Laboration III
API-utveckling

I denna laboration så kommer vi att arbeta med s.k. API-utveckling och mer exakt hur kommunikation kan ske mellan olika system. För att uppnå detta så kommer vi att skapa och implementera en webservice för vilken syftet är att hämta och transformera data från en extern systemtjänst (API). Den externa systemtjänsten erbjuder data i ett JSON-format men vi behöver i vår klient kunna arbeta med denna data i ett XML-format. Man kan därför tänka sig att domänen ser ut enligt nedan:

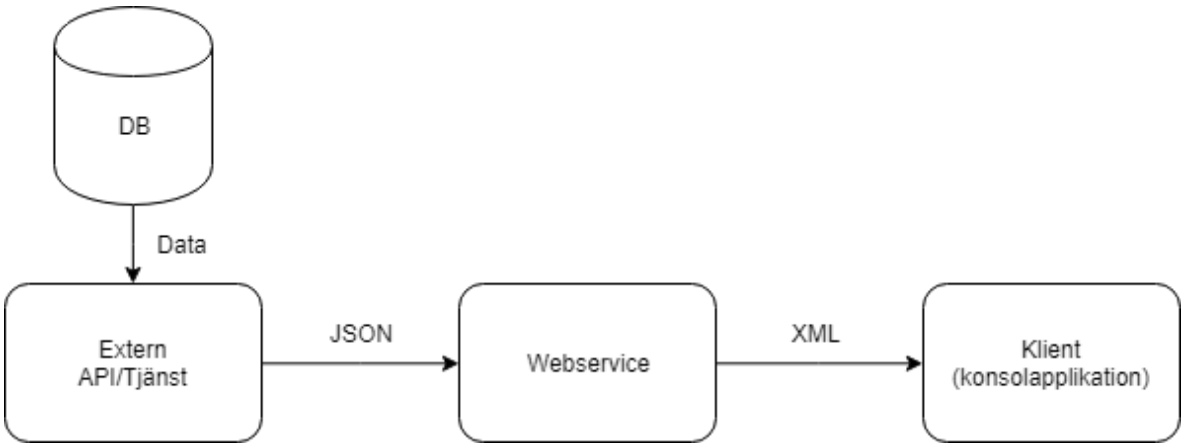

Fig. 1 – Domänen vi arbetar med.

I denna laboration så får man arbeta individuellt eller i par. **Notera** dock att om man arbetar i par så måste båda deltaga vid examinationen som sker muntligt den 18/3.

Innehållsförteckning

1 Introduktion

Vi har tidigare beskrivit hur domänen ser ut, men inget om funktionaliteten. En beskrivning av vad tjänsten och klienten förväntas kunna utföra beskrivs därför nedan.

WCF-service (tjänsten) Tjänsten ska ha följande funktionalitet:

- Hämta information från den externa tjänsten.
- Transformera denna information till ett XML-format från JSON.
- Exponera 6 metoder; `GetTestData`, `GetAllInterchanges`, `FilterByInterchangeID`, `FilterByInterchangeNode`, `FilterByInterchangeIDAndNode` och `FilterByInterchangeNodeValue` som klienten kan nyttja.

Konsolapplikation (klienten) Klienten ska ha följande funktionalitet:

- Hämta information från vår tjänst (som avses ovan).
- Spara data i olika filformat, t.ex. xml och txt.
- Öppna och arbeta med redan existerande filer.
- Exponera en iterativ meny vilken erbjuder användaren att genomföra ovan funktionalitet.

Uppgift 1 Skapa klienten och tjänsten

Du börjar med att skapa en ny konsolapplikation i Visual Studio. Vi kommer inte att göra något med konsolapplikationen nu i början, men denna applikation kommer att agera klient till den tjänst som du strax ska bygga. För att bygga denna tjänst så behöver vi lägga till ett nytt projekt till vår "Solution". Detta gör du enklast genom att trycka på File -> Add -> New Project. Därefter så väljer du alternativet "WCF Service Application". **OBS!** Kom ihåg att ge era projekt meningsfulla namn redan när ni skapar dem då det alltid är struligare att göra detta i efterhand.

Efter att du valt projekttyp och givit ett namn för projektet så kommer Visual Studio att generera ett antal filer efter att du klickat på "OK". Bland dessa filer så är det enbart två som vi kommer att arbeta med. Dessa är: `Service1.cs` och `IService1.svc`. Det är i `Service1.svc` som vi kommer att implementera funktionaliteten hos tjänsten. Notera även att denna klass implementerar interfacet `IService1`, vilket i sin tur representerar de funktioner som tjänsten exponerar till dess klienter. I detta interface så behöver vi därför

skriva metodsignaturen för de 5 metoder som vi vill exponera till vår klient. **OBS!** För både `Service1.svc` och `IService1.cs` så ska ni inte behålla det autogenererade namnet. Figur 2, 3 och 4 visar hur man döper om dessa på bästa sätt.

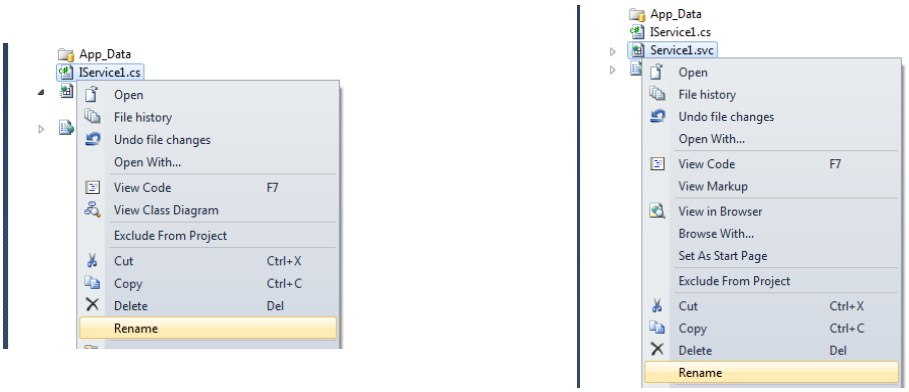

Fig. 2 – Rename via Solution explorer.

Notera att när interfacet döps om så frågar Visual Studio om du vill uppdatera samtliga referenser i koden till det nya namnet, klicka då på "Yes" så slipper du att manuellt ändra det i koden.

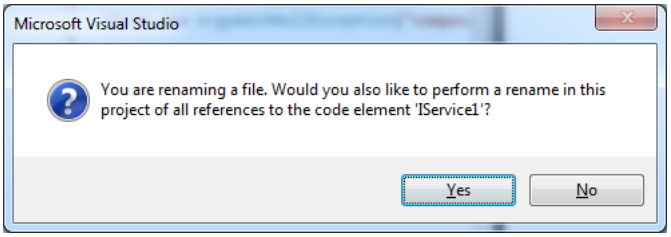

Fig. 3 – Referensuppdatering vid namnbyte.

När du ändrar namnet för klassen `Service1.svc` så rekommenderar vi att du gör detta med ovan metod och från själva klassen. Du gör detta genom att högerklicka på klassnamnet -> tryck på Rename och skriv därefter in namnet som du vill ändra till (Figur 4). Ändringen genomförs sedan när du klickar på "Apply" i rutan som dyker upp.

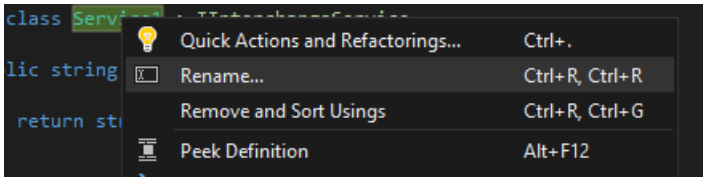

Fig. 4 – Namnbyte från klassen.

Nu när vi är klara med namnbytet så märker du säkert att det finns viss autogenererad kod i klassen och interfacet. Du får gärna titta på samt testköra koden om du vill, men denna kod ska därefter tas bort då den inte tillför något till uppgiften.

2 WCF service

Börja med att öppna interfacet och notera att det har ett attribut, `ServiceContract`. Detta attribut säger att det vi definierar i interfacet är vad vi kommer att exponera till våra klienter. Metodsignaturerna deklareras därefter precis som vanligt med metodnamn, ev. parametrar osv. Ett undantag är dock att vi även behöver tilldela `OperationContract`-attributet till respektive metodsignatur.

Uppgift 2 Deklarera metoder i interfacet

Deklarera nu samtliga metoder som tjänsten ska exponera:

```
GetTestData ()
GetAllInterchanges ()
FilterByInterchangeID (int id)
FilterByInterchangeNode (string node)
FilterByInterchangeIDAndNode (int id, string node)
FilterByInterchangeNodeValue (string node, string value)
```

För metoderna `GetTestData` och `GetAllInterchanges` så förväntas dessa metoder skicka tillbaka antingen testdata eller samtliga "interchanges". För `FilterByInterchangeID`, `FilterByInterchangeNode`, `FilterByInterchangeIDAndNode` samt metoden `FilterByInterchangeNodeValue` så ser vi ovan att dessa tar emot vissa parametrar. Dessa parametrar kommer sedan nyttjas för att filtrera vår data och skicka tillbaka resultatet av filtreringen.

Funktionaliteten hos samtliga metoder beskrivs ytterligare i Uppgift 7. Notera att dessa metoder förväntas returnera ett `XElement`. Vi bör därför ha något som efterliknar nedan:

```
[ServiceContract]
public interface INameOfInterface {

        [OperationContract]
        XElement MethodName1();

        [OperationContract]
        XElement MethodName2();

        // osv.
}
```

2.1 SOAP och XML

En kortfattad förklaring av hur WCF services fungerar är att kommunikationen mellan klienten och tjänsten genomförs via SOAP (Simple Object Access Protocol). SOAP möjliggör kommunikationen mellan två system genom att komma överens om hur de ska anropa varandra. Denna överenskommelse sker via ett s.k. "servicecontract". I vårt fall, som vi kan se i föregående uppgift, så ingår klienten och tjänsten ett avtal att kommunicera med typen `XElement`, dvs. XML. Vårt interface agerar alltså detta "servicecontract" mellan klienten och servicen.

3 Implementation av servicen

Men vi kan inte på något magiskt vis börja kommunicera mellan klienten och tjänsten utan vi behöver först lägga till en koppling mellan dem. Detta gör vi genom en s.k. "service reference". **OBS!** Denna referens ska enbart läggas till på klientsidan. Vi säger alltså att klienten ansluter till tjänsten, inte tvärtom.

Uppgift 3 Lägg till en service referens

Hur man går till väga för att lägga till en service reference samt ev. problem som kan uppstå vid tillägg av sådana behandlas i dokumentet "Introduktion_WCFServices".

Uppgift 4 Implementera funktionalitet i servicen

Nästa steg är att implementera våra service-metoder (som hittills enbart bör ha `throw new NotImplementedException();` inom dess måsvingar). Det första steget i denna process blir att skapa en koppling till den externa API:n som vi vill hämta informationen ifrån. Problemet är dock att den API-adress som vi har till den externa tjänsten är krypterad. Således behöver vi först dekryptera adressen (1), säkerställa att formatet i filen hos adressen är korrekt (2) och sedan ladda ned innehållet från filen (3). För att göra detta så kommer vi att nyttja följande kod:

```
using (WebClient webClient = new WebClient())
{
    webClient.DownloadString(Encoding.UTF8.GetString(
                Convert.FromBase64String("<Adressen Till API:n>")));
}
```

Här är det möjligtvis mycket nytt samtidigt, men kortfattat så skapar vi en ny instans utav en `WebClient`. `WebClient` är en klass som bl.a. tillhandahåller

funktionalitet för att skicka och hämta data över en nätverksresurs (t.ex. lokalt, intranät eller internet). Notera att vi gör detta i ett s.k. "using-statement". Anledningen till varför man nyttjar `using` är för att vi enbart vill nyttja detta objekt i denna metod, efter det så vill vi inte att objektet ska ligga och skräpa i minnet. `using` gör så att vi automatiskt kallar på metoden `Dispose`, vilket kastar objektet när vi är klara med det.

Vi använder oss sedan av en `WebClient`-metod, `DownloadString`. Denna metod hämtar informationen från en nätverksresurs (den okrypterade strängen) som en string (3). I metodanropet så nyttjar vi ännu en metod, `Encoding.UTF8.GetString`, för att avkoda en array av bytes till en okrypterad string med UTF8-format (2). `Encoding.UTF8.GetString` nyttjar i sin tur metoden `Convert.FromBase64String`, vilken konverterar en krypterad string till en osignerad array av bytes (1).

Notera att en nätverksresurs identifieras av en URI (vilket är "Adressen till API:n" i kod-exemplet ovan). För att hålla det så övergripligt som möjligt så kan vi tänka oss att en URI praktiskt taget är en webbadress.

OBS! Ni behöver inte nödvändigtvis förstå ovan kod, men den är väsentlig för implementationen då vi kommunicerar med vår externa service mha. två stycken hashade adresser. Koden är därför behövlig för att avkoda och hämta data från dessa adresser.

Det finns totalt två resurser att hämta hos den externa systemtjänsten och dessa är:

- ics.json, som innehåller samtliga interchanges.
- testData.json, som innehåller två stycken interchanges. Dess enda syfte är att kunna testa implementationer utan att behöva navigera en stor mängd data.

Resursernas resp. hashade adress är:

```
ics: aHR0cDovL3ByaXZhdC5iYWhuaG9mLnNlL3diNzE0ODI5L2pzb24vaWNzLmpzb24=
test:
    aHR0cDovL3ByaXZhdC5iYWhuaG9mLnNlL3diNzE0ODI5L2pzb24vdGVzdERhdGEuanNvbg==
```

3.1 Newtonsoft

Vår tjänst behöver dock kunna transformera datan från dessa resurser från ett JSON-format till ett XML-format. Lyckligtvis så finns det ett väldigt smidigt paket vi kan nyttja för detta kallat Newtonsoft.Json.

Uppgift 5 Installation av Newtonsoft

Men Newtonsoft.Json följer inte automatiskt med när vi skapar ett nytt projekt i Visual Studio så vi behöver därför först lägga till det i vårt projekt. Detta gör vi genom att nyttja Packet Manager Console.

Du hittar denna konsol genom att navigera till Tools -> välj NuGet Package Manager -> tryck på Packet Manager Console. Figur 5 visar hur du hittar hit.

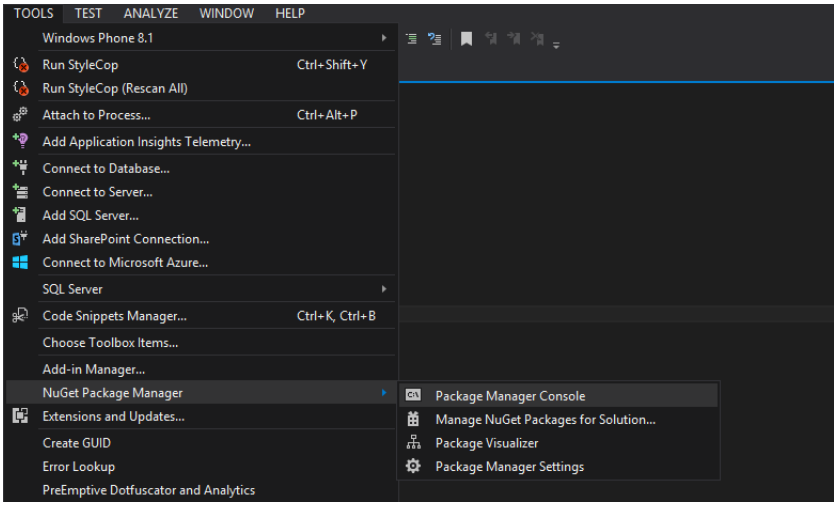

Fig. 5 – Package Manager Console.

Vänta därefter på att PowerShell startar upp. Vi vet att PowerShell är igång när "PM>" dyker upp i konsolen. Skriv då in: `Get-Project -All Install-Package Newtonsoft.Json`.

För att sedan använda Newtonsoft så lägger vi helt enkelt till följande using-referens i service-klassen där vi vill använda det:

```
1 using Newtonsoft.Json;
```

Uppgift 6 Arbeta med Newtonsoft

Nu när vi har installerat vårt paket så behöver vi skapa de variabler i vilka vi vill lagra informationen som vi hämtar från den externa API:n. Då vi sannolikt kommer att nyttja dessa variabler i fler än en metod så skapar vi instansvariabler enligt följande:

```
1 static XElement _testData;
2 static XElement _interchanges;
```

Notera att du till en början enbart vill arbeta med `_testData`. Anledningen är, som tidigare nämnt, att det är enklare att arbeta med mindre data under utvecklingsfasen. Men vi kan inte enkelt tilldela ett XElement värdet utav en string (vilket är vad vi hämtar från den externa API:n, denna string erhåller dessutom ett JSON-format). Vi behöver därför utföra någon form utav konvertering eller "parsing" och det är här Newtonsoft kommer in.

```
1 XElement xe = JsonConvert.DeserializeObject<XElement>(<string i
      json-format>);
```

I ovan exempel så nyttjar vi Newtonsoft (JsonConvert.DeserializeObject<>) för att konvertera en string i json format till ett XElement och lagrar det i den nyskapta variabeln xe. Vi nyttjar alltså vår parser mer eller mindre på samma sätt som vi normalt skulle nyttja exempelvis `Convert.ToInt32` eller dylika metoder.

För att återkoppla till Uppgift 4, där vi upprättade en koppling till den externa API:n, så kan vi kombinera det kodstycket med ovan kod för att först hämta ut en string i JSON-format och därefter direkt konvertera det till ett XElement. Exempelvis enligt:

```
1 string jsonString = webClient.DownloadString(
2 Encoding.UTF8.GetString(Convert.FromBase64String("<Adressen Till
      API:n>"))));
3 _testData = JsonConvert.DeserializeObject<XElement>(jsonString);
```

Genom att följa exemplet ovan, tilldela nu instansvariablerna `_testData` och `_interchanges` dess korrekta värden. Tänk på att respektive instansvariabel måste nyttja dess angivna URI. Dessa finns på s. 5 i dessa instruktioner.

3.2 Service funktionalitet

För att få en bild av hur XML-filen är uppbyggd så finns det ett exempel, interchangeExample.xml, tillgängligt för nedladdning där du hittade dessa instruktioner. Vi rekommenderar att du har denna fil öppen samtidigt som du arbetar med filtreringsfunktionaliteten.

Uppgift 7 Implementation av filtreringsfunktionalitet

Det är nu dags för att implementera våra servicemetoder. För metoderna `GetTestData` och `GetAllInterchanges` så är implementationen enkel, i den ena metoden skickar vi tillbaka den ena instansvariabeln (för testdata) och i den andra metoden så skickar vi tillbaka den andra instansvariabeln (för samtlig data). För metoderna `FilterByInterchangeID`, `FilterByInterchangeNode`, `FilterByInterchangeIDAndNode` och `FilterByInterchangeNodeValue` så förväntas du dock nyttja LINQ-to-XML för att filtrera data. `FilterByInterchangeID` tar emot en int och ska returnerna den interchange som matchar det ID:et. `FilterByInterchangeNode` tar emot en string som representerar namnet på en nod och returnerar dessa noder och dess värden. `FilterByInterchangeIDAndNode` tar emot en int och en string och förväntas returnera noderna och dess värden där både ID och nodnamn stämmer överens. Dvs. att om vi exempelvis anropar metoden med:

```
1         FilterByInterchangeIDAndNode(1, "Name")
```

så bör ett `XElement` som innehåller "<Name>Sten Frisk</Name>" skickas tillbaka.

`FilterByInterchangeNodeValue` tar emot två strängar och förväntas returnera de interchanges som noden och det tillhörande värdet förekommer i. Dvs. att om vi exempelvis anropar metoden med:

```
1 FilterByInterchangeNodeValue("Name", "Sten Frisk")
```

så bör ett `XElement` som innehåller samtliga interchanges där "<Name>Sten Frisk</Name>" förekommer som en barnnod.

Notera att om du gör några förändringar i metodsignaturen hos dessa metoder så kommer du att behöva uppdatera både metodsignaturen i interfacet samt själva service referensen. Hur man uppdaterar en service referens beskrivs i dokumentet "Introduktion_WCFService".

Uppgift 8 Testa implementationen

När du har implementerat metoderna i föregående uppgift så kan det vara fördelaktigt att testa huruvida implementationen är korrekt. För att göra detta så måste vi först arbeta lite med klienten. Det första steget är att instansiera servicen i klienten, detta gör man genom att först peka på service referensen och sedan klassen som avslutas med "client". Det ser mao. ut något i stil med:

```
1 ServiceReference.InterchangeServiceClient ic = new
      ServiceReference.InterchangeServiceClient();
```

Vi kan därefter nyttja detta objekt för att kalla på metoden som vi vill testa. Vi har däremot inte någon funktionalitet för att spara ned den data som vi hämtar från servicen till en fil ännu, så hur kan vi enklast kontrollera

att vår implementation fungerar? Det enklaste svaret är att vi gör detta genom debugging i Visual Studio mha. DataTips-verktyget. För att nyttja verktyget så placerar vi en breakpoint för metodanropet som vi vill testa. Vi kan därefter kontrollera innehållet i vår variabel och se om den innehåller den förväntade datan (se Figur 7).

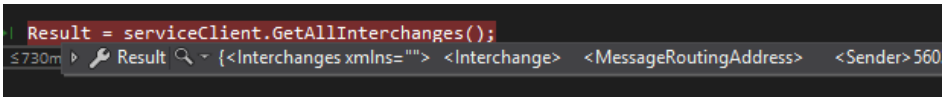

Fig. 6 – DataTips-verktyget.

Det kan däremot vara ganska svårt att tyda bara utifrån den långa raden vi kan se i Figur 7. Ett sätt att lösa detta på är genom att nyttja ett annat verktyg, Data Visualizer. Man får tillgång till detta verktyg genom att klicka på förstoringsglaset som vi kan se i Figur 7. Vi kan därefter välja om vi vill se det i XML-format, JSON-format, HTML eller text. Som vi kan se i Figur 8 är detta väldigt smidigt för just XML.

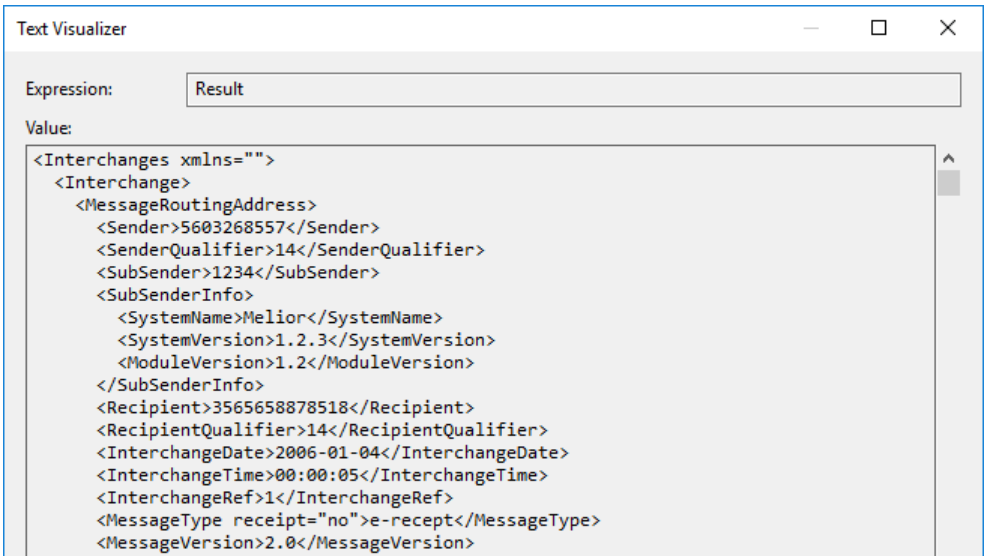

Fig. 7 – Data visualizer verktyget.

Genom att nyttja dessa verktyg, kontrollera att allting ser korrekt ut för resp. metod. **OBS!** Om du försöker hämta alla data (dvs. om vi anropar t.ex. `GetAllInterchanges`) så kommer du först att behöva lägga till ett attibut i klientens App.config-fil. Detta är nödvändigt för att tillåta tjänsten att skicka data av större storlekar. Hur man gör detta beskrivs i dokumentet "Introduktion_WCFService". Det kan även vara så att du stöter på felmeddelandet *"System.Configuration.ConfigurationErrorsException"*. Om så är fallet kan problemet lösas genom att flytta ditt projekt till skrivbordet (X:).

Om allting ser korrekt ut för resp. metod så är vi redo för att börja arbeta med klienten.

4 Implementation av klienten

Det första steget är att skapa en enkel iterativ meny i `Main`-metoden. Iterativ i den mening att användaren ska kunna arbeta med applikationen tills dess att denne avslutar den med ett kommando, t.ex. 'e' för exit. Skapa till en början nog med menyalternativ för att genomföra anrop till samtliga metoder från föregående uppgift samt ett sista för att avsluta applikationen.

4.1 Iterativ meny och interaktion med servicen

Det kan dock tänkas vara lite tokigt att vi instansierar en koppling till servicen direkt i `Program`-klassen. Vi flyttar därför detta till en ny klass, `ConnectToInterchangeService`, som är tänkt att sköta allt som har med kommunikationen till servicen att göra. Dvs. dels instansieringen, men även uthämtningen av data. Klassen ska därför ha 6 st. metoder enligt nedan:

```
1 void GetAll()
2 void GetTesData()
3 void GetFilteredByID(int id)
4 void GetFilteredByNode(string node)
5 void GetFilteredByIDAndNode(int id, string node)
6 void GetFilteredByNodeValue(string node, string nodeValue)
```

Klassen ska även ha en egenskap, `Result`, vilken kommer att innehålla resultatet från det senaste anropet. Dvs. att egenskapen ska tilldelas ett nytt värde för resp. metodanrop. Men vi vill även ha möjligheten att spara exempelvis specifik tidpunkt som informationen hämtades från servicen så att vi enkelt kan avgöra om informationen är utdaterad eller för att enkelt kunna jämföra data mellan två filer. I set-egenskapen för `Result` så vill vi därför lägga till ett XML-attribut i den omslutande noden vilket har värdet av datum och tid då datan hämtades. För att lägga till attribut i ett `XElement` så bör du nyttja `XAttribute`.

Uppgift 9 Uppdatera den iterativa menyn

Nu när vi har implementerat `ConnectToInterchangeService` så blir nästa steg att reflektera detta i menyn. Dvs. att vi exempelvis inte ska instansiera servicen i `Program`-klassen, men att vi bör instansiera `ConnectToInterchangeService`. Uppdatera därefter alla metod-anrop i menyn så att dessa kallar på metoderna i `ConnectToInterchangeService`. Varje gång användaren väljer att anropa en av dessa metoder så **ska resultatet från anropet skrivas ut i konsolen**.

4.2 Filhantering

I nuläget så ska vi kunna arbeta med XML-filer (i form av `XElement`) och hämta data från servicen. Detta sker dock enbart under run-time eftersom vi saknar möjligheten att spara ned och läsa in data från lokala filer. Dvs. att datan går förlorad när vi avslutar applikationen, vilket kan tyckas vara lite poänglöst.

Uppgift 10 FileBackup

Det första steget blir därför att kunna spara ned våra resultat i filer. Vi skapar därför först en klass med syftet att spara ned och läsa in filer, `FileBackup`. För att på ett så enkelt sätt som möjligt kunna arbeta med filer så kommer vi att nyttja inslag av Windows.Forms i vår konsol-applikation. Vi behöver därför lägga till en referens till Windows.Forms i vår applikation. Detta gör man genom att högerklicka på projektet -> navigera till Add -> välj Reference. Identifiera därefter Windows.Forms, klicka i checkboxen och klicka på OK (Figur 9). **OBS!** Om du har problem med att identifiera Windows.Forms referensen så kan du alltid nyttja sökfältet uppe till höger i "Reference Manager"-fönstret.

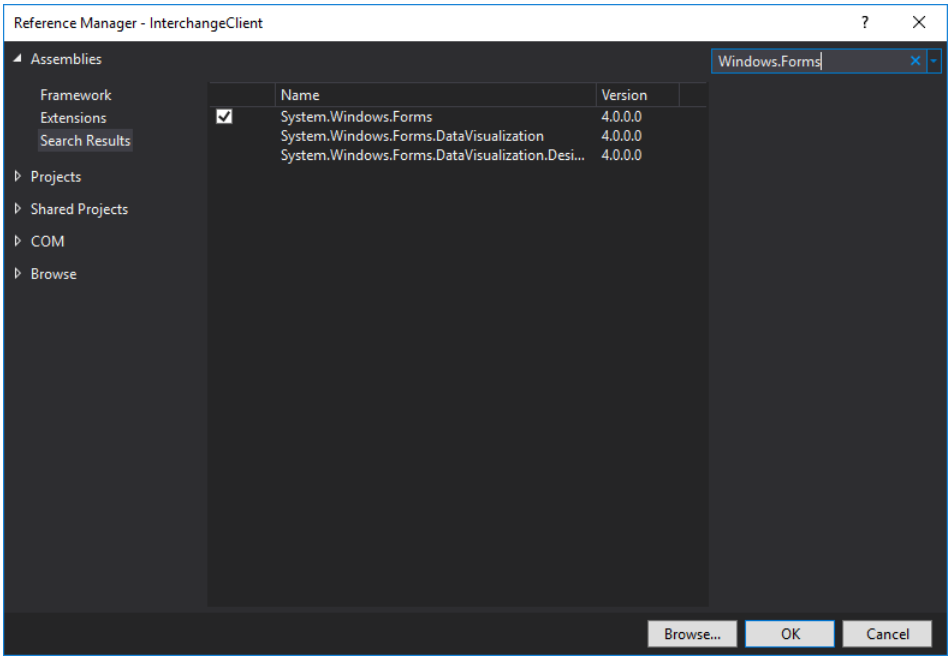

Fig. 8 – Reference Manager.

Lägg sedan till en using referens i klassen för denna referens samt för `System.IO`.

Uppgift 11 Funktionalitet i klassen

När vi nu har lagt till våra referenser så kan vi sedan börja lägga till funktionalitet i klassen. Vi börjar med att deklarera klassens två metoder:

```
1 public static void SaveToFile(string content)
2 public static string LoadFile()
```

För metoden `SaveToFile` så är tanken att denna metod tar emot en string och sedan skriver ned informationen från den till en fil genom att nyttja metoden `File.WriteAllText`. Men vi vill inte alltid skriva till samma fil, med samma namn osv. Vi instansierar därför först och främst klassen `SaveFileDialog`. Vi kan sedan nyttja detta objekt för att kalla på metoden `ShowDialog` vilket kommer att resultera i att ett fönster öppnas, precis som när man försöker spara en vanlig textfil från exempelvis Notepad. Användaren kan därifrån välja vart filen ska sparas, vilket format den ska erhålla och namnet på filen.

Vi behöver sedan kontrollera om/när användaren klickar på OK i fönstret och detta gör vi genom att jämföra resultatet från `ShowDialog` mot `DialogResult.OK`. Om detta stämmer överens så innebär det att användaren har valt vart filen ska sparas, format och namn och vi vet således att vi är redo för att spara filen. För att sedan spara filen så kommer vi att nyttja metoden `File.WriteAllText` som tar emot filvägen + namnet och en string som innehåller det som ska sparas ned. Filvägen + namnet kan vi få från vårt `ShowDialog` objekt genom egenskapen `FileName` och den string som innehåller vad som ska skrivas ned är den string som metoden tar emot som argument.

Vi vill alltså ha något i stil med:

```
1 public static void SaveFile(string content)
2 {
3         SaveFileDialog sfd = new SaveFileDialog();
4         OM sfd.ShowDialog() == DialogResult.OK
5         File.WriteAllText(sfd.FileName, content);
6 }
```

Notera att det även finns andra saker vi kan göra med `ShowDialog` såsom att ange tillåtna filformat. I vårt fall så vill vi kunna spara våra resultat som Textfiler, XML-filer eller Any (*).

Uppgift 12 STAThread

Men vi märker ganska snabbt att detta inte fungerar i nuläget då vi får ett `System.Threading.ThreadStateException` om vi försöker öppna fönstret mha. `SaveFileDialog`. Detta beror på att många komponenter i Windows kräver att vi öppnar dessa över en, och enbart en, tråd (single thread apartment, STA). Om vi inte anger detta explicit så kommer vi dock alltid först och främst att försöka öppna dessa över multipla trådar (multi thread apartment, MTA). STA vs MTA är sannolikt främmande för många och det är

inget vi egentligen täcker på denna kurs, men en kortfattad beskrivning är att vi arbetar med STA vid UI komponenter då detta säger att enbart denna tråd kan kommunicera med komponenten och kalla på dess metoder. MTA tillåts inte för UI komponenter. Denna länk förklarar konceptet något mer ingående om man är intresserad: https://stackoverflow.com/questions/127188/could-you-explain-sta-and-mta

I vårt fall så innefattar det att vi behöver lägga till ett `[STAThread]` attribut över vår `Main`-metod (då det är från vår meny som vi kommer att kalla på metoderna).

Lägg sedan till ett meny alternativ för att spara filer där du anropar `SaveToFile`, om du inte redan har gjort det, och kontrollera att detta fungerar.

Uppgift 13 Implementation av LoadFile

Implementationen för `LoadFile` kommer att se mer eller mindre likadan ut som för `SaveToFile`, men istället för en `SaveFileDialog` så vill vi ha en `OpenFileDialog` och vi vill inte skriva utan vi vill istället läsa text. Efter att vi har läst text från en fil så skickar metoden tillbaka en string som innehåller denna text.

4.3 Utskrift i konsolen

I nuläget så bör vi ha en klient som kan kommunicera med vår service, iterera tills användaren väljer att avsluta applikationen, spara ned resultat till filer och läsa in information från filer. När vi väl har läst in en fil så gör vi dock ingenting med denna information i nuläget.

Uppgift 14 PrettyInfoPrint

Vi skapar därför en ny metod, `PrettyInfoPrint`, som är tänkt att skriva ut relevant information från en inläst fil i klartext. Exempelvis kan man tänka sig att man är intresserad av att snabbt kunna identifiera läkaren, patienten, läkemedlet och doseringsbeskrivning utan att behöva navigera en hel XML (en "Interchange") från servicen. Implementationen av denna metod ska ske genom LINQ-to-XML och metoden tar därför emot ett XElement. Utskriften i fråga bör efterlikna nedan:

```
Patient: Name of patient
Physician: Name of physician
Medicine: Medicine ID
Dosage: Dosage description
```

Uppgift 15 Uppdatera menyn

Vi har tidigare implementerat metoden `LoadFile` i klassen `FileBackup` som skickar tillbaka en string (som representerar innehållet i en fil) och `PrettyInfoPrint` som tar emot ett XElement. Vi behöver mao. konvertera innehållet från en fil (string) till ett XElement innan vi kan anropa metoden `PrettyInfoPrint`. För att göra detta så finns det dock en smidig metod som vi kan nyttja: `XElement.Parse`.

Uppdatera menyn så att ovan avsedd funktionalitet fungerar.

Uppgift 16 Felhantering

Nu är vi klar med all funktionalitet i vår applikationen, men självfallet så behöver vi även lite felhantering. Se till så att applikationen inte kan krascha som ett resultat av felaktig användarinput, att användaren kan navigera i menyn (t.ex. gå från "service-läge" till "öppna-fil" läge) och att menyn i fråga åtminstone ser rimligt snygg ut (dvs. släng in `Console.WriteLine()` där det behövs, se till att formatet är läsbart och nyttja gärna `Console.Clear()` vid behov).

5 Redovisning

Säkerställ att allting är korrekt implementerat, att all funktionalitet är uppfylld och att applikationen inte kraschar. Själva examinationen kommer att ske i laborationssal den 18/3 till vilken man anmäler sig via en Doodle-länk. Denna Doodle kommer att skickas ut via mail den 12/3. Vid redovisning så förväntas du (och ev. din kollega, om du har arbetat med en annan student) kunna besvara vissa frågor kring implementationen och även visa upp samtliga aspekter av applikationen. Dvs. exempelvis påvisa att samtliga meny-alternativ och navigation mellan dessa fungerar.