellan dem. Med komponent i detta sammanhang menas en integrerad samling av fler än en *enhet*. När man testat komponenterna är nästa steg att kombinera dem till större delar av programmet och testa komponenterna med de i andra grupper i programmet. Idén är

att testa olika kombinationer av komponenterna. Till sist testas alla moduler tillsammans. Kort sagt kan man säga att integrationstestning identifierar problem som uppstår när man kombinerar *enheter.* /10/

## 3.2 Varför?

Enhetstester gör utveckling lättare på flera sätt, som till exempel:

- Lättare att hitta buggar.
- Lättare att upprätthålla koden.
- Lättare att förstå koden.
- Lättare att utveckla programmet.

Enhetstester möjliggör även automation av testningsprocessen. När man en gång kodat testerna är det bara att köra dem för att kontrollera att allt fungerar som förut efter att man ändrat eller lagt till något i koden. I komplexa delar av programmet gör enhetstester det lättare att upptäcka fel eftersom man ser mera precist var i programmet felet sitter. Testningen blir mera heltäckande eftersom man ger uppmärksamhet till varje enskild *enhet* istället för hela programmet. /7/

## 3.3 Klasser

Microsoft.VisualStudio.TestTools.UnitTesting är namnrymden där klasserna som används för att enhetstesta finns. Namnrymden innehåller attribut som används för att identifiera tester, bestämma i vilken ordning testerna körs, programhantering och skapa startdata. I följande underkapitel beskrivs de delar som användes i detta examensarbete samt några till. /6/

### 3.3.1 Assert

*Assert* är den klass som användes mest i examensarbetet. Den är kärnan i ett enhetstest och är lätt att använda. *Assert* är i princip en if-sats men har den funktionen att testet där *Assert* finns misslyckas ifall villkoret inte stämmer överens. *Assert* är bra när man vill att en funktion ska ha ett visst returvärde. Man använder då metoden *IsTrue* som visas i *Kodexempel 4.*

*Kodexempel 4. Metoden IsTrue.*

```
string input = "foo";
string output = SomeMethod(input);
Assert.IsTrue(input == output);
```

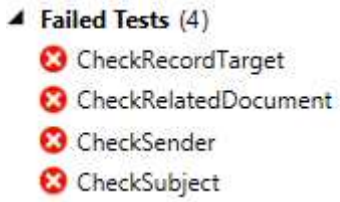
Metoden i figuren är menad att returnera inputvärdet tillbaka. *IsFalse* fungerar på samma sätt men tvärtom. Om man vill kolla att ett värde inte blir null kan man använda *IsNotNull* eller *IsNull* ifall man vill att det ska vara det. Metoden *Fail* används när man vill att testet ska misslyckas oavsett villkor. Det används mest om man har egna if-satser som man testar med. *Fail* visas i *Kodexempel 5* med samma exempel som i *Kodexempel 4.*

*Kodexempel 5. Metoden Fail.*

```
string input = "foo";
string output = SomeMethod(input);
if (input != output)
      Assert.Fail();
```

Andra användbara metoder är olika överlagringar av *AreEqual* och *AreNotEqual* som är ungefär samma som *IsTrue* men man sätter in värdena som inparametrar istället för att skriva ett villkor. De har stöd för flera olika typer som inparametrar som t.ex. string, single, double och object. Equal-metoderna med string-parametrar har även en tredje parameter som är boolean med vilken man kan ställa in om kontrollen ska vara skiftlägeskänslig. De flesta Assert-metoder har även en överlagring med en extra string-parameter var man kan ställa in ett meddelande som skrivs ut ifall testet misslyckas.

Alltid när en *Assert* misslyckas kastas undantaget *AssertFailedException*. Man ska inte fånga undantaget med *”try...catch”* eller dylikt utan detta betyder att enhetstestet har misslyckats och detta indikeras i Visual Studios *”Test Explorer”* med en röd ikon när man kört testet (*Figur 2*).




*Figur 2. Skärmbild av hur Test Explorern ser ut när man har misslyckade test.*

### 3.3.2 TestClassAttribute och TestMethodAttribute

*TestClassAttribute ([TestClass])* används för att visa att den innehåller testmetoder och krävs för att den ska köras när man kör enhetstesterna från *"Test Explorer"*. En testklass kan inte ärvas.

*TestMethodAttribute ([TestMethod])* används för att identifiera testmetoder på samma sätt som *TestClassAttribute* för klasser. Metoden måste finnas i en testklass. En testmetod måste alltid returnera *void* och kan inte ha några parametrar.

*Kodexempel 6. Hur TestClassAttribute och TestMethodAttribute implementeras.*

```
[TestClass]
public class UnitTest1
{
      [TestMethod]
      public void TestMethod1()
      {
      }
}
```

### 3.3.3 ExpectedExceptionAttribute

*ExpectedExceptionAttribute ([ExpectedException])* används på en testmetod för att indikera att ett undantag är väntat när testmetoden körs. Det används då tillsammans med *typeof* som i *Kodexempel 7.*

*Kodexempel 7. Hur ExpectedExceptionAttribute används.*

```
[TestMethod]
[ExpectedException(typeof(NullReferenceException))]
```

Om inte det väntade undantaget kastas så kommer testet beaktas som misslyckat. Endast en instans av detta attribut får finnas på en testmetod. /6/

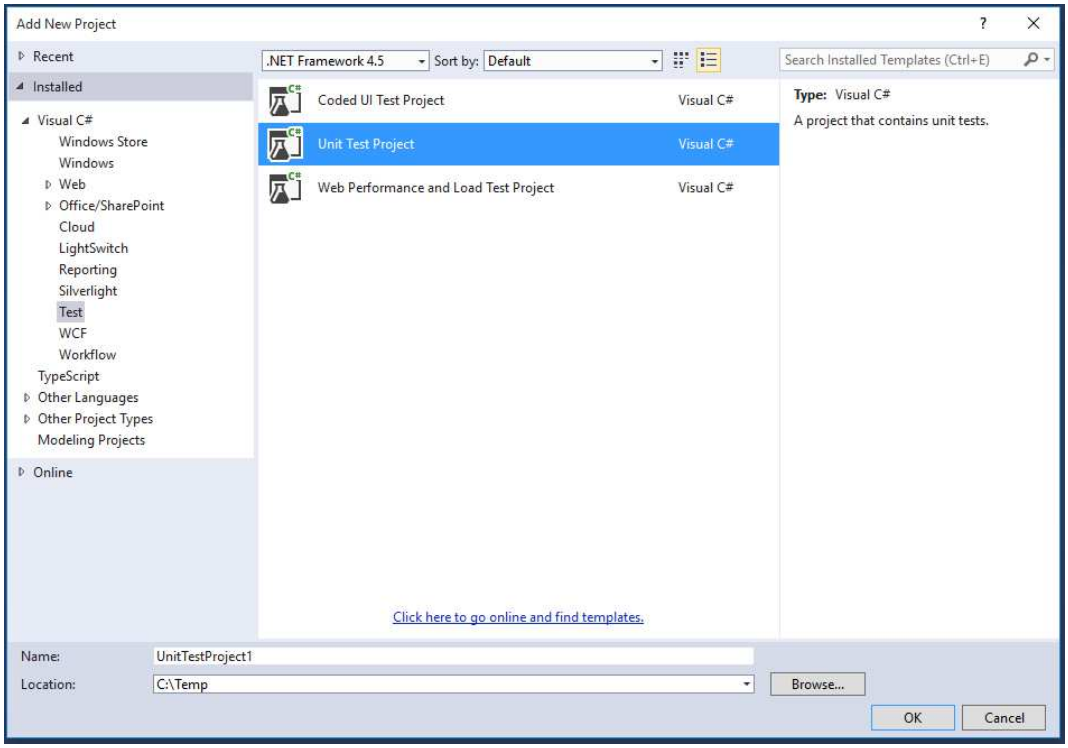
## 3.4 Namngivning

För att när man i framtiden ser på testerna snabbare ska förstå vad de hade för funktion är en klar namngivning viktig. Ett bra koncept ifall det är frågan om en metod är att använda ursprungliga metodnamnet som grund och sedan fylla på med scenariot. T.ex. om man har en metod som heter GetConfigCode så döper man testmetoden till TestGetConfigCode. Om man istället skrev ett enhetstest för att kolla ett objekt används Check istället före objektnamnet t.ex. CheckRecordTarget.

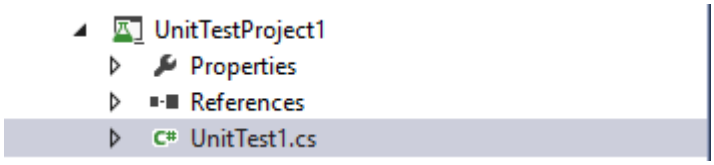
Namngivningen på projekt gjordes så att testprojekten var i samma namnrymd som koden de testade men med Tests på slutet, t.ex. testprojektet till Abilita.Kanta.CDA fick heta Abilita.Kanta.CDA.Tests. En sådan namngivning gör att det blir klarare vart testerna hör när man ser dem i Visual Studio.

## 3.5 Hur man skapar enhetstester

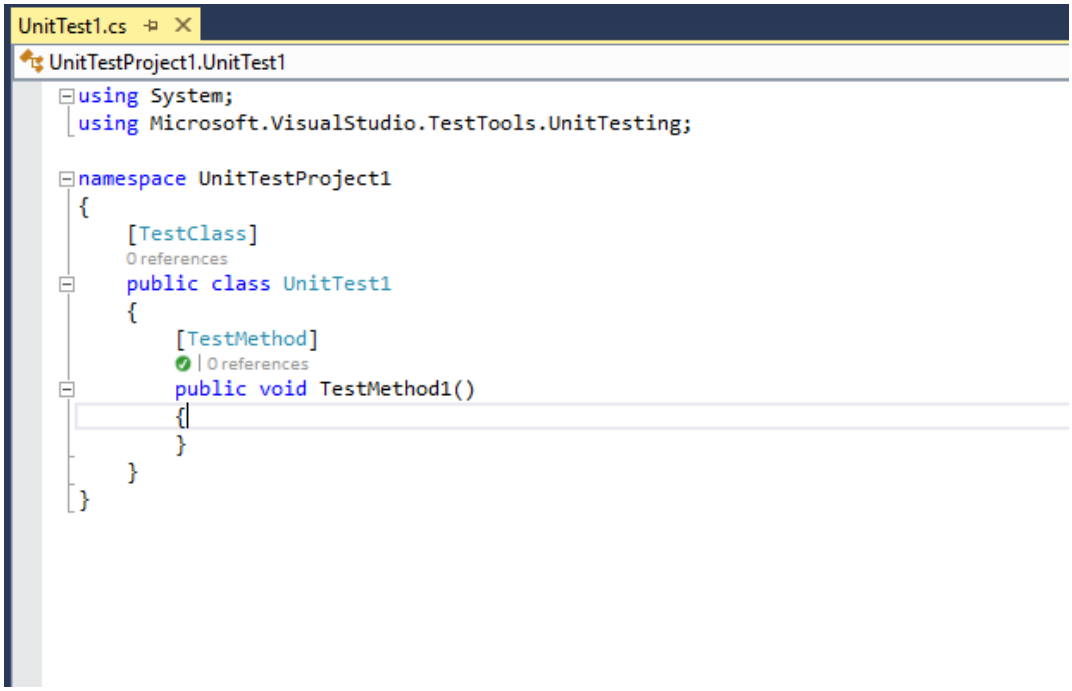
Här visas hur man skapar och kör enhetstester i Visual Studio. Exemplen här visas i Visual Studio 2013 Ultimate.



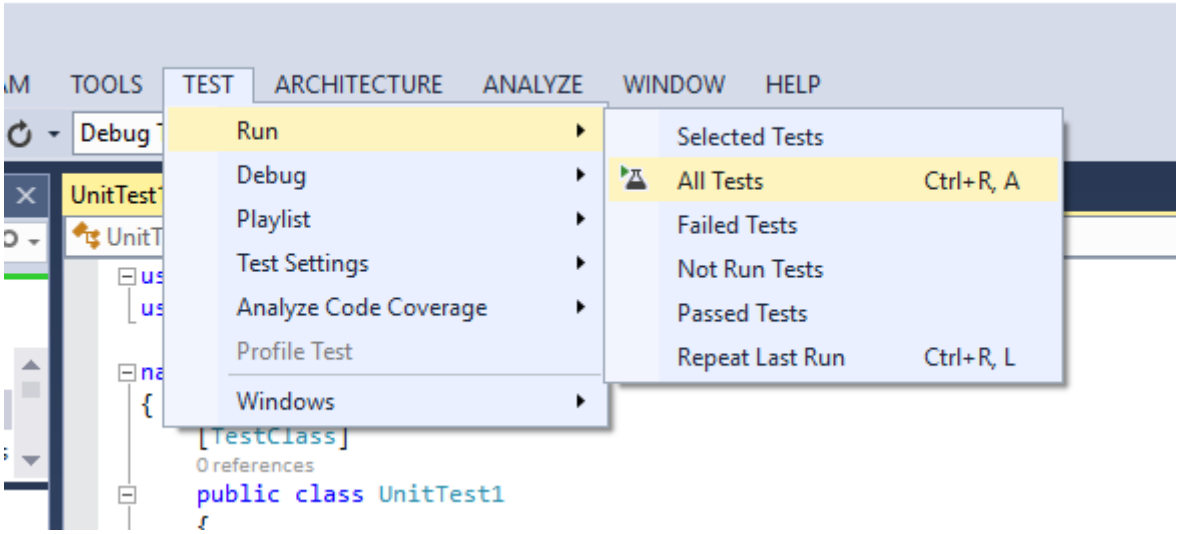

*Figur 3. Man börjar med att skapa ett Unit Test Project.*



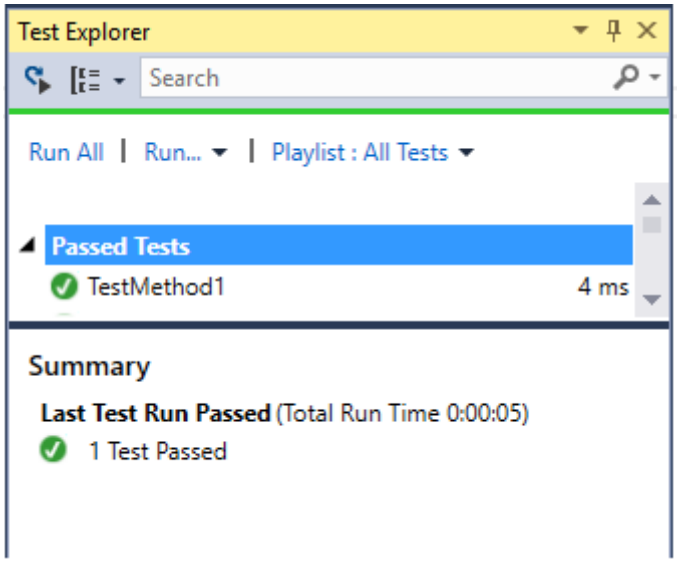

*Figur 4. Ett projekt blir skapat med en färdig testklass.*




*Figur 5. Testklassen får färdigt attribut som anger att det handlar om en testklass med en testmetod. Klassen får också en referens till assemblyn UnitTesting där metoder som behövs för att testa finns. För att kunna testa metoder i andra projekt behövs referenser dit också.*




*Figur 6. För att köra tester används Test -> Run -> All Tests. När man kör tester kompilerar Visual Studio projektet och kör alla test som hittas.*




*Figur 7. Resultatet syns sedan i Test Explorer. Testerna visas på metodnivå och ifall testet misslyckas finns även felmeddelande och info om vilken rad som fallerade.*

# 4 Implementation

Programmeringen utfördes vid Abilitas kontor i Jakobstad under sommarpraktiken. Medlemmar ur projektgruppen satt nära varandra så det gick smidigt att samarbeta. Det förklarades hur olika delar var menade att fungera så testerna kontrollerade rätt saker. Det hände också att buggar hittades med hjälp av testerna.

## 4.1 Planering

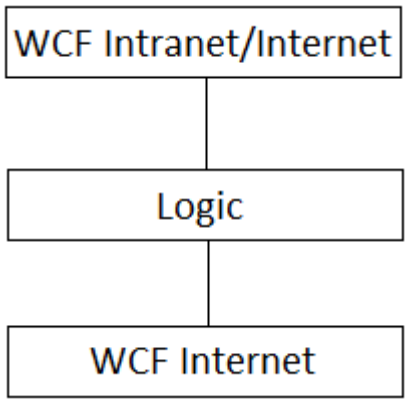
Planeringen skedde i form av veckomöten varje fredag med hela projektgruppen. På mötena gicks det igenom vad som gjorts på projektet sedan förra veckan, vad som bör göras inkommande vecka samt vad som är viktigast att få klart först. Genomgången gick till så att det gicks igenom både i det stora hela vad som är oklart men också så att var och en berättade vad de gjort och man fick samtidigt enskilda instruktioner.

Projektgruppen bestod av projektchefen och sex utvecklare. Två av utvecklarna programmerade användargränssnittet i patientvårdsprogrammet som behövde nya skärmar och integrerade programmet med det nya tillägget. Patientvårdsprogrammet är programmerat i Delphi. Planeringen sköttes av två andra utvecklare som även programmerade lite. Det krävdes mycket planering p.g.a. de strikta kraven från FPA.

De två sista var två utvecklare som skötte programmeringen av själva tillägget i .NET. I början var även en till utvecklare från företaget med som .NET-expertkonsult.

## 4.2 Projektstart

Arbetet med projektet inleddes med att programmet gemensamt sattes upp i Visual Studio. Programmet delades upp i många olika projekt och i tre huvuddelar, WCF Intranet/Internet, Logic och WCF Internet (*Figur 8*).




*Figur 8. Uppdelningen av programmet i tre huvuddelar.*

WCF Intranet/Internet:
Projekten i den här delen av programmet innehåller webbservicen mellan Abilitas patientvårdsprogram och logikdelen.

Logic:
I logikdelen finns det mesta av koden. Här finns logiken för hur data omvandlas till och från CDA-formatet (se kap 2.5) och alla databashämtningar.

WCF Internet:
I WCF Internet finns webbservicen på andra sidan logiken, dvs den som sköter kommunikationen mellan logiken och Kantas webbservice. Trafiken är krypterad.

Enhetstester finns bara i logikdelen eftersom det var där koden som skulle enhetstestas fanns. Man kan också köra enhetstester på webbservice men det var inte av intresse i det här fallet.

För att få lite förståelse för programmet så inleddes programmeringen med lite test-hämtningar. Hämtningarna startades från ett konsolprogram som fungerade som klient i WCF Intranet/Internet-delen. För att åstadkomma ett svar behövdes kod i alla delar av programmet och det här gav snabbt en bild av hur det hela skulle fungera till slut.

## 4.3 Forskning

Enhetstester var obekanta för mig när projektet inleddes så var tvungen att läsa på en del innan programmeringen kunde börja. Eftersom programmet programmerades i C#, som är Microsofts produkt, så fanns det mycket dokumentation lättillgängligt på MSDN (Microsoft Development Network). På MSDN fanns bra exempel på hur man använder sig av de olika klasserna som beskrevs i kapitel 3.3. När man såg exempel på hur små delar man skulle testa i gången så var det lättare att förstå hur enhetstester skulle tillämpas på det här komplexa projektet.

## 4.4 Infoblanketten

För att kunna köra enhetstester på infoblanketten behövdes först ett blankettobjekt att jobba mot. Blankettobjektet skapades genom att anropa en fabriksklass som skötte serialiseringen av objektet till ett XML-baserat CDA-objekt (se kap 2.5). CDAFactory, som fabriksklassen hette, tog ett objekt av klassen eArkivConfig som inparameter. För att få fabriksklassen att fungera behövdes en instans av eArkivConfig med de obligatoriska attributen ifyllda. eArkivConfig innehöll data som t.ex. patientens namn, information om servicestället och information om vem som fyllt i blanketten.

Serialiseringen använde sig även av tre olika tjänster varifrån CDAFactory fick värden till olika attribut. Tjänsterna var OrgOidService, ParameterService och ConfigService. OrgOidService gav information om organisationen var blanketten skapas. ParameterService och ConfigService var lite mera generella och innehöll en rad olika värden. ConfigService innehöll t.ex. olika versionsnummer på blankettdelar och behandlingskoder. Senare i produktion så hämtas förstås dessa tjänster data från en

databas men det steget hoppades över för att få bättre kontroll över vad som matades in samt för att man inte ska blanda in databaser i enhetstester. För att få simulerat databasen anropades tjänsternas *Preload*-metod för att ladda in värden till deras cache. Från cache hämtade sedan CDAFactory-värden till blanketten.

Det blev en hel del kod endast för att få skapat ett blankettobjekt så initialiseringen av blanketten flyttades tidigt till en skild klass som kallades *InfoFormTestInitializer.*

Efter initialiseringen av blanketten kunde själva testandet påbörjas. Testningen gick till största delen ut på att kontrollera blankettobjektets XML-struktur och värden. Strukturen kontrollerades genom att varje gång man gick ner en nivå på objektet kontrollera att nivån inte saknade värde med metoden *IsNull*. Man måste kontrollera en nivå i taget för att undvika ett undantag i testet ifall en högre nivå saknas och man försöker hämta en nivå lägre ner.

*Kodexempel 8. Exempel på kontroll av strukturen*

```
Assert.IsNotNull(recordTarget);
Assert.IsNotNull(recordTarget.patientRole);
Assert.IsNotNull(recordTarget.patientRole.patient);
Assert.IsNotNull(recordTarget.patientRole.patient.name);
```

Värdena på nivåerna kontrollerades genom att jämföra värdet med vad det borde vara genom att anropa metoden *IsTrue*. Värdena kontrollerades på i huvudsak tre olika sätt. Sättet det kontrollerades på berodde på hur fabriksklassen skapat nivån. Första sättet var genom att jämföra värdet mot det som matats in i eArkivConfig-objektet som användes när blanketten serialiserades.

*Kodexempel 9. Exempel på kontroll mot eArkivConfig.*

```
Assert.IsTrue(id.extension == _config.PatReg.Signum);
```

Andra sättet var att kontrollera mot ett värde som laddats in i en av tjänsterna.

*Kodexempel 10. Exempel på kontroll mot en tjänst.*

```
Assert.IsTrue(cda.confidentialityCode.code
        == ConfigService.GetConfigValue("Header", "ConfidentialityCode"));
```

Tredje sättet var att kontrollera mot ett hårdkodat värde eller en enum. Dessa två är i princip samma sak men det är programmeringsmässigt snyggare att programmera in strängarna i en enum istället för att hårdkoda direkt i koden.

*Kodexempel 11. Exempel på kontroll mot en enum.*

```
Assert.IsTrue(cda.relatedDocument.Any(d => d.parentDocument.classCode
        == ActClinicalDocument.DOCCLIN));
```

## 4.5 Envelope

Envelope är i princip ett kuvert till ett CDA-dokument. Generellt kallas kuvertet för SOAP-protokoll. Kuvertet innehåller information om dokumentet som det omsluter som t.ex. vem som skickat dokumentet och vem mottagaren är.

För att kunna testa Envelope behövdes ett Envelope -objekt. Objektet skapades på dylikt sätt som infoblanketten dvs. med en fabriksklass. Envelopes fabriksklass kallades EnvelopeFactory. EnvelopeFactory hade lite annorlunda inparametrar. Inparametrarna var en serviceförfrågan, ett infoblankettobjekt, ett eArkivConfig-objekt och en id för kuvertet. Serviceförfrågan var bara en sträng med en serviceförfrågningskod. Infoblankettobjektet och eArkivConfig-objektet initialiserades med samma kod som i infoblanketten så där sparades det in en hel del tid.

Testandet av Envelope är direkt jämförbart med infoblankettens tester och tas inte upp skillt.

## 4.6 Mindre tester

Förutom infoblanketten och Envelopes tester gjordes även enhetstester till de olika tjänsterna som användes i initialiseringen av blankettobjektet. Det gjordes tester till totalt fem olika tjänster. Enhetstesterna till dem blev ganska lika. De gick i stort sett ut på att testa att tjänsten initialiseras korrekt, att man inte kan initiera dem flera gånger i samma instans samt att testa deras hämtningsfunktioner. Testningen av hämtningar gick ut på att först köra *Preload* och sedan försöka hämta informationen med olika parametrar för att se att rätt information fanns i rätt fält. Kodexemplet i *Kodexempel 12* visar ett simpelt enhetstest av en tjänst.

*Kodexempel 12.*

```
[TestMethod]
public void TestGetConfig()
{
    _configService.Preload(new Config()
    {
        Section = "Header",
        Key = "key1",
        Value = "TestValue"
    });

    var config = ConfigService.GetConfig("Header", "key1");

    Assert.IsNotNull(config);
    Assert.IsTrue(config.Value == "TestValue");
}
```

I exemplet körs *Preload* direkt i testmetoden, i riktiga projektet kördes *Preload* i testklassens konstruktor. Testet laddar in ett objekt med testdata till tjänsten och hämtar sedan objektet med värdet på sektionen och nyckeln. Objektet testas sedan med två Assert-metoder. Den första, *IsNotNull,* kontrollerar att objektet inte är null vilket är bra att alltid köra först eftersom testet returnerar ett undantag istället för misslyckat testresultat om man försöker hämta attribut från ett objekt som är null. Den andra metoden, *IsTrue,* kontrollerar att värdena man sätter in i metoden är lika. Som namnet säger returnerar den då sant, i annat fall falskt.

# 5 Resultat och diskussion

Här presenteras resultatet, möjligheter till vidareutveckling, problem som uppstod samt diskussion.

## 5.1 Resultat

Resultatet av examensarbetet blev cirka 70 mera och mindre avancerade enhetstester som kan köras vid ändringar av programkoden för att se att funktionerna fungerar som tidigare. Informationsblanketten och Envelope blev två mera avancerade kollektioner av tester som både skapar ett blankettobjekt och sedan kör en serie tester för att kontrollera att de genererats korrekt.

## 5.2 Utmaningar

Första utmaningen var förstås storleken på projektet i och med att jag inte varit involverad i något lika omfattande projekt tidigare. Det underlättade att få vara med från början och se det byggas upp från grunden, istället för att hoppa in mitt i.

Ett problem var att det i projektet fanns typer som hette och egentligen var samma sak, men fanns i två olika namnrymder. Ifall man hade refererat till båda namnrymderna i, i mitt fall, ett test så uppstod fel när inte Visual Studio förstod vilken klass som skulle användas. Detta löstes med nyckelordet *global* och hela sökvägen till rätta klassen. Orsaken till att det fanns två olika klasser var att den ena fanns i logikdelen och andra i ena webbtjänstdelen.

Ett annat problem var att eftersom det inte alltid var exakt färdigt tolkat hur saker skulle vara uppbyggda, var vi tvungna att utföra vissa ändringar både i koden och i testerna. Detta bröt då mot regeln att man inte får ändra färdigt skrivna test, men eftersom det handlade om ändringar i strukturen av blanketten eller varifrån data skulle hämtas som blev blivit feltolkat, så fanns det förstås inget annat val än att ändra på båda ställena.

## 5.3 Vidareutveckling

Det fanns under tiden jag jobbade med examensarbetet bara den ena blanketten och dess kuvert samt de tjänster jag programmerade tester till. I och med att jag fick dem klara så är testerna fullständiga för tillfället. I framtiden kommer det att komma flera blanketter som då i sin tur behöver egna tester programmerade. Jag visade när jag var klar åt programmeraren som programmerade koden hur man skriver enhetstester, så att han skulle ha möjlighet att programmera egna tester på kommande blanketter och testbar framtida kod.

## 5.4 Diskussion

Det var lärorikt att vara involverad i ett det här projektet. Det var intressant att se hur man gör när man planerar och sedan verkställer ett projekt som omfattar flera månader. Utvecklingen av de olika delarna gick snabbt framåt eftersom det var flera programmerare inblandade och det var fascinerande att se hur otroligt mycket programkod det behövdes för att få programmet att fungera.

I och med den här erfarenheten förbättrades mina programmeringskunskaper avsevärt. Jag kunde från förut grunderna i C#, men lärde mig använda mera avancerade saker i och med det här projektet. Var lite osäker före om jag skulle klara detta, men det gick bra och nu efteråt blev jag mycket mera självsäker i min programmering. Som extra uppgift fick jag även ta reda på och dokumentera hur alla delarna i programtillägget fungerade och exakt vad som görs i dem. Detta gav ännu mera kunskap om hur ett stort .NET-program fungerar och är uppbyggt.

# 6 Källförteckning

/1/ Oy Abilita Ab. [Online]
http://www.abilita.fi
[hämtat: 13.2.2014]

/2/ C Sharp (programming language). [Online]
http://en.wikipedia.org/wiki/C_Sharp_(programming_language)
[hämtat: 22.2.2014]

/3/ Microsoft Visual Studio. [Online]
http://en.wikipedia.org/wiki/Microsoft_Visual_Studio
[hämtat: 27.3.2014]

/4/ .NET Framework. [Online]
http://en.wikipedia.org/wiki/.NET_Framework
[hämtat: 27.3.2014]

/5/ Language Integrated Query. [Online]
http://en.wikipedia.org/wiki/Language_Integrated_Query
[hämtat: 28.4.2014]

/6/ Microsoft.VisualStudio.TestTools.UnitTesting. [Online]
http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.aspx
[hämtat: 28.4.2014]

/7/ Unit Testing. [Online]
http://msdn.microsoft.com/en-us/library/aa292197%28v=vs.71%29.aspx
[hämtat: 6.5.2014]

/8/ The Art of Unit Testing. [Online]
http://artofunittesting.com/
[hämtat: 6.5.2014]

/9/ Watkins, J. 2010. *Testing IT – An off-the Shelf Software Testing Process.* Cambridge University Press.

/10/ Integration Testing. [Online]
http://msdn.microsoft.com/en-us/library/aa292128%28v=vs.71%29.aspx
[hämtat: 17.5.2014]

/11/ Freeman, A. and Rattz J.C. Jr 2010. *Pro LINQ - Language Integrated Query in C#.*

/12/ Hunt, A. and Thomas, D. 2007. *Pragmatic Unit Testing In C# with NUnit.*

/13/ CDA® Release 2. [Online]
http://www.hl7.org/implement/standards/product_brief.cfm?product_id=7
[hämtat: 9.1.2016]

/14/ Boone, K.W. 2011. *The CDA TM book.*

/15/ HL7 Finland ry. *HL7 CDA paikallistamisprojekti Soveltamisopas*. [Online]
http://www.kanta.fi/web/ammattilaisille/hl7
[hämtat: 9.1.2016]