* Hvorfor lige Xamarin (krav fra systematic?)
* Hvordan vil det være designet uden Xamarin?
* Hvordan er det designet med Xamarin?
* Hvilke fordele og ulemper er der ved at tage en ny teknologi i brug til dette?

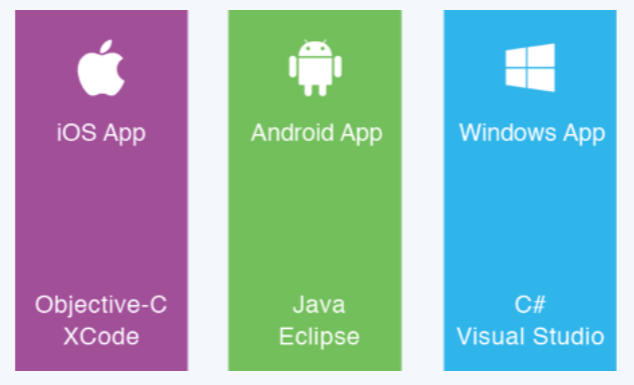
## PatientApp

### Cross-platform med Xamarin

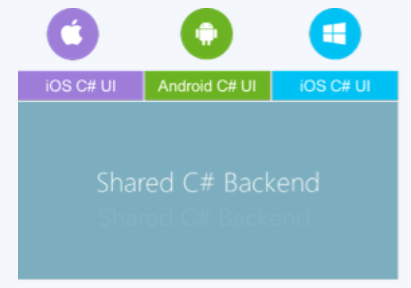
Ud fra kravspecifikationen og modulbeskrivelsen skal PatientApp’en udvikles i et miljø der gør det muligt at understøtte de tre største smartphone styresystemer iOS, Android og Windows, så de fleste af patienternes egne smartphones kan køre App’en. Den traditionelle løsning ville være at udvikle app’en hver for sig specifikt til hver af disse platforme og deres operativ systemer. Dette vil kræve en teknologisk viden til at programmere i 3 forskellige IDE’er, sprog og API’er:

* Xcode skrevet i enten objective-c eller swift.
* Android Studio skrevet Java.
* Visual Studio skrevet C#

Fordelen ved at benytte denne form for implementering for vores system, er at app’en vil kunne distribureres på alle platforme. Patienten vil ikke skulle være afhængig af at have en bestemt telefon for at kunne installere app’en, og det vil være fremragende for brugervenligheden. Ulempen vil være, at det rent teknisk vil koste meget udviklingstid, da hver styresystem er forskellig og som udvikler kræver det en bred, teknisk viden. Designmæssigt skal der designes en backend og en UI for hver platform.



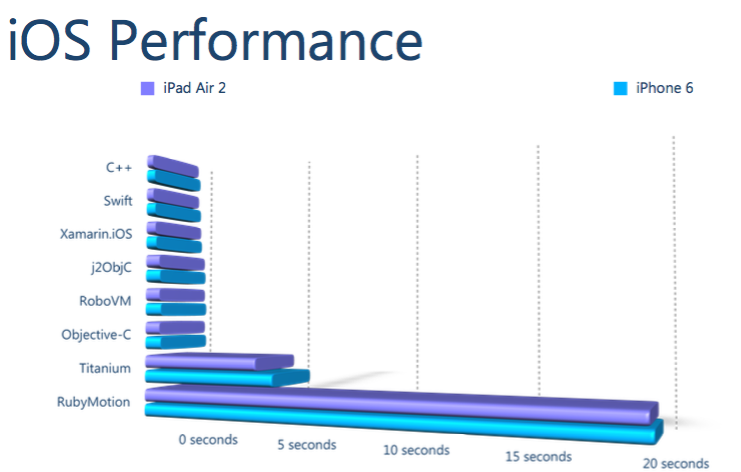
I stedet for overstående løsning, vil en endnu bedre, men ligeså fremragende løsning være brug af Xamarin platformen. Med Xamarin er det muligt at udvikle native iOS, Android og Windows apps i C# på en fælles kodebase. Her kan bruges den samme IDE, sprog og API alle steder. Alt hvad der kan udvikles i Objective-C, Swift eller Java, kan udvikles i C# med Xamarin. Med Xamarin apps har man adgang til the fulde spektrum af funktionalitet tilknyttet den underliggende platform og device, heriblandt platform specifikke API’er, såsom iBeacon og Android Fragments. Det dækkes 100 % med C#, dvs. man får det fulde udbytte for hver platform. Xamarin platformen kommer i to udgaver: En udgave hvor backenden er delt på tværs af alle platforme, men hvor UI’en skal udvikles for hver. Dette er den traditionelle Xamarin fremgangsmåde. Og en anden udgave er hvor både backenden og UI’en er delt. Dette er med Xamarin.Forms. Den løsning som PatientCare systemet benytter sig af, er den tradionelle Xamarin fremgangsmåde. Den ser således ud:



Men hvorfor er Xamarin bedst for f.eks. iOS?

* Apple’s iOS SDK kan fås 100 % med C#.
* Xamarin.iOS app’en kompileres direkte til native ARM assembly kode, som betyder at app’en kan distribueres direkte til App Storen, som man normalt ville.
* Det er muligt at benytte .NET navnekonventioner til at få adgang til Objective-C API’er.
* Kalde eksisterende Objective-C kode i C#

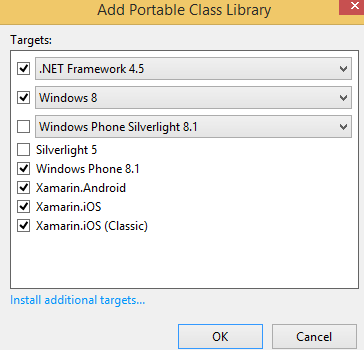
Performance mæssigt er en Xamarin løsning også bedre:



Det ses at Xamarin.iOS performer bedre end Objective-C. Det betyder meget for PatientCare systemt, at performance er god, da man som patient ikke skal blive frusteret over lange responstider – især ikke i kritiske tilfælde, som når vedkommende har moderate rygsmerter.

### Opsætning af cross-platform projekt

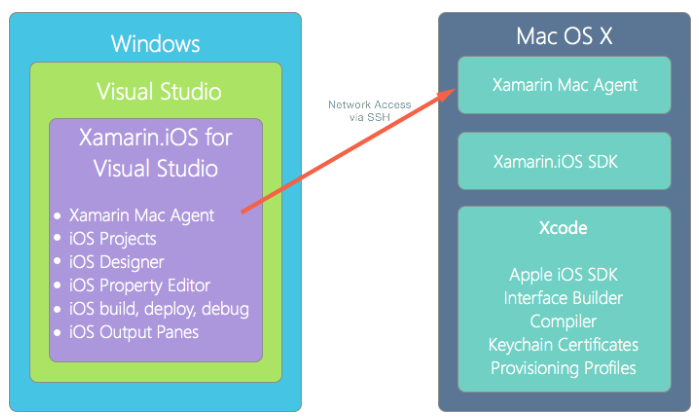
På Xamarins hjemmeside[[1]](#footnote-1) kan man installere et Xamarin-plugin til Visual Studio, hvilket er blevet gjort for PatientCare systemet, da denne IDE er den mest kendte. Xamarin installere automatisk de nødvendige SDK’er for Android og iOS – dog kræver det, at man har en Mac for at kunne oprette og bygge et Xamarin.iOS projekt. I guiden[[2]](#footnote-2) til at opsætte et cross-platform projekt laves et Portable Class Library projekt. Her vil bliver man spurgt hvilke targets denne PCL skal henvende sig imod:



Dernæst oprettes et projekt for iOS, Android og Windows. Da prototypen for PatientApp’en med udgangspunkt udvikles til iOS først, er der en række udfordringer.

1. En Mac er nødvendig
2. iOS applikationer kan ikke bygges uden Apple’s kompiler og de kan ikke deployes til et device uden Apple’s certifikater og code-signing værktøjer.
3. Xamarin.iOS SDK
4. Apple’s Xcode(7+) IDE og IOS SDK med den nyeste version.

Dette betyder at Xamarin.iOS projektet for Visual Studio kræver en forbindelse til en netværksforbundet Mac OSX computer for at udføre disse opgaver.



Kilde: http://developer.xamarin.com/guides/ios/getting\_started/installation/windows/#Linking\_to\_your\_Xamarin\_Account

Derudover kræver det en Apple Developer konto for at kunne deploye applikationen til App Storen.

Udover disse restriktioner er det super nemt at oprette og rediger iOS projekter og cross-platform løsninger som inkluderer Xamarin.Android og Xamarin.Windows projekter. For PatientCare systemet udvikles PatientApp’en på en privat Macbook Pro med fuld mulighed for at oprette, bygge og deploye et Xamarin.iOS projekt.

### Systemarkitektur

På baggrund af Xamarin platformen skal der bygges en systemarkitektur for PatientApp for at fastlægge byggeklodserne der tilsammen udgør en funktionel app. Arkitekturen for PatientApp er også vigtig for PatientCare systemet, da den skal være åben for nye ændringer og fremtidigt arbejde. Det indebære også evt. nye udviklere der skal forstå PatientApp’en og dens kommunikation med resten af PatientCare systemet og betydningen heraf.

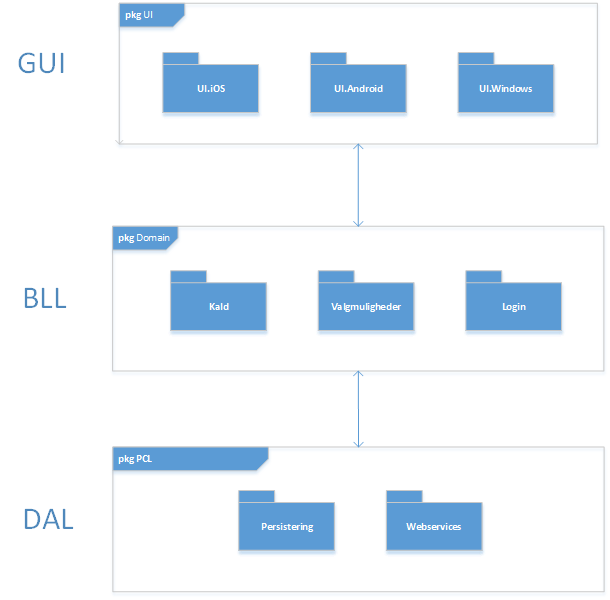
Da den tradionelle Xamarin løsning allerede sætter nogle tanker i gang omkring arkitekturen, er det alligevel vigtigt at forstå, hvad denne løsning og dens indre struktur helt præcist består af. For den tradionelle Xamarin løsning, er brugergrænsefladen platformsafhængig pga. de hver især har deres API og bibleoteker til at kunne implementere f.eks. en tabelliste. På iOS laves en tabel via en UITableViewController og et UITableView og på Android laves det med en Adapter og et ListView. Men domænet og baggrundslogikken er den samme. F.eks. når listen af valgmuligheder blive indlæst fra lokal databasen. Backenden dvs. alt tilgangen til en lokal database og forespørgsler til omverdenen kan via den tradionelle Xamarin løsning også være den samme. Det kræver blot en PCL.

For at kort opsummere kan den tradionelle Xamarin løsning byde på:

* Forskellig implementering af brugergrænsefladen.
* Fælles logik og kommunikation til omverdenen, heraf http request til en webserver.
* Fælles backend til at persistere data.

For at opbygge en god arkitektur der passer til denne løsning, er en oplagt fremgangsmåde at koncentrere alt kode relateret til fælles logikken og kommunikationen til omverdenen i et lag for sig selv og isoleret fra brugergrænsefladen og backenden. Dette gør nemt at lægge hver partition af PatientApp’en i sit eget ansvarsområde. Dette nedbryder også PatientApp’en i delsystemer, som ofte er opbygget som en lagdelt model af pakker og deres indbyrdes sammenhænge. Fordelen ved dette er man har ”separation of concerns”, og det reducer også koblingen og afhængigheder. Derudover giver det også et potentiel ”reuse”. Det betyder f.eks. at de nederste lag nemt kan blive brugt i andre applikationer. Og da der skal laves en mobil applikation til iOS, Android og Windows vil dette være til stor gavn, fordi data’en kan tilgås fra samme lag.

De tre typiske lag, en applikation har er: GUI, BLL, DAL

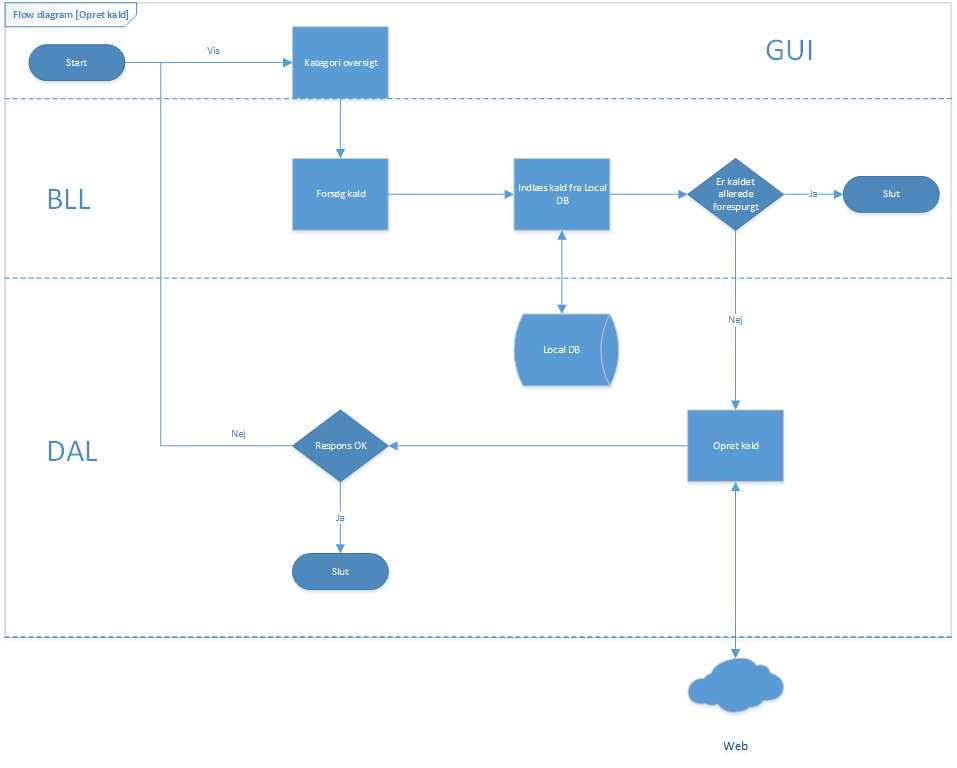


GUI (Også kaldt for PLL – Presentation Logic Layer) er laget, der indeholder en række forskellige pakker af forskellige API’er og bibleoteker der sammen med tilhørende logik udgør en brugergrænseflade, som patient kan se. Dette lag vil blive vist forskelligt afhængig af hvilken platform, PatientApp’en er installeret på.

BLL (Også kaldt for Business Logic Layer) er laget, der indeholder alt logikken for PatientApp’en med udgangspunkt i use casene, som patienten vil kunne interagere med. Patienten har f.eks. som mål at oprette et kald og den logik, der sørger for dette ligger, som det ses i Figur X, i en pakke kaldt ”Kald”.

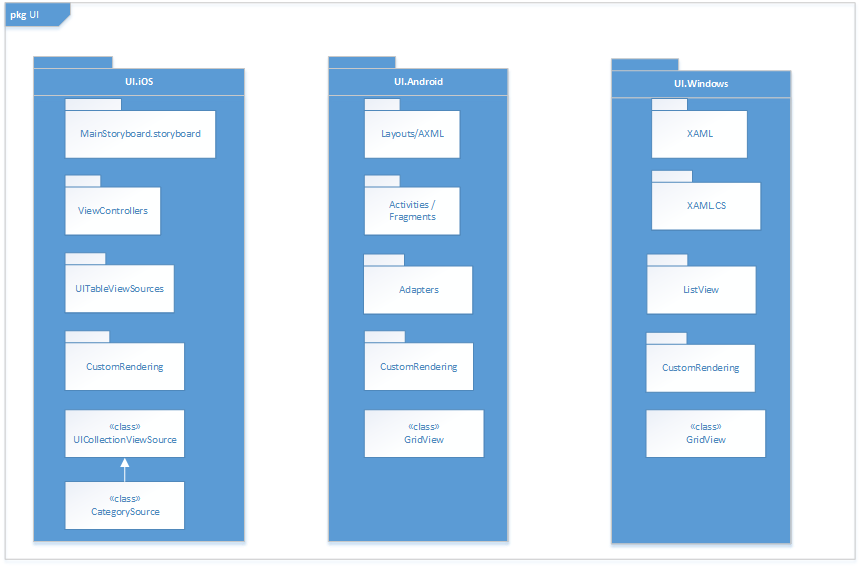
DAL (Også kaldt for Data Access Layer) er laget, der indeholder pakker som simplificere adgangen til data persisteret en database samt kommunikationen til omverdenen via nogle webservices. F.eks. vil pakken ”Persistering” som i Figur X returner en reference til et objekt med attributter i stedet for en række med felter fra en database tabel. Dette kan f.eks. være et objekt der indeholder attributter omkring en valgmulighed: Drikke som kategorien, Kaffe som typen og Mælk som detaljen. Dette lag skjuler dermed kompleksiteten af den underliggende datalagring fra omverdenen og kommunikationen indimellem.

Med udgangspunkt i at patient opretter kald, kan flowet for kommunikationen mellem de tre lag beskrives i følgende diagram:



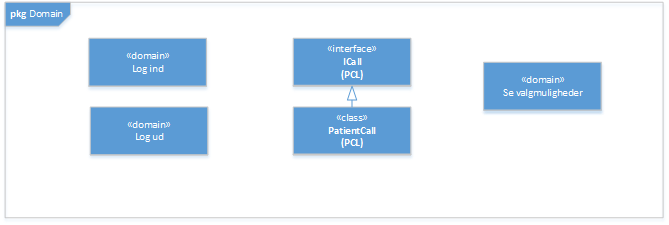
Som start får patienten vist kategorioversigten på brugergrænsefladen. Patienten vælger dernæst en årsag til et kald og trykker på en control i GUI’en der udløser et event. Dette event benytter sig af relevant logik i BLL laget for derefter at lave et tjek på lokal databasen. Kaldet til lokal databasen sker i DAL laget hvor der tjekkes på om kaldet allerede er forespurgt. Hvis kaldet ikke allerede er forespurgt, oprettes kaldet ved at lave en forespørgsel på Web API, som giver et svar tilbage. Er svaret ”OK” er casen slut, ellers får patienten vist kategorioversigten og kan prøve igen.

Som beskrevet i figur X kan pakkerne i hvert lag pakkes ud til nogle flere pakker. GUI laget har en pakke for hver platform men disse pakker indeholder en række forskellige controllers og andet logik til at få vist en brugergrænseflade.



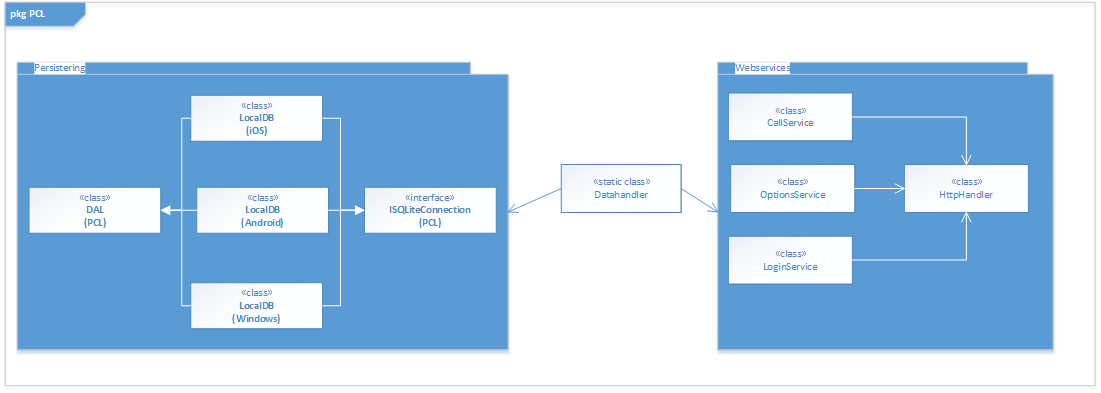
Hver platform har som sagt sin egne API’er og bibleoteker til at f.eks. fremstille et view. På iOS bruges Storyboards. På Android bruges Layouts/AXML og på Windows bruges XAML. Logikken bag hver view ligger i controllers. På iOS bruges ViewControllers, på Android Activites og/eller Fragments og på Windows Code-Behind c-sharp filer. Derudover er der andre custom renderinger dvs. forskellige måde at customisere en menu knap eller en tabel.

Ligesom GUI laget har pakker med andre pakker, har BLL laget det på samme vis.



Figur X tager udgangspunkt i kaldedomænet hvor patienten opretter et kald. Her er pakken ”Kald” pakket ud i et interface og en klasse som implementere dette interface. Grunden til at ”Kald” er lavet som et interface, er fordi at det er tænkt som et kald både i form af et portør kald fra Culoumnia systemet fra Systematic men også i form af et patient kald. Det opstiller nogle definitioner for en gruppe af relatede funktionaliteter som en klasse skal implementer. I dette tilfælde relaterede til et kald, i form af et portørkald, patientkald eller noget helt tredje.

DAL laget består af to pakker. En pakke der står for persistering af data og en pakke der står for kommunikationen til omverdenen. Hver af disse er pakket ude til følgende klasser:



Persisteringen består af nogle klasser som hver platform skal implementere, da hver af dem skal have deres egen lokale database. Men metoderne til at oprette tabeller, indlæse og gemme data kan alt sammen tilgås via den samme klasse. Dette kan lade sig gøre fordi, at der findes et API[[3]](#footnote-3) til SQLite, der gør muligt at gemme data i sqlite3 databaser med C# og kan kompileres til at virke på alle platforme. (Mono for Android, .NET, Silverlight, WP7, WinRT, Azure, osv).

Pakken der indeholder Webservices sørger for at lave http requests ud til et WebAPI, der sender respons tilbage i JSON-format, der derefter vil blive deserialiseret til konkrete objekter i DAL laget.

Alt data udtræk fra WebAPI’et og lokal databasen holdes der styr på i en klasse kaldt ”Datahandler”. Derved når man som udvikler har behov for enten at indlæse mine kald fra lokal database eller oprette et kald til WebAPI’et, kan udvikleren finde den rette metode i denne klasse.

### Fælles Backend(PCL)

Den traditionelle Xamarin fremgangsmåde indebære brugen af en fælles backend, også kaldt Portable Class Library. Her skal alt PatientApp logikken ligge og tages med overalt. Ud fra de funktionelle og ikke-funktionelle krav, er der blevet stillet følgende punkter op for hvad logik der kan deles i PCL’en, og hvad hver platform skal implementer:

**Hvad skal deles?**

* Indlæsning/gem af data herunder valgmuligheder og mine kald.
* HttpRequest til og fra adminsmodulet (WebAPI) og brugen af Object Relationel Mapping(ORM)
* CPR validering i native koden men også serverside validering via adminsmodulet.
* Andet logik, f.eks. en wrapper klasse til at pakke valgmuligheder til et kald, der senere skal persisteres.

**Hvad skal hver platform implementer?**

* Egen lokal database
* Filstien til hvor hver database lokalt skal ligge.
* Oprettelse til databasen
* UI og andet custom rendering såsom en menuknap eller en dialogboks. (da disse f.eks. jo er forskellig fra platform til platform)

På trods af at hver platform skal have deres egen lokale database, vil database opbygningen og implementeringen være den samme.

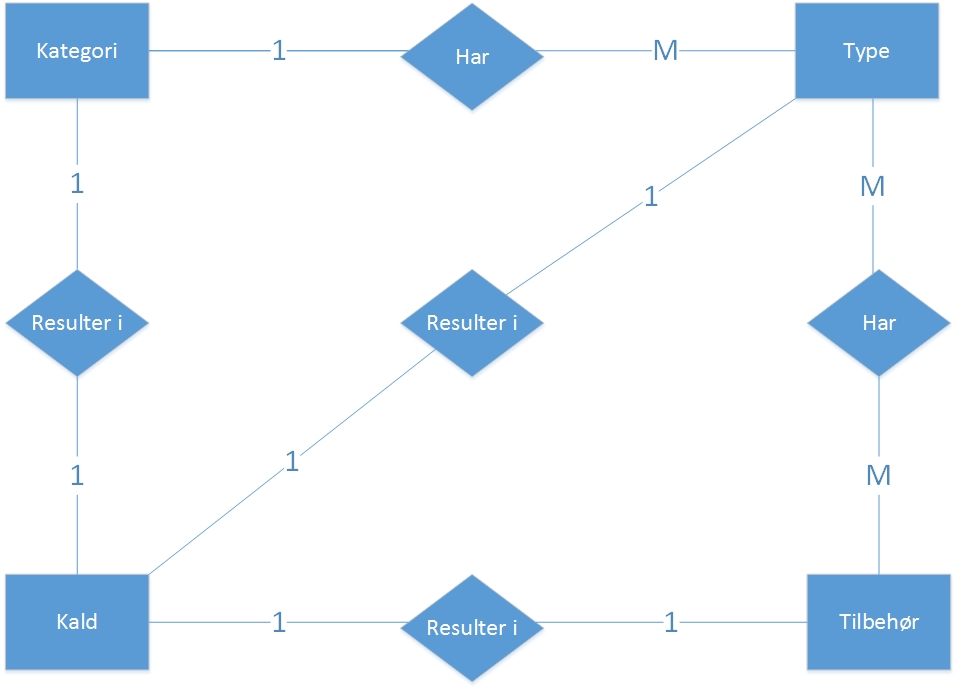
**Fælles Database opbygning**

Fremgangsmåden i at opbygge en fælles database tager udgangspunkt i objekt-orienterede systemudviklingsmetoder. Først løbes kravene igennem for PatientCare systemet, hvor entiterne identificeres, der i OO svarer til et objekt af bestemt type. Entiteterne er altså et objekt, som vi ønsker at modellere og gemme information om. De entiteter som er relevant for at blive persisteret lokalt på PatientApp’en er opstillet ud fra Use Casene. De er som følger:

* Valgmuligheder
  + Kategori
  + Type
  + Tilbehør
* Mine kald

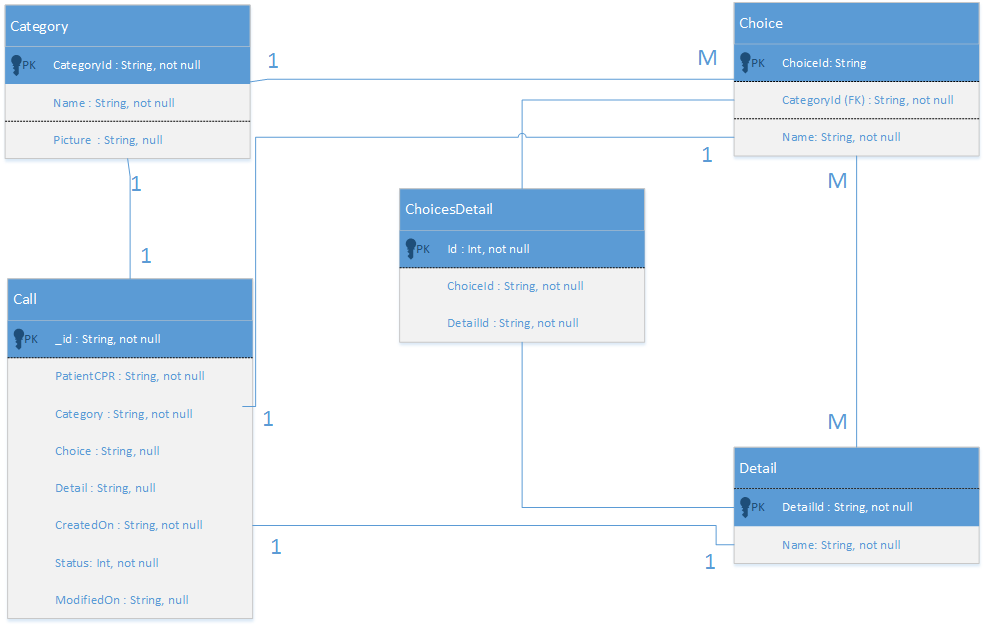
Årsagen til at Valgmuligheder og mine kald skal persisteres er, at man som patient stadig skal have mulighed for at navigere rundt i app’en selvom internettet er ustabilt eller ikke til rådigt. Det er hensigtmæssigt for brugervenligheden og det opretholder gennemsigtigheden, dvs. at app’en stadig kan tages i brug men at funktionaliteten er begrænset.

Entiteterne og deres relationsships opstilles i følgende ER diagram:



Af figur X ses det at en Kategori har en-til-mange Typer og en Type har mange-til-mange Tilbehør. Derudover er der en stærk binding mellem Kategori og Type og Type og Tilbehør. Med andre ord, en type skal tilhør mindst én kategori og én tilbehør skal tilhør mindst én type. Hver af disse valgmuligheder, som patient kan vælge, giver en årsag til et kald. Dette er wrappet i en entitet for sig selv kaldt ”Kald”. Relationerne er fundet ud fra en logisk tankegang men også på baggrund, hvad der er mest hensigtsmæssigt i forhold til hvad, en Patient kan vælge af serviceydelser.

Alt data der er relateret til en entitet er vist via attributter. En attribut er en property på en entitet og dens værdi kan findes ud fra dens domæne. Domænet er dette tilfælde valgmuligheder, herunder kategori, type og tilbehør og hver har en attribut af typen String med et navn på. Dette fører frem til følgende diagram:

****

Fremmednøglen for en-til-mange relationsship skal defineres i mange entiteten af relationsshippet. En-til-mange relationsships supporter Lister og Arrays, derfor har f.eks. Category en liste af choices, hvor hver af dem har en fremmednøgle tilbage til Category. Mange-til-mange relationsships kan ikke udtrykkes ved at bruge en fremmednøgle i en af entiterne, fordi fremmednøgler repræsenterer X-til-en relationsships. I stedet kræves en mellemliggende entitet, hvor disse fremmednøgler er erklæret. I figur X ses det at ”ChoicesDetail” er dette mellemliggende entitet. Denne vil dog aldrig blive brugt direkte i app’en, og der vil derfor ikke blive oprettet en tabel for den. Entiterne som er defineret her ender ud i klasser, som skal implementeres i en PCL og inkluderes i hver platform specifikke projekt. Men hvordan får PatientApp’en persisteret data’en fra disse klasser og indlæst igen, når der er behov for det? Dette løses ved at benytte en teknologi som SQLite gør brug af. Her vil det være muligt at mappe data’en fra lokal databaserne direkte til disse klasser helt automatisk. Hvorefter disse klasser kan traversere rundt på tværs af iOS, Android og Windows via PCL’en og benyttes i logikken og vises på brugergrænsefladen i app’en.

**SQLite-Extension**

Logikken i PatientApp skal kunne bygges på en PCL og deles for et Xamarin.iOS, Xamarin.Android og Xamarin.Windows projekt. Persistering af data skal dermed også være af samme fremgangsmåde og bygges i samme PCL. For dette er valgt en SQLite-Extension[[4]](#footnote-4), som er en meget simpel ORM der tilbyder alle de relationsships, database modellen vist i Figur X for PatientApp har brug for på toppen af et sqlite-net[[5]](#footnote-5) bibliotek. Det er open-source, der tillader .NET og Mono applikationer at gemme data i SQLite3 databaser. Med SQLite - Net Extension udvides sqlite3’s funktionalitet og hjælper bedre udvikleren til at håndtere relationships mellem sqlite-net entities. Med SQLite – Net Extension kan man via attributer på sine properties angive relationerne. Og man skal ikke oprette tabeller eller kolonner i databasen. Af denne grund, har man fuld kontrol over databasens skema til at persistere entiterne. SQLite-Net Extensions kræver kun, at man angiver fremmede nøgler, der anvendes til at håndtere relationer og resten finder SQLite den selv ud af.

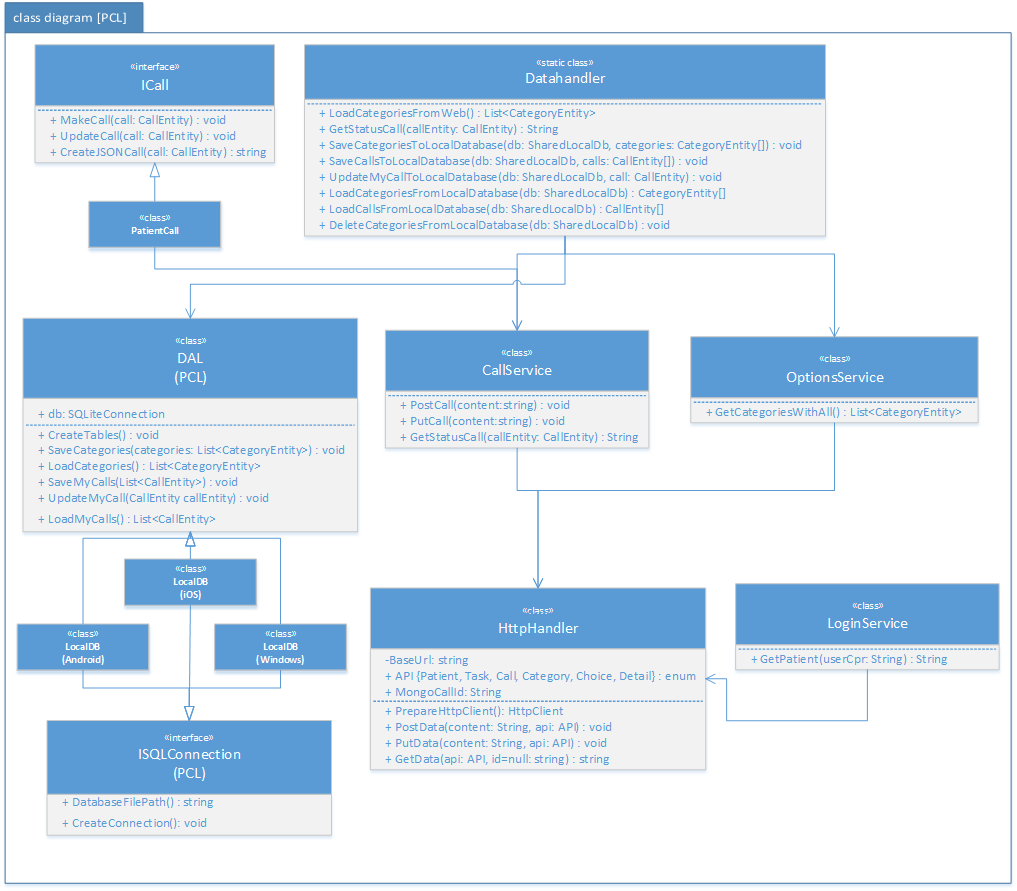
Et eksempel på brugen af SQLite-Net Extension er metoden GetAllWithChildren<Category>. Metoden kigger på alle de relationships der er specificeret i modellen vist i Figur X, finder eventuelle fremmednøgler og automatisk fylder properties i Kategori entititen. Man slipper altså for at skrive queries, som man normalt ville med sqlite3.

Som nævnt, skal hver platform implementere en metode til at returnere filstien for hvor sqlite databasen lokalt skal ligge. Og hver skal implementer en metode til at oprette forbindelse til deres sqlite database.

Som vist i Figur X er dette beskrevet som en interface klasse(ISQLiteConnection), som implementeres af tre platformsspecifikke klasser med hver af deres SQLiteConnection. DAL klassen har en række metoder der skal bruges ifm. Indlæsning/persistering af data fra SQLite databaserne, og denne klasse nedarves også i de samme, tre platformsspecifikke klasser.

### Klassediagram for PCL

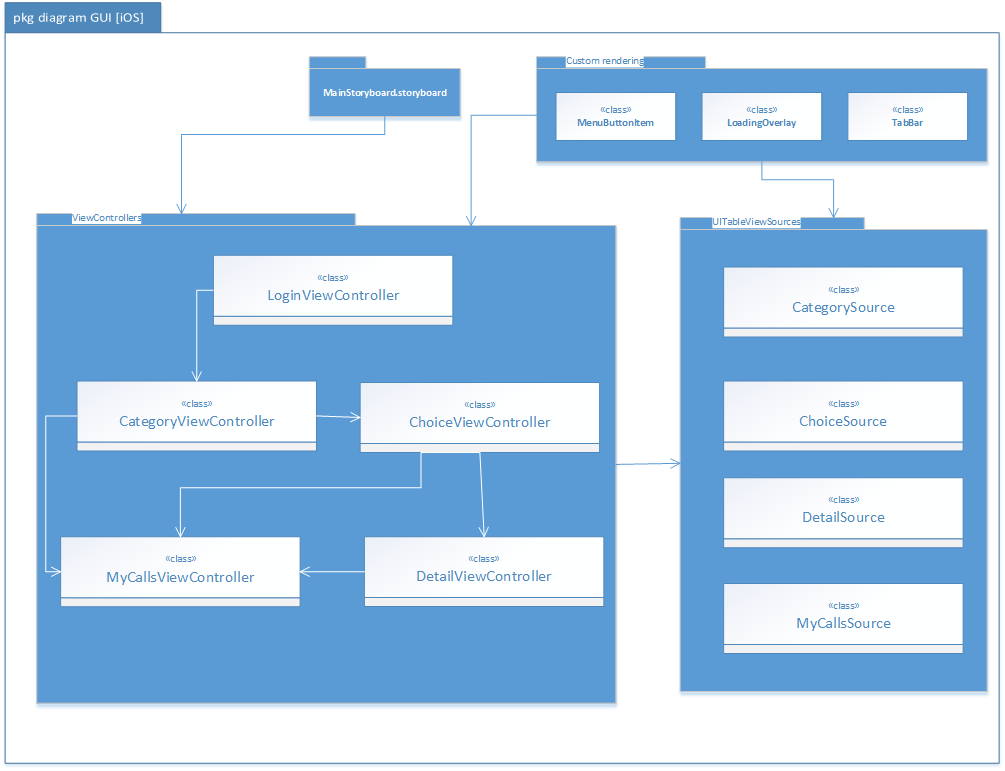
For at beskrive strukturen i PatientApp vises nedstående et klassediagram for PCL’en. Dette viser PatientApp’s klasser for PCL’en, deres attributter, metoder og relationen mellem objekterne. Dette klassediagram tager udgangspunkt i Figur X, hvor DAL er beskrevet.



Klassen ”Datahandler” som kort beskrevet i Figur X under DAL laget har en association til CallService, som håndtere oprettelsen og opdateringen af et kald. Den sørger også for at hente status(Aktiv, Udført, Fortrudt) på et specifikt kald. Ligeledes har ”Datahandler” en association til ”OptionsService” som henter alle valgmuligheder fra adminsmodulet. Hver service har association til en ”HttpHandler” klasse hvor den konkrete implementering af httprequest og httprespons til et WebAPI sker.

### Klassediagram for GUI (iOS)

Nedstående viser klassediagram for GUI’en for PatientApp med udgangspunkt i prototypen, som er udviklet til Xamarin.iOS.



På iOS arbejder man med storyboards[[6]](#footnote-6) som er en visuel repræsentation af brugergrænsefladen og flowet af applikationen. Det indeholder en sekvens af alle scener hvor hver scene repræsenterer en viewcontroller og dens views. Disse views vil indeholde bestemte objekter og kontroller som gør at brugeren kan interagere med applikationen. I figur X ses det at ”MainStoryboard.storyboard” har en association til en pakke med viewcontrollers. Essensen er at hver viewcontroller er tilknyttet et view eller en scene i storyboadet. Disse viewcontrollers er logikken bag viewet og alt håndtering af brugerinteraktion sker her. Det ses at pakken med viewcontrollers består af 5 viewcontrollers:

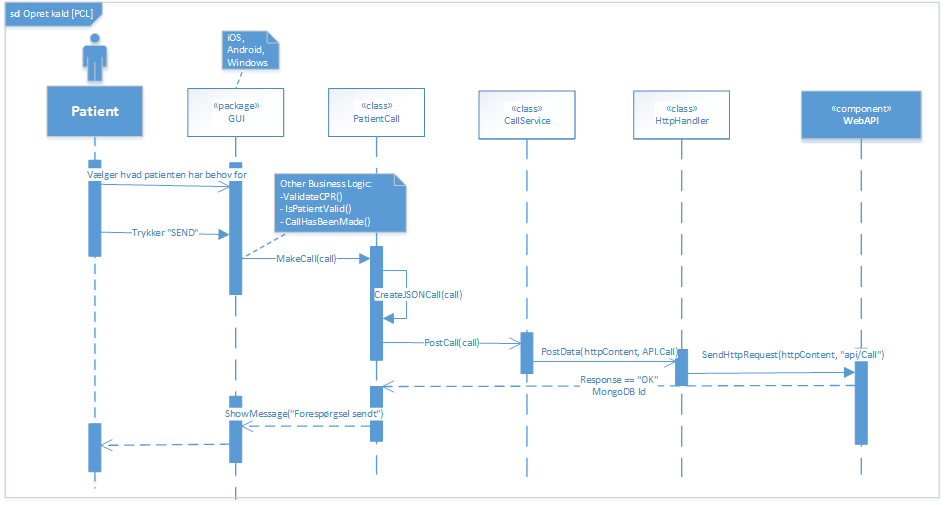
* LoginViewController – står for at håndtere login, heraf CPR validering og tjek af patient indlæggelse
* CategoryViewController – står for at håndtere visning og valg af kategorier
* ChoiceViewController – står for at håndtere visning og valg af typer tilknyttet en kategori
* DetailViewController – står for at håndtere visning og valg af detailer tilknyttet en type.
* MyCallsViewController – står for at håndtere visning af mine kald og holder styr på status på et kald (om det er aktivt, udført eller fortrudt)

Hver viewcontroller har en association til en ”UITableViewSource” som er en klasse der fungerer som ”code behind” mht. visning af en tabel, for hvor hver valgmulighed bliver vist i.

## Design

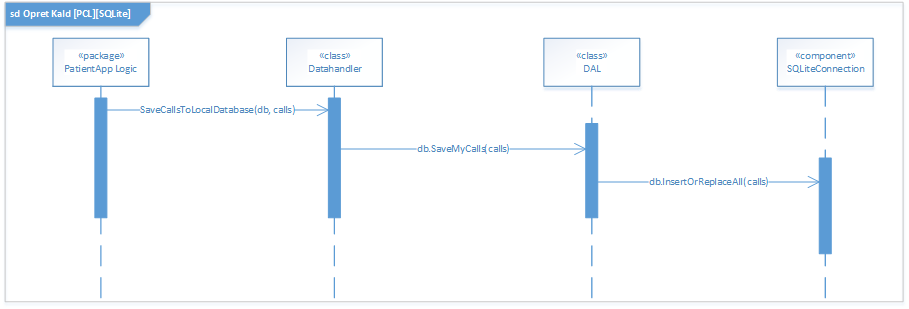
### Sekvensdiagram(Opret kald)

Med udgangspunkt i Use Casen ”Opret kald” laves et sekvensdiagram der beskriver sekvensen fra at en patient vælger en årsag til et kald og sendes afsted til det bliver modtaget med en bekræftelse.



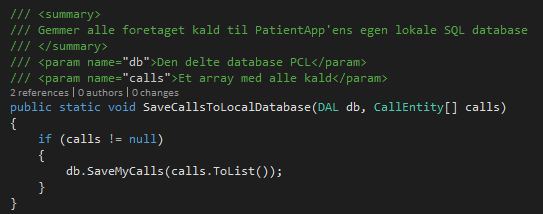
Patienten har behov for at oprette et kald. Patienten vil finde ud af der er flere steps han/hun skal igennem afhængig af om kaldet skal specificeres yderligere. Dette foregår via brugergrænsefladen via en form for tabeloversigt illustreret i Figur X som en pakke af grafiske elementer, da tabellernes opbygning er platformsspecifikt. Efter patienten har valgt en valgmulighed på brugergrænsefladen, sker der en række andre ting som vedrører business logikken, bl.a. CPR validering og tjek på om patienten er registreret i PatientCare systemet. Derefter oprettes kaldet i ”PatientCall” klassen, hvor der laves en JSON repræsentation af kaldet. Denne JSON repræsentation sendes videre til en service, der sørger for at poste det videre til ”HttpHandler” klassen. Denne klasse står for at opstille http clienten med request headers, body og url’en til WebAPI’et sammen med det indhold, der skal sendes af Content-Typen ”application/json”. WebAPI’et giver derefter en response kode. Er denne ”OK” betyder det, at kaldet er blevet oprettet. I responsen fra WebAPI’et gives der et id på kaldet, der netop er blevet oprettet i MongoDB. Dette id skal gemmes sammen med dette kald, der netop er blevet oprettet af patienten. På denne måde er id’et på kaldet ens i både lokal databasen men også på fællesdatabasen på MongoDB. Dette er vigtigt, da samme kald senere skal opdateres enten i form af, at det fortrydes af patienten eller udføres af personalet.

I Figur X er der ikke vist den del af sekvensen, hvordan kaldet persisteres i lokal databasen efter det er blevet oprettet. Dette vises i et separat sekvensdiagram som følgende:

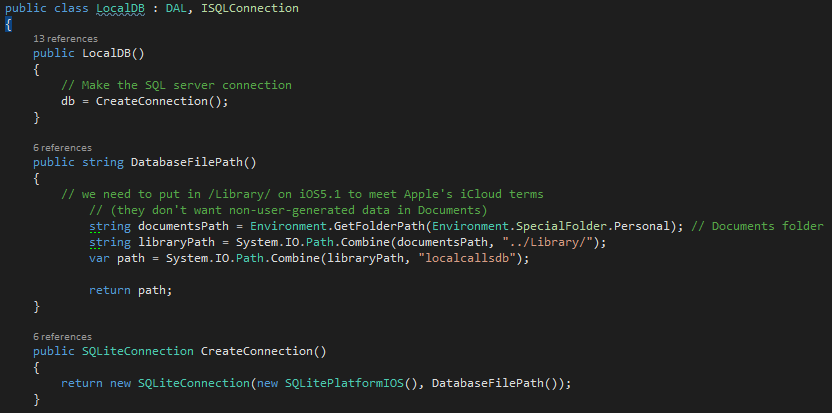


Efter responset fra WebAPI’et vist i Figur X, tilføjes kaldet sammen med det returneret MongoDB id til en liste af andre kald, som lokal databasen har persisteret. Listen sendes med metodekaldet til Datahandleren sammen med en reference til den platformsspecifikke, lokaldatabase klasse. Herefter kaldes en metode på denne reference til at sende kaldet videre til den platform, hvor det er implementeret i DAL. Sidst sker selve indsættelse af kaldet via en metode ”InsertOrReplaceAll” fra SQLite API’et.

Et kodeudsnit fra Datahandler hvor ”SaveCallsToLocalDatabase” er vist her:



db objektet som sendes med her er af typen DAL som hver platform skal nedarve fra. Et eksempel på hvordan det fungerer ses i klassen LocalDB:



Eksemplet viser hvordan DAL og SQLiteConnection klassen implementeres, hvori constructoren angives en implementering af ISQLitePlatformen og filstien til lokaldatabasen. LocalDB klassen skal ligge sammen med Xamarin.iOS projektet og kopiers i Xamarin.Android og Xamarin.Windows projektet. Her skal ISQLConnection og DAL implementeres på ny og de samme metoder overskrives på samme måde som i Figur X. Sammen med overstående to metoder, som hver platform skal overskrive, får de også hver især de samme metode definitioner fra DAL klassen. Disse metoder kan ses i klassediagrammet i Figur X.

### Forbedring/Videreudvikling

#### Xamarin.Forms

I forbindelse med teknologiundersøgelse mht. Xamarin stødte gruppen i bedre tekniske alternativer ifm. Et oplæg hos Mjølner Informatics omkring Xamarin platformen. Der blev drøftet to Xamarin løsninger, hvor den ene løsning var den traditionelle løsning, som PatientCare systemet har valgt, og en anden løsning som gør brug af Xamarin.Forms. På dette tidspunkt var gruppen allerede godt i gang med implementeringen af den traditionelle løsning men alligevel fik gruppen rigtig meget inspiration til hvordan PatientApp’en kunne have udvikles på en mere effektiv, og hurtigere måde for Proof of Concept af PatientCare systemet. På runtime vil hver side og dens kontroller blive mappet til platform-specific native brugerelementer. F.eks. vil et tekstfelt med Xamarin.Forms blive til et UITextView på iOS, EditText på Android og TextBox på Windows.

**Fordele**

Xamarin.Forms bygger på også på princippet om at have en C# fælles kodebase som den traditionelle Xamarin løsning, men med Forms deles UI også og dermed er der 100 % code sharing. Xamarin.Forms elementer mappes automatisk til nativ kontroller og adfærd.

**Ulemper**

UI’en skal skrives direkte i C# eller i XAML, da det ikke er muligt at bruge drag-and-drop funktionalitet som på den tradionelle måde, eftersom UI’en er forskellig. Derudover er custom rendering svært og bør undgås i Xamarin.Forms, da man ellers ligeså godt kan bruge den tradionelle, da det skal laves for hver platform. Style layoutet er også svær, da hver layout er forskellig fra hver platform, f.eks. font størrelsen kan være forskellige selvom samme størrelse er sat på dem alle. Derudover skal man kende til de forskellige platformes API’er på trods af alt skrives i C#.

For PatientCare systemet har den tradionelle Xamarin være en udmærket løsning. Kendskab til iOS, Android og Windows udvikling har på forhånd være god, hvilket har gjort det nemt at sætte sig ind i de forskellige API’er med brug af C#. Men havde systemet benyttet sig af Xamarin.Forms vil det udviklings-og tidsmæssigt være en meget bedre løsning for Proof of Concept og pga. den korte udviklingstid i forhold nuværende løsning ville prototypen af PatientApp hurtigere havde nået en Android og Windows løsning. Det ville have betydet, at systemet ville have bedre levet op til konceptet om ”bring your own device”, som var det mål der blev sat fra start.

#### Push i stedet for pull

Prototypen for PatientApp fungerer lige nu sådan, at den benytter sig af pull funktionalitet, når nye valgmuligheder skal indlæses fra adminsmodulet. Samme fremgangsmåde benyttes når status på mine kald for patienten skal opdateres. Her skal patienten manuelt trække ned i på viewet for at refreshe og dermed lave et pull request for at opdatere data’en. Dette er ikke hensigtmæssigt for brugervenligheden, når patienten måske i nød ligger og har rygsmerter og gerne vil have respons med det samme, så snart kaldet er sendt. I stedet vil en bedre løsning være at lave et push request, når data’en har ændret sig. Grunden til at prototypen benytter sig af pull funktionalitet er, at teknologi mæssigt er det hurtigere og nemmere, og det var netop målet med prototype. I fremtidigt arbejde vil det være oplagt at benytte push funktionaliteten og det vil en yderst forbedring for brugervenligheden og brugeroplevelsen for patienten.

### Udfordringer

En af de udfordringer der har været med at udvikle PatientApp’en har været at sætte sig ind i Xamarin platformen og opsætte et cross-platform projekt med Visual Studio. Det har været en udfordring, fordi det er ny teknologi og det kræver meget indsigt i forskellige styresystemer og deres API’er på trods af, at applikationerne programmeres i C#. Det har også været en udfordring at få bygget og deployet et Xamarin.iOS projekt i Visual Studio. Da Visual Studio er en IDE, der kun som udgangspunkt kan køres på Windows, kræver det en virtual maskine på en Mac OSX computer, hvor en Mac Build OS server på Windows skal kunne snakke med en Mac Build OS server på Mac OSX computeren. Kan man ikke kobles på build serveren i Visual Studio, kan projektet ikke bygges. Og det har haft udfordringer fordi versionen af denne server skal passe i begge styresystemer – både i den virtuelle og ikke-virtuelle. Og da Xcode kom med den nye version 7,1 skulle build serveren opdateres, hvilket gav nogle problemer.

### Resultat

PatientCare systemet har formodet at lave en cross-platform applikation, som patienterne kan installere og bruge på deres egen telefon. Cross-platform løsningen er dog kun lavet til iOS da denne er en prototype med det mål at få vist konceptet af PatietCare systemet. Funktionalitets mæssigt har applikationen opfyldt alle Use Casene på baggrund af prototypen.

1. https://xamarin.com/download [↑](#footnote-ref-1)
2. http://developer.xamarin.com/guides/cross-platform/getting\_started/visual\_studio\_with\_xamarin/cross\_platform\_visual\_studio\_project\_setup\_walkthrough/ [↑](#footnote-ref-2)
3. https://github.com/oysteinkrog/SQLite.Net-PCL [↑](#footnote-ref-3)
4. https://bitbucket.org/twincoders/sqlite-net-extensions [↑](#footnote-ref-4)
5. https://github.com/praeclarum/sqlite-net [↑](#footnote-ref-5)
6. https://developer.xamarin.com/guides/ios/user\_interface/introduction\_to\_storyboards/ [↑](#footnote-ref-6)