CODEN:LUTEDX/(TEIE-3068)/1-39/(2016)

Industrial Electrical Engineering and Automation

Message Handler

Alaa El-Khalil

Ilaz Rexhepi

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Examensarbete

Lunds Tekniska Högskola

Högskoleingenjörsprogrammet - Datateknik

Message Handler

Utfört av

Alaa El-Khalil

Ilaz Rexhepi

Handledare

Mats Lilja, LTH Examinator

Christian Nyberg, LTH Handledare

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Inustriell Elektroteknik och Automation
Lunds universitet
Lund 2016

Sammanfattning

Nutid AB levererar idag ett kassasystem som de installerar på Sharps hårdvara, som i stort sett är en dator, pekskärm och kassaregister i ett. Denna specifikt framtagna hårdvara medför att kassasystemet blir väldigt dyrt för kunderna.

Företaget satsade på att få ut en applikation till marknaden i början av 2017. Vår del i arbetet bestod av att implementera en Message Handler som skulle sköta kommunikationen mellan kassasystemet och den trådlösa hårdvaran utanför. Kvittoskrivare, scanner, betalningsterminal skärmar och kassaregister är alla trådlösa hårdvaror som kan koppla upp och ansluta sig till kassasystemet via Wi-Fi eller Bluetooth.

Projektet slutade med att vi hade en tydlig, fungerande produkt som vi kunde visa upp. Den Message Handler som byggdes var av stor användning för företaget och de var extremt nöjda med resultatet. Precis som specificerat kunde Message Handler kunde ta emot ett JSON-objekt (ett meddelande), tolka det och sedan utföra uppgiften som beskrivs av JSON-objektet.

Uppgiften kunde exempelvis innebära en utskrift från en kvittoskrivare. För att utföra uppgiften trådlöst anslöt vi oss till den trådlösa enheten med hjälp av Wi-Fi (eller Bluetooth), omvandla meddelandet från mänskligt språk till datorspråk (binär vektor) och skicka meddelandet för att skrivaren slutligen skulle skriva ut kvittot.

Nyckelord: Kassasystem, apputveckling, trådlös kommunikation, hårdvara, JSON.

Abstract

Nutid AB currently supplies a POS (Point of Sale) system that they install on Sharps hardware, which consists of a computer, a touchscreen and a cash register all in one. This explicitly produced hardware results in the POS system being very expensive for the customers.

The company focused on getting an application out on the market at the beginning of 2017. Our part of the job consisted of implementing a Message Handler, that would handle alla communication between the POS system and the wireless hardware on the outside. Receipt printers, scanners, payment terminals, screens and cash registers are all wireless hardware that can connect to the POS system via Wi-Fi or Bluetooth.

We ended up delivering a clear, functional product that we could show. The Message Handler that was built was of great use to the company and they were extremely pleased with the results. Just as specified, the Message Handler could receive a JSON object (a message), interpret it and then perform the task described by the JSON object.

The task could be, for example, a printout from a receipt printer. To perform the task wirelessly, we joined the system to the wireless device using Wi-Fi (and Bluetooth), transforming the message from human language into computer language (binary array) and send the message to the printer which finally prints the receipt.

Keywords: POS System, App Development, Wireless Communication, Hardware, JSON.

Terminologi

BT	Bluetooth **(sv. Blåtand)** En standard som har tagits fram för trådlös kommunikation mellan olika enheter.
Display	Utmatningsenhet som används för att visuellt presentera information.
DLL	Dynamic-Link Library **(sv. Delade programbibliotek)** En fil som innehåller programfunktioner som kan delas med olika program. Exempel på innehåll är programkod, data eller resurser i form av bilder, ljud eller ikoner.
JSON	JavaScript Object Notation
NFC	Near-Field Communication **(sv. Närfältskommunikation)** Överföringsmetod för kontaktlöst utbyte av data över korta sträckor.
PC	Personal Computer **(sv. Persondator)**

PCL	Portable Class Library **(sv. Portabelt klassbibliotek)** Ett klassbibliotek vars största fokus ligger på kompatibilitet. Denna teknik gör det möjligt att skapa en applikation som fungerar på flera plattformar, istället för att skapa en applikation för varje plattform.
SDK	Software Development Kit **(sv. Utvecklingsverktyg)**
UWP	Universal Windows Platform **(sv. Universal Windows-plattform)**
UI	User Interface **(sv. Användargränssnitt)** Länk mellan användaren och den hårdvara eller programvara som användaren arbetar med. Exempelvis inmatning, där användaren har möjlighet att påverka systemet. Eller utmatning, där systemet visar resultatet av användarens påverkan.
Wi-Fi	Teknik för trådlösa nätverk.
XAML	Extensible Application Markup Language **(sv. Programmeringsspråk för användargränssnitt)**

Innehållsförteckning

1. INTRODUKTION

1.1 Bakgrund

Nutid AB har mer än 20 års erfarenhet inom utveckling av kassasystem för såväl olika sorters branscher inom detaljhandeln som för restauranger och caféer. Ett av deras system, Nutid PC Kassa, har tagits fram genom tätt samarbete med olika kundgrupper – allt för att skapa ett program som uppfyller de behov som finns i detaljhandeln idag. Med över 100 återförsäljare runt om i Sverige har Nutid AB gjort sig till ett känt namn inom kassasystem. ([1])

Under 2016 siktar Nutid på att lansera en mobil klient för att modernisera kassahanteringen. Detta innebär att de behöver skapa nya kommunikationsmetoder med externa enheter såsom skrivare, scanners, kontrollenheter, betalningsterminaler och kassalådor.

Samtidigt som företaget analyserade vilket operativsystem som skulle passa företagets målgrupp bäst stötte vi på Xamarin. Det är ett tillägg för Visual Studio som gör det möjligt att implementera applikationen en enda gång som fungerar på alla plattformar. Detta koncept är väldigt nytt och aningen ostabilt, men i stort sett så slipper man programmera samma applikation tre gånger för tre olika plattformar, vilket underlättar för företaget enormt rent tidsmässigt.

Xamarin fungerar helkompatibelt i princip upp till 80 % i jämförelse med om man hade programmerat i det inhemska språket. Det som fick företaget att trots detta satsa på Xamarin var att de resterande procenten kan man skriva med det inhemska språket för att uppnå fullständig funktionalitet.

Det fördes en diskussion med företaget där vi kom fram till att de främst siktar på att använda Wi-Fi, Bluetooth (för exempelvis kvittoskrivare), och NFC. Testningen för Bluetooth-utvecklingen kommer att ske via en etikettskrivare. Vår Message Handler ska alltså kunna ta emot ett meddelande och bland annat skriva ut det på en etikett.

Den mobila klienten ska bestå av en applikation (PC Kassa) som installeras på butikspersonalens surfplattor/smartphones. Personalen ska därefter kunna logga in i klienten och använda PC Kassan för att driva deras försäljning. Idag finns PC Kassa, fullt fungerande, för datorer.

1.2 Syfte och mål

Visionen med examensprojektet var att implementera kassasystemet på en surfplatta (iOS, Android och UWP). Denna idé var otroligt bra enligt Nutid, då de nyligen hade börjat tänka i de banorna och anställt en apputvecklare samtidigt som vi började. Om man genomför denna uppgradering skulle det medföra att Nutid skulle nå ut till många fler kunder, eftersom priset till kunderna skulle kunna sjunka markant. Detta är inte förvånansvärt, då det idag är betydligt mycket billigare för kunderna att införskaffa surfplattor än skräddarsydd hårdvara. Dessutom blir kassasystemet plattformsoberoende och för kunden innebär detta fler valmöjligheter.

Examensarbetet går ut på att skapa en Message Handler för att hantera kommunikationen för den mobila klienten. Kommunikationen ska kunna ske via Bluetooth, Wi-Fi, och (eventuellt) NFC. Message Handler ska kunna ta emot ett meddelande och skriva ut det på en annan.

Vårt mål är att leverera en fungerande Message Handler till företaget. Message Handler ska kunna ta emot ett meddelande och skriva ut det på en annan enhet via Bluetooth, Wi-Fi, och (eventuellt) NFC. Message Handler ska vara en del av den mobila klienten.

Exempel (förutsatt att klienten är installerad på en surfplatta):
Fruktsäljaren väljer att sälja artikeln “Äpple (Grönt) - Spanien” med pris 5 kr och artikelnummer 80801515 på klienten. Ett meddelande går ut till vår Message Handler som tar emot meddelande och skriver ut “Äpple - 5 kr” på displayen. Detta är effektivt eftersom man kan välja att filtrera den betydelsefulla informationen för kunden.

1.3 Problemformulering

Problemet kan delas upp i mindre delproblem och dessa överensstämmer i princip med de krav som ställdes upp i avsnitt 2.2. Beroende på vilken lösning som väljs för att hantera ett av kraven så kan det bli aktuellt att ändra lösningen till något av de övriga kraven. Med hänvisning till detta kan problemet delas in i tre olika delar:

1. Vilken plattform utvecklar vi till (iOS, Android, Windows)? Varför?
2. Vilket eller vilka programmeringsspråk använder vi oss utav? Varför?
3. Vilka verktyg använde vi? Varför?

I samband med att vi skulle påbörja projektet gjordes det en kort analys tillsammans med Nutid gällande vilket programmeringsspråk och vilka plattformar som var aktuella för dem. Då de flesta klienter använder Windows för persondatorerna och Android för surfplattorna blev det självklara beslutet att ha Android och UWP som målplattformar.

Därefter var val av programmeringsspråk det enda som återstod. Tanken var att vi skulle använda Xamarin för att slippa programmera först till Android och sedan till UWP. Xamarin tillåter oss att programmera en enda lösning som är generell och fungerar lika bra på Android som på UWP, men även på iOS. I vårt fall så var iOS inte med som målplattform, men man skulle kunna se det som en möjlighet för vidareutveckling i framtiden. Eftersom Xamarin enklast skrivs i C# blev detta vårt val av programmeringsspråk.

1.4 Avgränsningar

Eftersom källkoden skrivs i Xamarin med hjälp av en så kallad portable solution, så är portningen till ett annat operativsystem väldigt enkel. En stor fördel med portable solution är att man kan lägga till “native code”, det vill säga specialanpassningar för ett visst operativsystem, utöver Xamarin-koden vilket gör att man slipper utveckla samma applikation tre gånger för olika operativsystem. Xamarin är ganska nytt ute på marknaden, så en stor nackdel blir då att det inte har hunnit släppas några designerverktyg utan all kod måste skrivas manuellt. Dessutom återstår faktumet att det fortfarande finns många funktioner som Xamarin inte har stöd för i dagens läge.

Fokus låg främst på att utveckla stöd för Wi-Fi och Bluetooth för applikationen, då Message Handler är beroende av dessa tekniker. Dock, som ovan nämnts, fanns det fortfarande en del funktioner som Xamarin inte har stöd för, vilket innebar att vi behövde ta omvägar i utvecklingen för att kringgå denna brist och uppnå vårt mål. I mån av tid skulle vi även ha utvecklat mail- och sms/mms-tjänster för applikationen, men inlärningskurvan för Xamarin och C# var högre än vi hade räknat med, så det blev ingen tid över till det.

Message Handler har en enda uppgift i systemet – hantera meddelanden som kommer in och skicka vidare till en extern enhet. Den är alltså helt frikopplad ifrån databaser och design för applikationen.

2. METOD OCH ANALYS

2.1 Scrum

Scrum är en agil systemutvecklingsmetod som skapades av Jeff Sutherland och Ken Schwaber. Metoden har tillämpats sedan 90-talet och det är sätt att fördela arbetsuppgifter i projektet med fokus på att leverera värde till kunden. För att möjliggöra detta använder man sig av en så kallad PBI, Product Backlog Items, som kunden alltid har tillgång till. PBI är ungefär som en kravlista (på bland annat funktioner) där kunden hela tiden kan gå in och ta bort, prioritera och lägga till nya krav. För att PBI ska fungera måste kunden hela tiden se till så att den är korrekt prioriterad. Detta säkerställer då att scrum-gruppen utvecklar det som kunden prioriterar först – med andra ord – det som har störst värde för kunden.

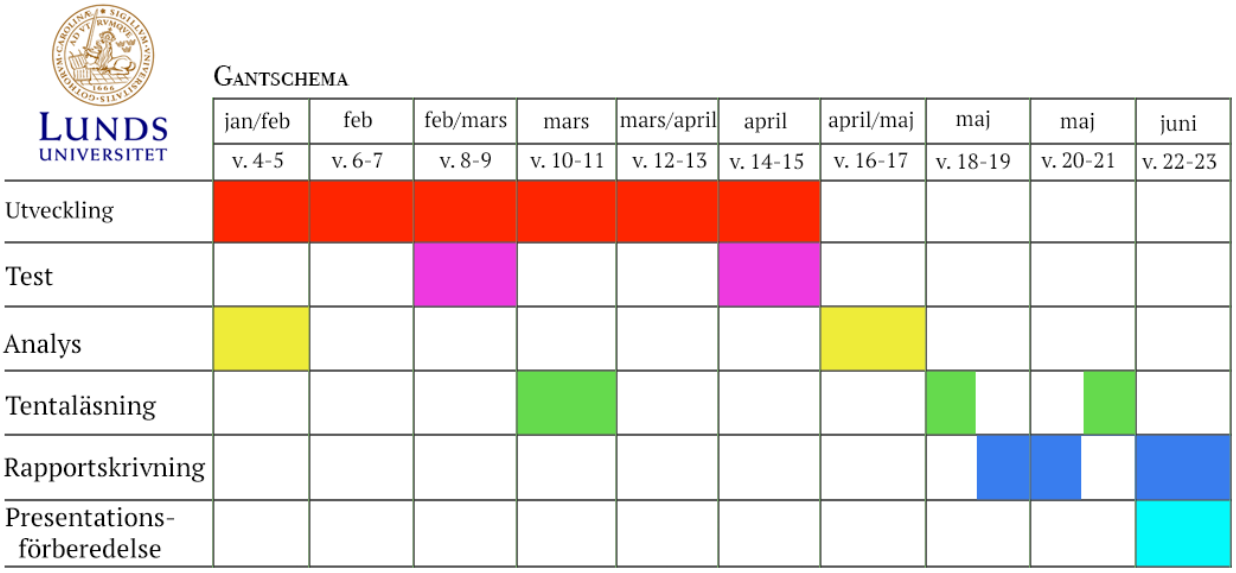
Scrum innehåller en handfull event som underlättar arbetet för utvecklargruppen. Bland dessa finns det ett event som kallas för Daily Scrum. Detta är ett möte på högst 15 minuter, där varje utvecklare kort besvarar tre frågor:

- Vad gjorde jag igår?
- Vad ska jag göra idag?
- Finns det några hinder?

Denna teknik säkerställer att alla utvecklare vet vad de andra i gruppen arbetar på, vilket gör att man känner ett gemensamt ansvar för att leverera rätt produkt i rätt tid. På Nutid hade Daily Scrums och eftersom vi nu var en del av utvecklingsteamet så fick vi naturligtvis delta. Viktigt att notera är att chefen inte deltar i detta möte, utan det är endast avsett för utvecklarna.

Ett annat Scrum-event som företaget utnyttjade var Sprint Planning. Det är ett detaljerat möte som i regel kan ta upp till 4 timmar. På mötet så redovisar chefen (produktägaren) vad som behöver göras och sammanställer tillsammans med utvecklarna en tidsplan för de nästkommande två veckorna. Sprint Planning ska i regel hållas i början av varannan vecka, men eftersom vår oföränderliga plan var färdig i början av projektet så behövde vi inte delta.

2.2 Tidplan



Figur 1 – Gantschema [14]

En av de mest fundamentala förberedelserna inför ett projekt är naturligtvis att se till så att man har all mjukvara installerad. På så vis kan man direkt dyka in i brainstorming och planering. Därför inleddes projektet just med att installera all mjukvara. Denna bestod av Visual Studio och Xamarin, samt ett antal andra tillägg som skulle tänkas behövas i specifika fall under genomförandet.

Då Xamarin var något helt nytt för oss, var nästa steg efter mjukvaruinstallationen att samla in information. Xamarins egen dokumentation kändes av naturliga skäl som den bästa källan och en bra punkt att börja på. Inläsningens syfte var då att lära oss om Portable Class Library (PCL), Shared Projects och vilka komponenter som Xamarin består av. Xamarins hemsida försåg oss även med tutorialvideor som vi med glädje tog åt oss. Tack vare den enorma databasen med samples (exempelkoder) som var tillgängliga blev kunskapen påtaglig väldigt fort, vilket satte oss i stadiet där man börjar prova skriva egna kodsnuttar och testköra triviala program.

Snabbt fick man en uppfattning om hur PCL fungerade, samt en insyn på hur man kan använda processordirektiv för att styra koden direkt mot olika plattformar i samma klass. Initialt kändes

metoden väldigt abstrakt, och det höll i sig en bra bit in i projektet innan man faktiskt förstod hur det hängde ihop.

Inlärningsprocessen var som förutspått stort lastad med ny kunskap och nya metoder. Det är faktiskt en sanning att projektet bestod till stor del av inläsning och inlärning. Utöver ovanstående var man tvungen att bekanta sig med exempelvis Blend, som är en komponent som underlättar uppbyggnaden av XAML-filer genom grafiskt gränssnitt, och NuGet, som visade sig vara en ovärderlig komponent längre fram i projektet.

NuGet handlar i stort sett om att importera färdiga bibliotek som man kan dra nytta av och använda i sitt projekt. Exempelvis så hade UWP inget bra sätt att hantera Sockets på, vilket gjorde att det blev svårt att skriva kod som kopplade upp en mot Wi-Fi nätverket. Detta löstes genom att man fick importera ett NuGet-package som simplifierade skrivandet av just denna typ av kod. Självklart finns det NuGets för andra ändamål också. Man kan se det som ett litet tillägg för projektet. Så fort man saknar något i ursprungskoden kan man kolla upp om det finns en NuGet som täpper igen hålet.

Med små steg förstod man snart hur allting hängde ihop. Det var nu möjligt att tänka ut en plan genom att brainstorma om exempelvis vilka klasser som skulle kunna tänkas behövas. Dessa skulle sedan struktureras och designas så att vi på ett enklare sätt kan börja skriva de senare. Den svåra utmaningen, och även det största kravet här var att se till så att klasserna är förberedda för nya funktioner i framtiden. Med andra ord så måste alla våra klasser vara dynamiska.

Sakta men säkert skapades en modell som man kunde utgå ifrån för att påbörja implementationen. Ett första utkast på en lösning var under konstruktion och funktionaliteten var i fokus. Dock var det minst lika viktigt för företaget att ta hänsyn till det andra perspektivet inom objektorienterad programmering – nämligen struktur.

Strukturen är lika viktig för projektet som för själva implementationen. En projektstruktur kan bestå av en tidsplan och en modell på arbetet som ska göras. Projektgruppen har alltså hittills

lyckats med en del av strukturen. Den andra delen tillhör som sagt implementationen, och där är det viktigt att man tänker på namnsättningen för variabler, metoder och klasser. Lika viktigt är det att man har en lättnavigerad layout samt att använda sig utav regions och kommentarer för att koden ska bli lättläst och lättförstådd.

2.3 Källkritik

Det enda sättet för oss att testa källornas trovärdighet var att faktiskt prova koden. Oftast är de bästa källorna Open Source, det vill säga att alla har tillgång till källkoden. Microsoft är ett exempel på en sådan källa. Med andra ord så kommer källorna från ursprungsskaparna och därför fanns det ingen mening med att vara källkritisk.

En annan aspekt är att det inte fanns många källor att välja bland. Eftersom företagen har sin egen ”originala” dokumentation väljer gemenskapen att följa den dokumentationen istället för att skriva en egen version av den. Xamarin är ett bra exempel. Deras dokumentation täcker precis allt man behöver veta för att kunna använda deras produkt. Återigen så innebär detta att källkritik inte var nödvändigt för detta projekt.

3. TEKNISK BAKGRUND

I detta avsnitt beskriver vi programmeringsspråken och de tekniker som vi har använt för projektet. Då dessa tekniker oftast har enorma dokumentationer har vi försökt att sammanfatta de viktiga delarna.

3.1 Visual Studio

Visual Studio är en komplett uppsättning av utvecklingsverktyg för att bygga ASP.NET webbapplikationer , XML Web Services, skrivbordsapplikationer och mobila applikationer. Visual Basic, Visual C # och Visual C ++ alla använder samma integrerade utvecklingsmiljö (IDE), som gör det möjligt för verktygsdelning och underlättar skapandet av flerspråkiga lösningar. Dessutom använder dessa språk funktionerna i .NET Framework, som ger tillgång till en nyckelteknik som förenklar utvecklingen av ASP webbapplikationer och XML Web Services. ([26])

3.2 C#

C# syntax är mycket uttrycksfull, men det är också enkelt och lätt att lära sig. Måsvingssyntaxen i C# kommer att vara omedelbart igenkännlig för alla som känner till C, C ++ eller Java. Utvecklare som känner till något av dessa språk är typiskt redo för att börja arbeta produktivt i C# inom en mycket kort tid. C# syntax förenklar mycket av komplexiteten i C++ och ger kraftfulla funktioner såsom nullable värdetyper, uppräkningar, delegater, lambda-uttryck och direkt minnesåtkomst, som inte finns i Java. C# stödjer generiska metoder och typer, som ger ökad typsäkerhet och prestanda, samt iteratorer, som gör det möjligt för de klasser som implementerar insamlingsklasserna att definiera egna iterationsbeteenden som är enkla att använda med klientkoden. Language Integrated Query (LINQ) gör att de hårdtypade databasfrågorna (Queries) blir en förstklassig språkkonstruktion. ([14])

Som ett objektorienterat språk stöder C# begreppen inkapsling, arv och polymorfism. Alla variabler och metoder, inklusive den huvudsakliga metoden, programmets startpunkt, är inkapslade i klassdefinitioner. En klass kan ärva direkt från endast en basklass, men kan implementera ett oändligt antal gränssnitt (interface). Metoder som överskuggar virtuella

metoder i en överordnad klass kräver nyckelordet override som ett sätt att undvika oavsiktlig omdefiniering. I C# är en struct som en lätt klass; det är en stackallokerad typ som kan implementera gränssnitt men stöder inte arv. ([14])

Utöver dessa grundläggande objektorienterade principer, gör C# det enkelt att utveckla mjukvarukomponenter genom flera innovativa språkkonstruktioner, bland annat följande:

- Inkapslade metodsignaturer (delegates), som gör det möjligt att typsäkra händelsemeddelanden.
- Egenskaper (properties), som fungerar som en mellanhand för privata medlemsvariabler.
- Attribut, som ger deklarativ metadata om olika typer vid körning.
- Inline XML-dokumentationskommentarer.
- Langugage Integrated Query (LINQ) som ger inbyggda databasfrågor på en mängd olika datakällor.

Om man måste interagera med andra Windows-program såsom COM-objekt eller infödda Win32 DLL, kan man göra detta i C# genom en process som kallas "Interop". Interop tillåter C#-program att göra nästan vad som helst som en inhemsk C++ applikation kan göra. C# stödjer även pekare och begreppet ”osäker kod” för de fall där direkt minnesåtkomst är helt avgörande.

Byggprocessen för C# är enkelt jämförbar med C och C++ och mer flexibel än i Java. Det finns inga separata huvudfiler, och inget krav på att metoder och typer deklareras i en viss ordning. En C# källkodsfil kan definiera ett godtyckligt antal klasser, structs, gränssnitt och händelser (events). ([14])

3.3 F#

F# är ett modernt funktionellt språk för .NET-plattformen som utvecklats av Microsofts forskningsteam. Det har varit under utveckling sedan 2005 och gjorde det till version 3.1 och med december 2013. Det började som ett akademiskt forskningsprojekt och har under flera år mognat till ett produktionsfärdigt språk som används av många kommersiella företag, särskilt inom finanssektorn.

Tomas Petricek och Phillip Trelford startade F# Software Foundation (FSSP) i 2012 som en informell och gemenskapsdriven organisation med målet att främja och tillhandahålla en gemenskapsröst för F# programmeringsspråk. De skapade fsharp.org som en webbplats, som gemenskapen underhåller, för resurser kring programmeringsspråket F#, inklusive allmän information och utbildningsresurser. ([15])

Organisationen växte snabbt och blev ansvarig för en stor del av den öppna källkoden F# kodarkiv och utbildningsresurser, inklusive den öppna upplagan av F# kompilatorn för plattformsoberoende användning och språkspecifikationen för F#. Tekniska arbetsgrupper från viktiga gemenskapsmedlemmar har också bildats för att hjälpa till att stödja och främja F# användning inom den akademiska världen och olika branscher samt för att konsolidera resurserna. ([15])

År 2014 beslutade arrangörerna av F# Software Foundation, tillsammans med de ursprungliga grundarna, att tillväxten av F# Software Foundation har motiverat fram en formell organisation, och att en legal struktur skulle gynna de övergripande målen samt den informella organisationens uppdrag. I slutet av 2014 bildades F# Software Foundation som ett ideellt företag i delstaten Nevada i USA, och resurser som är förknippade med den ursprungliga informationsorganisationen flyttades inom ramen för den juridiska enheten. ([15])

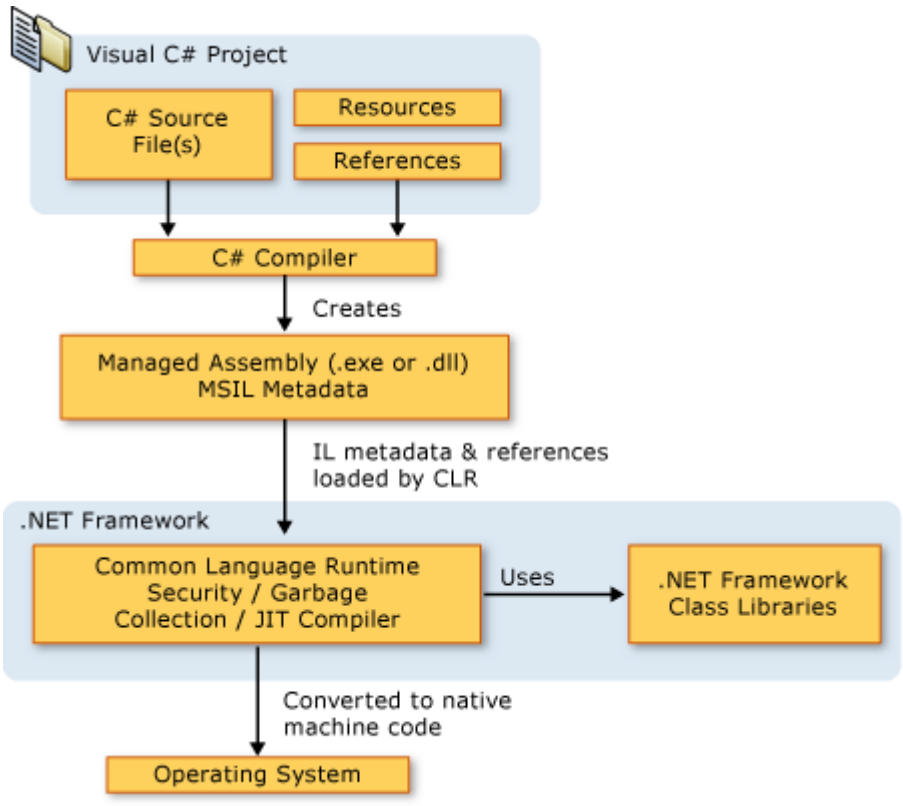
3.4 .NET Framework Plattformsarkitektur

C#-program körs på .NET Framework (sv. Ramverk), en integrerad del av Windows som innehåller ett virtuellt exekveringssystem som kallas Common Language Runtime (CLR) och en enhetlig uppsättning klassbibliotek. CLR är den kommersiella implementationen från Microsoft av Common Language Infrastructure (CLI), en internationell standard som är grunden för att skapa exekverings- och utvecklingsmiljöer där programmeringsspråk och bibliotek samarbetar sömlöst. ([2], [14])

Källkod som är skriven i C# kompileras till ett mellanliggande språk (IL) som överensstämmer med CLI-specifikationen. IL-kod och resurser, såsom bitmappar och strängar, lagras på disk i en körbar fil som kallas för assembly, typiskt med en förlängning av .exe eller .dll. En

sammanställning innehåller en manifest som ger information om typ, version, kultur och säkerhetskrav för assemblyfilen. ([2], [14])

När C#-programmet exekveras laddas assemblyfilen upp i CLR, som kan ta olika åtgärder baserat på informationen i manifestet. Om säkerhetskraven uppfylls utför CLR en så kallad Just In Time (JIT) kompilering för att omvandla IL koden till inhemska maskininstruktioner. CLR ger också andra tjänster med anknytning till automatisk sophämtning (Garbage Collection), undantagshantering (Exception Handling) och resurshantering (Resource Management). Kod som exekveras av CLR kallas "förvaltad kod" i motsats till "oförvaltad kod" som sammanställs till ursprunglig maskinkod som riktar ett specifikt system. Följande diagram visar kompilering- och Runtime-relationer C# källkodsfiler, .NET Framework klassbibliotek, assembly och CLR. ([2], [14])



Figur 2 - Compiletime/Runtime-relationer för C# källkodsfiler [14]

Språkinteroperabilitet är en viktig del av .NET Framework. Eftersom IL kod som produceras av C#- kompilatorn överensstämmer med Common Type Specification (CTS), kan IL kod som genereras från C# samverka med kod som genererades från .NET versioner av Visual Basic, Visual C++, eller någon av mer av de 20 andra CTS-kompatibla språken. En enda enhet kan innehålla flera moduler skrivna i olika .NET-språk, och typerna kan referera till varandra precis som om de var skrivna i samma språk. ([2], [14])

Utöver Runtime-tjänsterna innehåller .NET Framework också ett omfattande bibliotek med över 4000 klasser organiserade i Namespaces (namnområden) som erbjuder ett brett utbud av användbara funktioner för allt från filöverskrivning och filinläsning, till strängmanipulationer, till XML-tolkning, till Windows Forms kontroller. Det typiska C#-programmet använder .NET Framework klassbibliotek i stor utsträckning bara för att hantera vanliga hantverkarsysslor. ([2], [14])

3.5 JSON

JSON (JavaScript Object Notation) är ett lätt datautbytesformat. Det är lätt för människor att läsa och skriva samt för maskiner att tolka och generera. Den är baserad på en delmängd av programmeringsspråket JavaScript. JSON är ett textformat som är helt språkoberoende men använder konventioner som är bekanta för programmerare i C-språkfamiljen. Detta inkluderar C, C++, C#, Java, JavaScript, Perl, Python, och många andra. Dessa egenskaper gör JSON till det ideala datautbytesspråket. ([23], [24], [25])

JSON bygger på två strukturer:

- *En samling av namn- och värde-par.*

  På olika språk realiseras detta som ett objekt, struct, ordbok, hashtabell, nyckellista, eller associativ array.

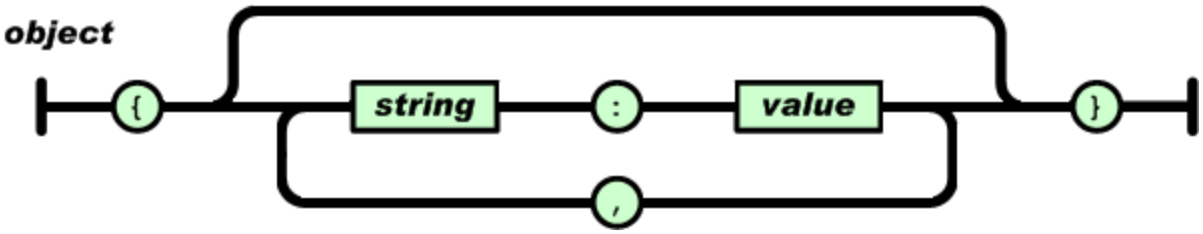
- *En ordnad lista av värden.*

  I de flesta språk, är detta realiseras som en matris, vektor, lista, eller sekvens.

Dessa är universella datastrukturer. Så gott som alla moderna programmeringsspråk stöder dem i en eller annan form. Det är logiskt att ett dataformat som är ersättningsbart med programspråk också baseras på dessa strukturer.

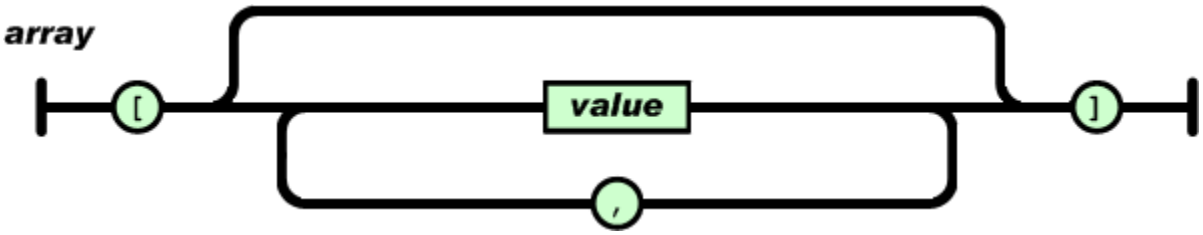
I JSON kan det se ut på följande sätt:

Ett objekt är en oordnad uppsättning namn- och värde-par. Ett objekt börjar med { (vänster måsvinge) och slutar med } (höger måsvinge). Varje namn följs av : (kolon) och namn- och värde-paren är separerade med , (kommatecken).



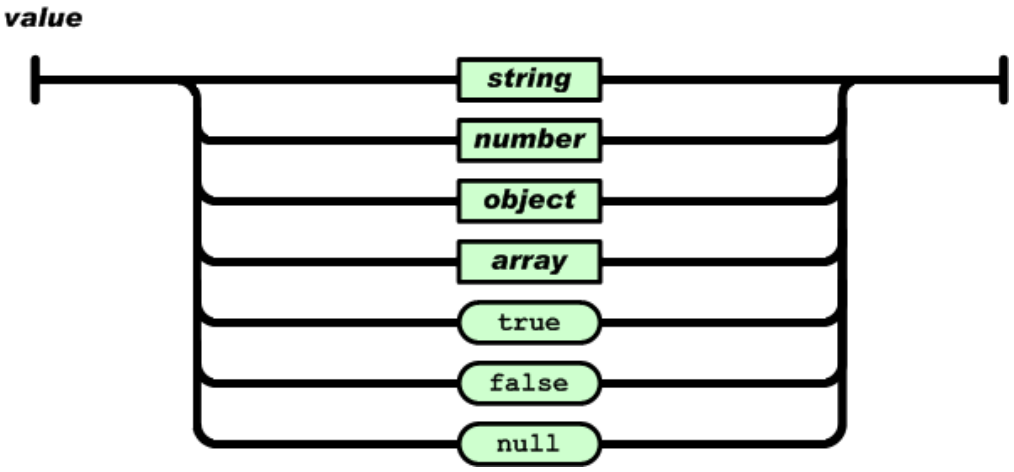
Figur 3 - JSON-objektens konstruktion [23]

En array är en ordnad samling av värden. En array börjar med **[** (vänster hakparantes) och slutar med **]** (höger hakparantes). Värdena är separerade med **,** (kommatecken).



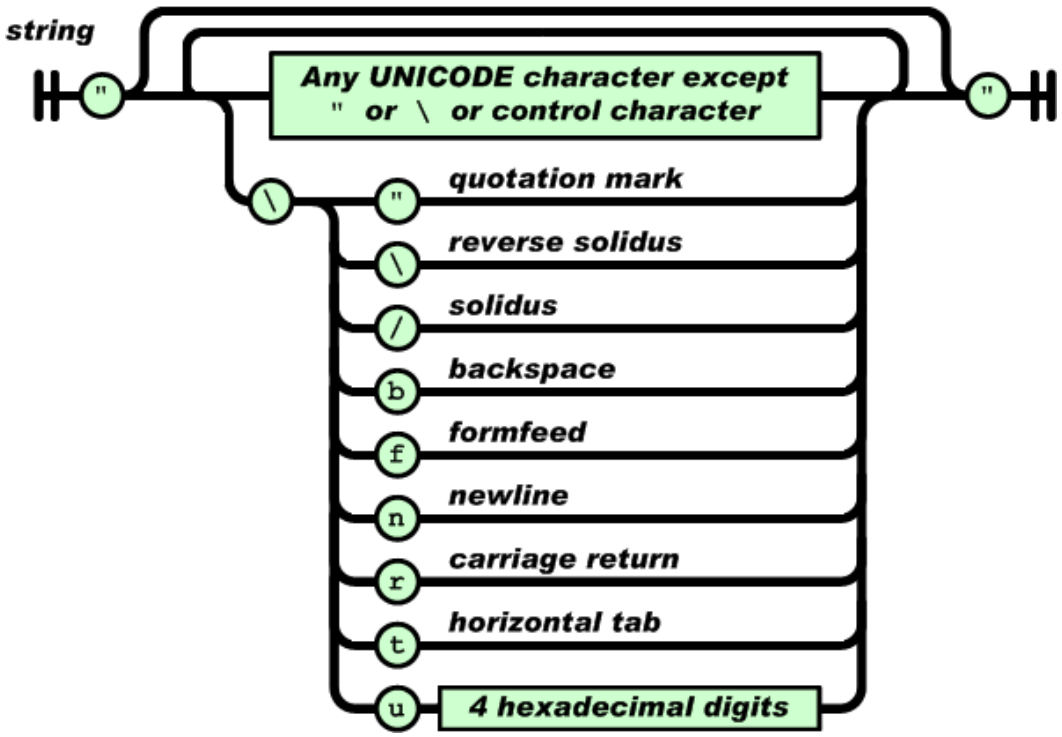
Figur 4 - Arrayens konstruktion [23]

Ett värde kan vara en sträng inom citattecken, ett nummer, ett booleskt uttryck (sant eller falskt), null, ett objekt eller en array. Dessa strukturer kan kapslas.



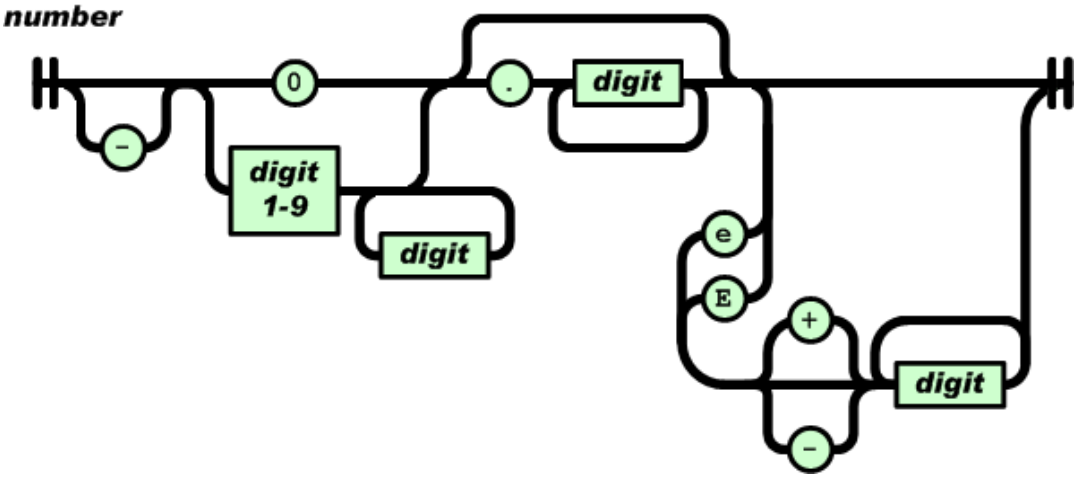
Figur 5 – Ett värdes möjliga beståndsdelar [23]

En sträng är en sekvens av noll eller flera Unicode-tecken insvepta i citattecken med hjälp av escape-tecken. Ett tecken representeras som en enda teckensträng. Strängen är väldigt lik tängarna i C eller Java.



Figur 6 - Strängens konstruktion [23]

Ett nummer är mycket likt ett C- eller Java-nummer, förutom att oktala och hexadecimala format inte används.



Figur 7 – Nummerkonstruktionen [23]

3.6 NuGet

NuGet är pakethanteraren för Microsofts utvecklingsplattform Visual Studio inklusive .NET. Klientverktygen som NuGet erbjuder ger möjlighet att producera och konsumera paket. NuGet Gallery är den centrala paketsamlingen som används av alla paketförfattare och konsumenter.

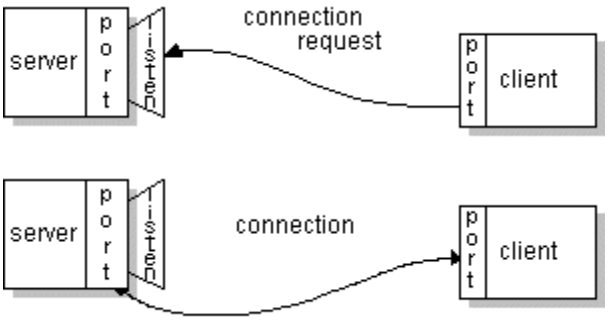
NuGet är ett verktyg låter dig ladda ner och inkludera källfiler (cs, vb, js, etc.) eller delade programbibliotek (DLL) i Visual Studio-projekt. Idag finns det en hel del kodbibliotek på webben; öppen källkod är inte nytt. Det svåraste är att hitta den koden, ladda ner, referera den från projektet och hantera framtida uppdateringar av det biblioteket när de blir tillgängliga. Detta måste upprepas för varje beroende som finns i programbiblioteket. NuGet förenklar alla dessa steg till bara ett par klick.

Även om det finns en hel del öppen källkod från tredjepartsbibliotek tillgängliga bör det noteras att även Microsoft är starkt beroende av NuGet också. Om Microsoft skulle ladda upp varje nytt bibliotek de skapar till det officiella .NET Framework skulle ramverket vara flera terabyte stort. Istället är det mycket bättre att tillgängliggöra alla bibliotek på nätet och låta programmeraren

ladda ner bara precis det hen behöver och distribuera det i sitt egna projekt för att hålla det borta från .NET Framework. ([27], [28])

3.7 Sockets

En socket är en ändpunkt i en länk mellan två program som körs över nätverket. Socketklasser används för att representera en förbindelse mellan ett klientprogram och ett serverprogram. Java.net biblioteket förser oss med två klasser, Socket och ServerSocket, som implementerar klientsidan av förbindelsen respektive serversidan av förbindelsen. ([5])



Figur 8 – Socketuppkopplingen [7]

En socket är en de mest fundamentala teknologierna inom datorkommunikation. Sockets tillåter att applikationer kommunicerar med varandra genom att använda standardmekanismerna som är inbyggda i hårdvara och i operativsystemen. Mjukvaruapplikation som förlitar sig på internet och andra datorkommunikationssätt växer ständigt i popularitet. Många av dagens mest populära mjukvarupaket förlitar sig helt på sockets. Detta inkluderar webbläsare, chattapplikationer och ”Peer to Peer” fildelningssystem. ([6], [7], [8])

I stort sett representerar en socket en enda uppkoppling mellan exakt två mjukvaror. Vill man kommunicera med fler mjukvaror samtidigt så går det bra, förutsatt att det sker via en klient/serversystem, men då krävs det flera sockets för att åstadkomma detta. Socketbaserade program körs vanligtvis på två separata datorer i nätverket, men sockets kan i detta fall användas för att kommunicera lokalt på en och samma dator. ([6], [7], [8])

Sockets är dubbelriktade, vilket innebär att båda sidorna om förbindelsen är kapabla att skicka samt ta emot data. Ibland så kallas det program som initierar kommunikationen för klient och det andra programmet för server, trots att båda programmen kan uppfylla varandras roller. Dock leder detta terminologiska problem till en onödig förvirring inom icke-klient/serverbaserade system och bör därför undvikas. ([6], [7], [8])

Socketgränssnitt kan delas in i tre kategorier. Stream socket, som troligtvis är den mest frekvent använda, implementerar en förbindelseorienterad semantik. I huvudsak så kräver en stream att de två kommunicerande partierna fastställer en socketförbindelse. Därefter så garanteras det att all data som skickas via den förbindelsen kommer fram i exakt samma ordning som den skickades. ([6], [7], [8])

Datagram socket tillför en förbindelselös semantik. Med datagram är förbindelserna implicita istället för explicita som de är i stream. Något av partierna skickar ett datagram när de behöver och väntar helt enkelt på att det andra partiet ska svara. Meddelanden bär risken att försvinna under transporteringen eller att de kommer fram i felaktig ordning. Dock är detta applikationens och inte datagrams uppgift att sköta denna typ av problem. Att implementera datagram sockets kan ge vissa applikationer en prestandaökning och utökad flexibilitet i jämförelse med stream sockets, vilket rättfärdigar deras användning i vissa situationer. ([6], [7], [8])

Den tredje och sista typen av sockets, den så kallade Raw socket, kringgår bibliotekets inbyggda support för standardprotokoll som TCP och UDP. Raw sockets används för skräddarsydda lågnivå protokollutveckling. ([6], [7], [8])

4. GENOMFÖRANDE

Det här avsnittet beskriver tillvägagångssättet för projektet. Här besvaras frågor som *Hur gjorde ni för att uppnå ert resultat?* Det förklaras i detalj hur vi gjorde för att slutligen lyckas skapa vår Message Handler.

4.1 Bluetooth

Det man var tvungen att göra här var att sätta på Bluetooth på både enheterna, och koppla ihop dem med varandra. Sedan så söker man via vår kod i enheten som användaren är kopplad med och leta upp rätt enhet för att skicka över bytes (meddelandet som ska skickas över till den uppkopplade enheten).

4.2 Wi-Fi

Vi gjorde ett program som pingar olika IP-adresser i det lokala nätverket för att få reda på vilka enheter som är uppkopplade. Därefter kunde man ansluta sig till enheten, exempelvis skrivaren, och skicka den information som ska skrivas ut. Denna teknik kan appliceras på alla enheter som är uppkopplade mot det trådlösa nätverket. Eftersom varje enhet får en unik IP-adress finns det ingen risk för kollision vid dataöverföring, då programmet skickar till den specifika IP-adressen som den identifierat som mottagare.

4.3 Second Screen

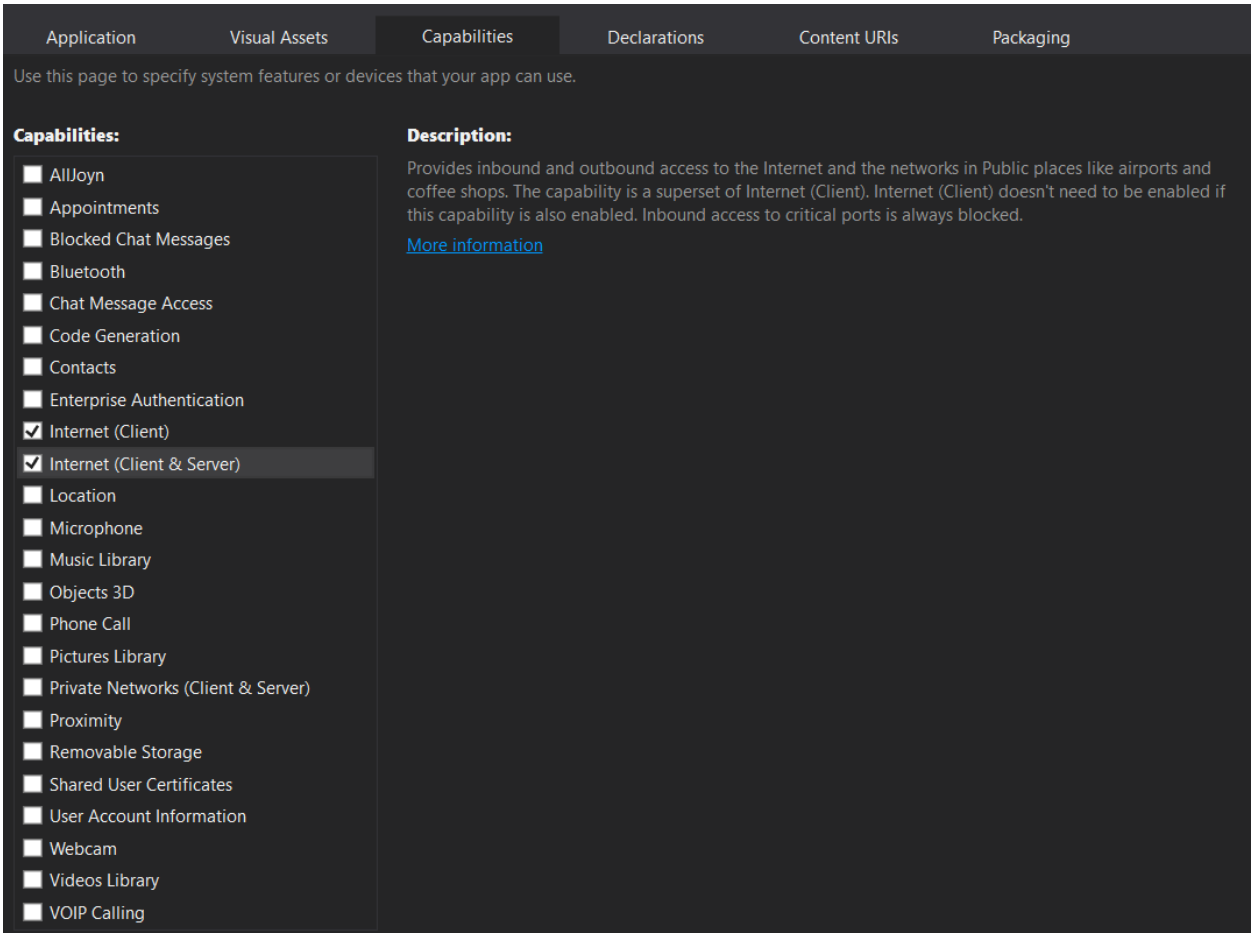
Här använde vi oss utav socket för att skapa en klient och en server som skickar meddelande emellan varandra. För att kunna använda sockets i .Net-applikationer var man tvungen att lägga till using statements.

```
using System.Net;
using System.Net.Sockets;
```

Man kan därefter skapa ett socketlistener objekt.

```
Socket sListener;
```

Sedan skapar vi ett klickevent som aktiverar vår socket och sätter ett värde på dess IpEndPoint och protokolltyp. Men innan det behöver socketen tillåtelse att arbete, eftersom den kommer att använda ett stängt portnummer. En ruta kommer att dyka upp och be om tillåtelse för att skicka data.



Figur 9 - Appen behöver tillåtelse för internettjänsterna [Skärmdump]

Sockets använder protokoll för att överföra data. Det mest kända protokollet i vårt sammanhang är UDP, som är väldigt snabbt men inte pålitligt, och TCP, som är väldigt pålitligt men inte särskilt snabbt. Dock anser vi att pålitlighet är viktigare än hastighet när man skickar meddelande, vilket är anledningen till att vi huvudsakligen använder oss av TCP.

```
sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
```

Socketen behöver ha en adress. Denna adress är av typen IpEndPoint. Varje socket kan identifieras via denna IP-adress, vilket är väldigt användbart för att lokalisera värdens dator, och portnumret för att identifiera vilket program som använder socketen i datorn.

```
IPHostEntry ipHost = Dns.GetHostEntry("");
IPAddress ipAddr = ipHost.AddressList[0];
ipEndPoint = new IPEndPoint(ipAddr, 4510);
```

Nu associerar vi vår socket med IpEndPoint:

```
sListener.Bind(ipEndPoint);
```

För att se till att vår socket blir klar för användning så kan vi börja lyssna på ett förvalt portnummer (4510). Man kan välja vilket portnummer man vill så länge det inte redan är upptaget, dock måste klienten vara medveten om detta.

Placera en socket i lyssningstillstånd och specificera hur många klienter som kan koppla upp sig mot den:

```
sListener.Listen(10);
```

Server kommer att genomföra en asynkron operation för att acceptera ett försök. En av de mest kraftfulla funktionerna hos en socket är användningen av den asynkrona programmeringsmodellen. Tack vare den kan vårt program fortsätta köra medan socketen genomför sina uppgifter.

```
AsyncCallback aCallback = new AsyncCallback(AcceptCallback);
sListener.BeginAccept(aCallback, sListener);
```

Om det finns ett försök från en klient som försöker koppla upp sig, kommer följande kod att exekveras:

```
Socket listener = (Socket)ar.AsyncState;
Socket handler  = listener.EndAccept(ar);
```

För att asynkront börja ta emot data behöver vi en vektor av typen Byte för den mottagna datan, den nollbaserade positionen i buffern samt antalet bytes som ska tas emot.

```
handler.BeginReceive(buffer, 0, buffer.Length,
          SocketFlags.None, new AsyncCallback(ReceiveCallback), obj);
```

Om klienten skickar ett meddelande kommer servern att försöka hämta det. Eftersom sockets skicka data i form av binär typ är det viktigt att man omvandlar (konverterar) de till strängar. Det är värt att känna till att servern, men även klienten, inte alls har någon kännedom om hur långt meddelandet är eller hur lång tid det tar för servern att ta emot hela meddelandet. Därför anropar vi en speciell sträng ″<Client Quit>″ och stoppar in det i slutet av meddelandet för att signalera att meddelandet slutar där. För att ta emot data anropar vi BeginReceive:

```
byte[] buffernew = new byte[1024];
obj[0] = buffernew;
obj[1] = handler;
handler.BeginReceive(buffernew, 0, buffernew.Length,
        SocketFlags.None, new AsyncCallback(ReceiveCallback), obj);
```

Efter att klientens meddelanden har mottagits, vill man kanske att servern ska svara på något sätt. Dock måste detta svar först konverteras från sträng till bytes data, eftersom sockets endast kan manipulera bytes.

```
byte[] byteData = Encoding.Unicode.GetBytes(str);
```

Servern kommer nu att skicka data asynkront till den uppkopplade socketen:

```
handler.BeginSend(byteData, 0, byteData.Length, 0, new AsyncCallback(SendCallback), handler);
```

När man använder sig av den asynkrona programmeringsmodellen för att bygga detta exempel, löper man ingen risk av att applikationen eller gränssnittet blockeras. Uppgifterna kommer att utföras av olika trådar i processorn.

Efter att ha skrivit koden kanske man vill hantera de olika metoderna genom ett användargränssnitt, för att på ett enklare sätt kunna testa och visualisera olika ändringar eller tillägg i koden.

Klienten kommer att försöka koppla upp sig mot servern, men den behöver första känna till IP-adressen. Efter att man har skapat en SocketPermission för att förbifartleda tillgångsrestriktioner och skapat en socket med en matchande IpEndPoint (IP-adress), kan vi upprätthålla en uppkoppling mot servern:

```
senderSock.Connect(ipEndPoint);
```

Man måste notera att det skapade IpEndPoint inte kommer att användas för att identifiera klienten, utan kommer istället att användas för att identifiera serverns socket.

För att skicka meddelanden, lägger klienten på ″<Client Quit>″ för att markera slutet på meddelandet och måste naturligtvis omvandla textmeddelandet till binärt format, precis som servern gjorde. Därefter kommer socketen att skicka meddelandet genom att åberopa metoden Send, som tar in det binära meddelandet som inparameter.

```
byte[] msg = Encoding.Unicode.GetBytes(theMessageToSend + "<Client Quit>");
int bytesSend = senderSock.Send(msg);
```

För att ta emot data från servern, konverterar den bytevektorn till en sträng och fortsätter därefter med att läsa data ur strängen tills det inte finns någon mer data tillgänglig:

```
String theMessageToReceive = Encoding.Unicode.GetString(bytes, 0, bytesRec);
while (senderSock.Available > 0)
{
  bytesRec = senderSock.Receive(bytes);
  theMessageToReceive += Encoding.Unicode.GetString(bytes, 0, bytesRec);
}
```

För att stänga ner förbindelsen bör vi använda oss av Socket.Shutdown (SocketShutdown.Both) och Socket.Close().

5. RESULTAT

Trots att vi inte kunde C# när vi började med projektet så insåg vi snabbt hur mycket flexiblare det är än det vi kunde, nämligen Java. En stor bonus för C# är att man kan använda Visual Studio som utvecklingsmiljö. Visual Studio anser vi är det absolut bästa verktyget för en utvecklare och är betydligt mycket bättre än Eclipse eller Android Studio, som är Javas två populäraste utvecklingsmiljöer.

Visual Studio har exempelvis stöd för Source Control som går direkt mot GitHub, som är ett vedertaget versionshanteringssystem. Det som händer då är att om man arbetar i ett team, vilket man i de allra flesta fallen gör i ett projekt, så kan medlemmarna se ändringarna som man har gjort i koden och applicera de på sin egen kod. Detta är särskilt bra om man arbetar på samma projekt i Visual Studio. Om vi tar ett datorspel som exempel, så kan en medlem arbeta på spellogiken medan den andra arbetar på spelgrafiken, och båda får varandras kod utan att behöver skicka någonting mellan sig.

Det är dessutom mycket mindre kod att skriva i C#. Till exempel så har man två separata metoder för att hämta eller ändra ett värde i en klass. Om vi tänker oss att vi har en klass *Point* med attributet Coordinates så måste man alltså ha en metod setCoordinates och en metod getCoordinates. Dessa metoder har endast en uppgift och det är att uppdatera respektive returnera attributets värde, och metoderna är oftast endast en rad kod. I C# har man istället Properties, någonting som är mittemellan attribut och metoder. *Property* är alltså ett attribut som har två inbyggda metoder, nämligen *get* och *set.* Så det räcker att man anropar Point.Coordinates om man vill få ut värdet (get) och Point.Coordinates = nytt värde för att uppdatera värdet (set).

Projektet slutade med att vi hade en tydlig, fungerande produkt som vi kunde visa upp. Den Message Handler som byggdes var av stor användning för företaget och de var extremt nöjda med resultatet. Precis som specificerat kunde Message Handler kunde ta emot ett JSON-objekt (ett meddelande), tolka det och sedan utföra uppgiften som beskrivs av JSON-objektet.

Uppgiften kunde exempelvis innebära en utskrift från en kvittoskrivare. För att utföra uppgiften trådlöst anslöt vi oss till den trådlösa enheten med hjälp av Wi-Fi (eller Bluetooth), omvandla meddelandet från mänskligt språk till datorspråk (binär vektor) och skicka meddelandet för att skrivaren slutligen skulle skriva ut kvittot.

JSON-objektet innehöll en enhetstyp och en uppgift. Exempelvis kunde JSON-objektet se ut på det här sättet:

```
{ "DeviceType" : "Printer" , "Action" : "Print" , "Receipt" : [
        { "product" : "äpple" , "unitPrice" : "3" , "quantity" : "3" , "totalPrice" : "9" }
        { "product" : "banan" , "unitPrice" : "5" , "quantity" : "3" , "totalPrice" : "15" }
        { "product" : "mango" , "unitPrice" : "20" , "quantity" : "3" , "totalPrice" : "60" }
]}
```

Detta objekt tolkas av Message Handler som:

- Leta upp en skrivare, om det finns flera – lista dem och låt användaren välja
- Anropa metoden Print för den skrivaren

Metoden Print hade hand om att koppla upp sig mot skrivaren som användaren har valt och skriva ut ett finformaterat kvitto med alla produkter som fanns med i Receipt.

Kortterminalen och CCU hade vi inte tillgång till för att testa på, men med den dynamiska kod som har skrivits är det endast små tillägg som företaget behöver göra på vår kod för att det ska fungera felfritt. Företaget skaffade fram en kortterminal som vi försökte med, men testkorten fungerade inte och vi uteslöt därmed den biten.

6. SLUTSATS

I stort sett gick projektet väldigt smidigt. Vi var delaktiga i företagets Daily Standups (Dagliga Scrum-möten) trots att den större delen av mötena var irrelevanta för vår del. Dock var det viktigt för företagets andra programmerare att få ett hum om hur det går med apputvecklingen, vilket gjorde att vårt projekt ofta kom upp på tal.

På tal om Scrum så lade vi under projektets gång märke till att verkligheten skiljer sig en del från teorin. Ett viktigt event i Scrum är Sprint Retrospective, där man i slutet av varje Sprint (varannan vecka) samlar alla som är delaktiga i projektet för att prata om hur man kan förbättras som ett lag. Man tar då upp hinder som har dykt upp och andra problem som har uppstått. Det hölls tyvärr inga Retrospectives på Nutid. Företaget påstod att de använde Scrum som projektmodell, men följde inte reglerna fullständigt, utan det kändes som att de plockade godbitarna som passande de bäst och använde det istället.

Vid flera tillfällen kunde det kännas som att kodandet har tagit stopp, och ibland kunde vi sitta fast med samma problem i flera dagar. Då företaget hade en strikt och snar deadline att hålla sig till blev det inte lika mycket handledning från företagets håll som vi hade hoppats på. Men det bidrog i sin tur till att man lärde sig hur man tar sig an svåra problem på egen hand, och man fick se många olika exempellösningar på nätet, vilket breddade vår erfarenhet markant.

1. *Vilken plattform utvecklar vi till (iOS, Android, Windows)? Varför?*

Tillsammans med företaget valde vi att utveckla en Message Handler som fungerade på alla plattformar, men fokus låg ändå på Android och Windows, eftersom majoriteten av företagets kunder använder sig av dessa plattformar. Detta innebar för oss att vi behövde utveckla i Xamarin, för att få en lösning som fungerar upp till cirka 80 % på Android, Windows och iOS. Därefter implementerade vi de resterande procenten i Android respektive Windows Native Code.

2. *Vilket eller vilka programmeringsspråk använder vi oss utav? Varför?*

All kod som vi skrev för projektet var i C#. Vi hade endast C# och F# att välja mellan, då dessa var de enda språken som stöds av Xamarin. Anledningen till att vi valde C# var dels för att vi hade lärt oss Java på universitetet, vilket gav oss en stabil grund att stå på vid inlärningen av C#, men även för att Message Handler indirekt krävde att vi arbetade objektorienterat, vilket vi var vana vid från Java. Eftersom F# är ett funktionellt språk hade vi inte bara behövt lära oss ett nytt programmeringsspråk, utan vi hade behövt sätta oss in i ett helt nytt sätt att programmera på. Vi tror att det är tillräckligt bra argument för att utesluta F# från att vara vårt val av programmeringsspråk.

3. *Vilka verktyg använde vi? Varför?*

De verktyg vi huvudsakligen använde oss utav var Visual Studio (naturligtvis), NuGet för pakethantering, Xamarin för plattformsoberoende lösningar, .NET-ramverket för C#-stöd, Sockets för uppkoppling och slutligen JSON för tolkning och skrivning av meddelanden.

Var och en av dessa verktyg har haft effekten av att bespara oss jättemycket tid för projektet. Visual Studio erbjuder en kraftfull kompilator och debugger som underlättar och försnabbar felsökningen för koden. NuGet tillåter oss att med ett par knapptryck ladda hem programbibliotek i stora mängder, som vi i vanliga fall hade behövt ladda hem manuellt, ett programbibliotek i taget. Med Xamarin slipper vi skriva ett helt program för varje plattform. .Net-ramverket ser till så att vi har allt stöd vi behöver för att kunna skriva produktiv C#-kod. Socket är ju faktiskt en del av .Net-ramverket och abstraherar bort komplexiteten för att skapa en uppkoppling mellan två enheter samtidigt som den sköter trådsäkerheten, som vi annars hade fått sköta manuellt i vår kod. Trådsäkerheten är utan tvekan det svåraste inom systemutveckling enligt majoriteten av gemenskapen.

Och slutligen, JSON. Denna teknik ingår inte i .Net-ramverket utan skapades ursprungligen för att i webbutveckling kunna simulera stateful, och fungerar bäst med JavaScript (JSON står ju för JavaScript Object Notation). Dock växte denna teknik in i C# väldigt snabbt och gjorde så att

man kunde serialisera ett komplext objekt, det vill säga en klass i C#, till JSON med en enda rad kod. Man kunde med hjälp av det få JSON-representationen för innehållet av ett komplext objekt. Om JSON inte hade funnits så hade vi istället fått använda XML och arbeta med hierarkiska datastrukturer, som är större rent kodmässigt och kanske lite otydligare.

Möjligheter och Vidareutveckling

En möjlighet är att vidareutveckla Message Handler genom att inte endast koppla upp sig via Wi-Fi och Bluetooth, utan även NFC. En vision för Message Handler hade kunnat vara att den blir så pass modulär och generisk att man kan sälja den som en hyllvara för andra system som är i behov av en smidig dataförmedlingsteknik.

7. REFERENSER

Nutid

[1] *Official Company Site*
www.nutid.se *[Access: 26 juli 2016]*

Network

[2] *Network Programming in the .NET Framework*
https://msdn.microsoft.com/en-us/library/4as0wz7t(v=vs.110).aspx *[Access: 26 juli 2016]*

[3] *Networkinformation.Ping*
https://msdn.microsoft.com/en-us/library/system.net.networkinformation.ping(v=vs.110).aspx
[Access: 26 juli 2016]

[4] *Mail.SmtpClient*
https://msdn.microsoft.com/en-us/library/system.net.mail.smtpclient(v=vs.110).aspx *[Access: 26 juli 2016]*

Sockets

[5] *Network.Socket*
https://hackage.haskell.org/package/network-2.6.2.1/docs/Network-Socket.html *[Access: 26 juli 2016]*

[6] *All About Sockets*
http://journals.ecs.soton.ac.uk/java/tutorial/networking/sockets/index.html *[Access: 26 juli 2016]*

[7] *What is a socket?*
https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html *[Access: 26 juli 2016]*

[8] *Socket Class*
https://msdn.microsoft.com/en-us/library/system.net.sockets.socket(v=vs.110).aspx *[Access: 26 juli 2016]*

[9] *Using TCP Services*
https://msdn.microsoft.com/en-us/library/k8azesy5(v=vs.110).aspx *[Access: 26 juli 2016]*

[10] *Using UDP Services*
https://msdn.microsoft.com/en-us/library/tst0kwb1(v=vs.110).aspx *[Access: 26 juli 2016]*

[11] *Socket.BeginSend*
https://msdn.microsoft.com/en-us/library/7h44aee9(v=vs.110).aspx *[Access: 26 juli 2016]*

[12] *Socket.Send*
https://msdn.microsoft.com/en-us/library/system.net.sockets.socket.send(v=vs.110).aspx
[Access: 26 juli 2016]

[13] *Socket.Receive*
https://msdn.microsoft.com/en-us/library/system.net.sockets.socket.receive(v=vs.110).aspx
[Access: 26 juli 2016]

[14] *C# och .NET Framework*
https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx *[Access: 26 juli 2016]*

[15] *F#*
http://foundation.fsharp.org/history *[Access: 26 juli 2016]*

Xamarin

[16] *Sharing Code Options*
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/ *[Access: 26 juli 2016]*

[17] *Overview*
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_0_-_overview/ *[Access: 26 juli 2016]*

[18] *Understanding the platform*
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_1_-_understanding_the_xamarin_mobile_platform/ *[Access: 26 juli 2016]*

[19] *Architecture*
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_2_-_architecture/ *[Access: 26 juli 2016]*

[20] *Setting up a Xamarin cross platform solution*
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_3_-_setting_up_a_xamarin_cross_platform_solution/ *[Access: 26 juli 2016]*

[21] *Portable Class Library*
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries *[Access: 26 juli 2016]*

[22] *XAML*
https://msdn.microsoft.com/en-us/library/cc295302.aspx *[Access: 26 juli 2016]*

JSON

[23] *Official Site*
http://www.json.org *[Access: 26 juli 2016]*

[24] *Newtonsoft introduction*
http://www.newtonsoft.com/json/help/html/Introduction.htm *[Access: 26 juli 2016]*

[25] *An Introduction to JavaScript Object Notation (JSON) in JavaScript and .NET*
https://msdn.microsoft.com/en-us/library/bb299886.aspx *[Access: 26 juli 2016]*

Visual Studio

[26] *Visual Studio*
https://msdn.microsoft.com/en-us/library/fx6bk1f4(v=vs.90).aspx *[Access: 26 juli 2016]*

NuGet

[27] *NuGet*
https://docs.nuget.org/consume/overview *[Access: 26 juli 2016]*
[28] *What is NuGet*
http://geekswithblogs.net/TimothyK/archive/2014/03/27/what-is-nuget.aspx *[Access: 26 juli 2016]*