Joonas Virtanen

LINQ-hakuarkkitehtuuri ja sen käyttäminen SQLite-tietokannassa Windows 10 Mobile -alustalla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

28.3.2016

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Joonas Virtanen LINQ-hakuarkkitehtuuri ja sen käyttäminen SQLite-tietokannassa Windows 10 Mobile -alustalla 36 sivua + 1 liitettä 28.3.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmointi
Ohjaaja(t)	tutkintopäällikkö Markku Karhu

Insinöörityön aiheena oli esitellä ”Language Integrated Query” -hakuarkkitehtuuria, joka on lyhyemmin LINQ. Insinöörityötä varten oli tehty Windows 10 Mobile -ohjelma IHaveIdeas, jonka lähdekoodia käytetään SQLiten tietokantaoperaatioiden koodiesimerkeissä. Ohjelma on ideageneraattori, jonka tarkoituksena on antaa käyttäjälle inspiraatiota esimerkiksi tarinan luomiseen. IHaveIdeas-ohjelman tietokanta tehtiin kahdella eri tavalla, jotka on esitelty tässä työssä.

Tämän insinöörityön tavoitteena oli ymmärtää LINQ-hakuarkkitehtuurin toimintaa ja tarkastella, miten sitä voi käyttää hyödyksi oikeassa sovelluksessa, kuten mobiiliohjelman tietokantaoperaatioissa. Koska insinöörityötä varten laadittu ohjelma IHaveIdeas on tehty Windows 10 Mobile -alustalla, niin insinöörityössä esiteltiin myös Windows 10 -alustaa. Työn tavoitteena oli myös ymmärtää Windows 10 -alustalle ohjelman kehittämisestä.

Tässä insinöörityössä tehdyissä LINQ-esimerkeistä ilmeni, että LINQ-hakuarkkitehtuuri on erittäin kätevä tiedonkäsittelyssä. LINQ-hakuarkkitehtuurilla voi saada tiedonhaut erittäin helposti luettavaksi verrattuna muihin vaihtoehtoisiin tapoihin. Insinöörityö on tarkoitettu niille, joilla on jo kokemusta .NET-alustasta. Tämä insinöörityö voi auttaa .NET-ohjelmoijia paremmin ymmärtämään LINQ-hakuarkkitehtuuria.

Avainsanat	LINQ, SQLite, tietokannat, Windows 10, C# 3.0

Author(s) Title Number of Pages Date	Joonas Virtanen LINQ query architecture in Windows 10 Mobile in SQLite database 36 pages + 1 appendix 28 March 2016
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Programming
Instructors	Markku Karhu, Dean

The aim of this study was to introduce “Language of Integrated Query” query architecture, the acronym of which is LINQ. For this study Windows 10 Mobile software called IHaveIdeas was created. The source code of IHaveIdeas was used in the SQLite database operation examples. The program is an idea generator, which is was designed to give the user an inspiration, for example, ideas to create a story. The database of the IHaveIdeas program was made in two different ways, which are presented in this thesis.

The aim of this thesis was to understand the LINQ query architecture operations and how it can be used in real applications, such as in the database operations of a mobile application. Because the program IHaveIdeas, which was created in this study, has been made for the Windows Mobile 10 platform, this thesis also presents the Windows 10 platform. The aim was also to understand application development for the Windows 10 platform.

LINQ code examples showed that the LINQ query architecture is very useful when processing data. With the LINQ query architecture, it is much easier to read the data queries compared to other alternative methods. The study is intended for those who already have some experience of the .NET platform. This study can help .NET programmers to better understand the LINQ search architecture.

Keywords	LINQ, SQLite, databases, Windows 10, C# 3.0

Sisällys

Lyhenteet

LINQ	Language Integrated Query. LINQ-hakuarkkitehtuuri on .NET-komponentti, joka mahdollistaa datakyselyjen suorittamisen kokoelmia vasten.
SQL	Structured Query Language. Erittäin suosittu hakukieli.
C#	Microsoftin luoma oliopohjainen ohjelmointikieli.
C++	Bjarne Stroustrupin kehittämä olio-ohjelmointikieli.
Java	Sun Microsystemsin luoma olio-ohjelmointikieli.
XAML	Transaction Authority Markup Language on XML-pohjainen merkintäkieli.
CLR	.NET Framework -alusta tarjoaa ajonaikaisen käyttöympäristön nimeltään Common Language Runtime, joka juoksee itse koodin ja tarjoaa palveluita, jotka tekevät kehitysprosessista helpompaa.
API	Application programming interface. Ohjelmointirajapinta, jolla eri ohjelmat voivat tehdä pyyntöjä ja vaihtaa tietoa.

1 Johdanto

Tässä insinöörityössä selvitetään LINQ-hakuarkkitehtuurin toimintaa ja sen käyttämistä SQLitessä Windows 10 Mobile -alustalla. Insinöörityön alussa esitellään C# 3.0:n tärkeimmät ominaisuudet, Windows 10 -alusta, LINQ ja sen käyttö SQLite-tietokannassa. Insinöörityössä on tehty Windows 10 Mobile -ohjelma, jonka koodia käytetään työn koodiesimerkeissä.

Päätavoite tässä insinöörityössä on oppia LINQ-hakuarkkitehtuurin toiminnasta. Työn tavoitteena on myös ymmärtää, miten LINQ-hakuarkkitehtuuria voi käyttää hyödyksi mobiilisovelluksen tietokannan tietokantaoperaatioissa. Koska insinöörityötä varten tehty ohjelma ”IHaveIdeas” on tehty Windows 10 Mobile -alustalla, niin insinöörityössä esitellään myös Windows 10 -alusta. Työssä kuitenkin keskitytään Windows 10 Mobile -alustaan. Työn aihe on valittu sen takia, koska LINQ-hakuarkkitehtuuri herätti oman mielenkiinnon tietokantoihin ja .NET-maailmaa kohtaan.

Tässä työssä esitellään, miten voidaan tehdä erilaisia tietokantaoperaatioita SQLite-tietokannassa käyttäen Windows 10 Mobile -ohjelmaa. SQLite:n kanssa on käytetty SQLite.Net-PCL-käärettä, joka mahdollistaa LINQ-kyselyt. Koodiesimerkkejä käyttäen näytetään, miten SQLite-tietokanta voidaan luoda ja miten voidaan lisätä, päivittää ja poistaa tietoa tietokannasta. Insinöörityötä varten on tehty kaksi eri tapaa tehdä SQLite-tietokanta. Ensimmäinen on paikallinen tietokanta ja toinen on eristetty tietokanta. Tässä työssä käydään läpi, mitä eroja näillä kahdella tietokannalla on, ja esitellään, miten ne kumpikin luodaan.

Insinöörityössä on tehty Windows 10 Mobile -ohjelma, jonka lähdekoodia käytetään LINQ-esimerkeissä. Ohjelma on ideageneraattori nimeltä IHaveIdeas, jonka tarkoituksena on antaa käyttäjälle inspiraatiota esimerkiksi tarinan luomiseen. Ohjelma esitellään tarkemmin luvussa 6. Windows 10 Mobile -ohjelma on tehty C#-ohjelmointikielellä.

LINQ tai ”Language Intergrated Query” julkaistiin osana C# 3.0:aa ja Framework 3.5:sta. LINQ on hakuarkkitehtuuri, joka on integroitu C#- ja Visual Basic -ohjelmointikieliin. LINQ antaa uudenlaisen tavan hakea tietoa tietokannasta. Käyttäen LINQ-hakuarkkitehtuuria ohjelmoija voi hakea dataa XML-tiedostosta, relaatiotietokannasta tai oliokokoelmasta.

Mikä erottaa LINQ-hakuarkkitehtuurin muista ohjelmointirajapinnoista on sen integroituminen C# ja Visual Basic -ohjelmointikieliin. Mitään ylimääräisiä kirjastoja ei tarvitse asentaa, jos haluaa käyttää LINQ-hakuarkkitehtuuria Visual Basic- tai C#-ohjelmointikielillä. [1.]

2 C# 3.0:n tunnusomaiset omaisuudet

C#-ohjelmointikielestä on elämänkaaren aikana julkaistu kuusi eri versiota. Tässä insinöörityössä esitellään versio C#:n 3.0, koska LINQ julkaistiin ensin C# 3.0 -version mukana. Melkein kaikki C# 3.0:n uusista ominaisuuksista koski LINQ-hakuarkkitehtuuria, paitsi automaattiset ominaisuudet, jotka myös esitellään tässä työssä. LINQ-hakuarkkitehtuuria esitetään tarkemmin luvussa 4 ja 5.

C#-ohjelmointikieli on tarkoitettu olemaan yksinkertainen, moderni, oliopohjainen ja tyyppiturvallinen ohjelmointikieli. C#-ohjelmointikieltä on vahvasti vaikuttanut C-ohjelmointiperhe, jonka ansioista C-, C++- ja Java-ohjelmoijien on helppo tutustua kieleen.

Turvallinen tyyppijärjestelmä tuo turvallisuutta C#-ohjelmointikieleen, jota C- ja C++-ohjelmointikielillä ei ole. C#-ohjelmointikielessä tyyppiturvallinen koodi ei voi esimerkiksi lukea arvoja toisesta luokasta, jos ne on merkattu ”private”-suojausmääreellä. Tyyppiturvallinen ohjelmointikieli ei myöskään salli ohjelmoijan käyttää string-tyyppiä int-tyyppisenä, ilman erillistä muutosta. C#-ohjelmointikieli on tunnettu vahvasti tyypitettynä ohjelmointikielenä, koska sen tyyppisäännöt ovat erittäin kovat. [2, s. 2.]

2.1.1 Implisiittisesti tyypitetty paikallinen muuttuja

Implisiittisesti tyypitetty paikallinen muuttuja ”var” antoi C# 3.0 -versiossa vaihtoehtoisen tavan alustaa paikallisia muuttujia. Koodiesimerkissä 1 näkyy, miten Dictionary alustettiin ennen kuin C# 3.0 julkaistiin. Kuten koodiesimerkissä 1 näkyy, Dictionaryn alustaminen on turhan pitkä. Directory-tyyppi ilmoitetaan kaksi kertaa yhtä suuri kuin merkin kummallakin puolella.

```
Dictionary<string, List<PhoneNumber>> whitepages = new Diction-
ary<string, List< PhoneNumber>>();
```

Koodiesimerkki 1. Dictionary alustaminen ennen C# 3.0:n julkaisemista.

Koodiesimerkissä 2 näkyy, miten C# 3.0:n versiossa voi Dictionaryn alustaa muuttujalla ”var”. Seuraavassa koodiesimerkissä annetaan kääntäjän määritellä, mitä tyyppiä Dictionary on. Implisiittisesti tyypitetyllä paikallisella muuttujalla voi siis määritellä muuttujan, jolle ei tarvitse antaa erikseen tietotyyppiä yhtä suuri kuin -merkin vasemmalle puolelle. Hyötynä paikallisissa muuttujissa on tietenkin se, että koodin määrä on pienempi ja sitä on helpompi lukea. [3, s. 65–66.] Seuraavassa luvussa esitellään, miten var-muuttujaa käytettiin LINQ:in kanssa.

```
var whitepages = new Dictionary<string, List<PhoneNumber>>();
```

Koodiesimerkki 2. Dictionary alustaminen paikallisella muuttujalla var.

2.1.2 Anonyymit tyypit

Anonyymit tyypit ovat kätevä tapa kapseloida joukko vain luettavia ominaisuuksia yhteen olioon, ilman että tarvitsee määritellä tyyppiä ensin, kuten koodiesimerkissä 3 näytetään.

```
var v = new { Id = 108, Email = "joonas.virtanen@gmail.com" };
Console.WriteLine(v.Id + v.Email);
```

Koodiesimerkki 3. Anonyymi tyyppi on alustettu kahdella ominaisuudella.

Ohjelmoija voi luoda anonyymejä tyyppejä käyttämällä new-operaattoria olioalustamisen kanssa. Anonyymi tyyppi voi sisältää yksi tai useampi vain luettavia ominaisuuksia. Anonyymiä tyyppiä ei voi käyttää määrittelemässä luokkia, metodeja tai tapahtumia. [4.]

Anonyymi tyyppejä voi nähdä LINQ hakujen select-kohdassa. Alla on koodiesimerkki 4, jossa tehdään LINQ-haku anonyymityyppilistasta. Haussa myös uudelleenimetään haun anonyymien tyyppilistan kenttien nimet kohdassa select. Alla olevassa koodiesimerkissä annetaan kääntäjän määritellä, mitä tyyppiä hakuoperaatio on, joka helpottaa koodin lukua huomattavasti.

```
var customersList = new[] { new { Name = "Joonas Virtanen", Age = 22
}, new { Name = "Teemu Hynninen", Age = 26 } };

var result = from customers in customersList
             where customers.Age > 23
             orderby customers.Name
             select new {  CustomerName = customers.Name, CustomerAge
= customers.Age};
```

```
// result: { CustomerName = Teemu Hynninen, CustomerAge = 26 }
```

Koodiesimerkki 4. Anonyymityyppi-listasta LINQ-haku.

2.1.3 Olioiden alustaminen

Olioiden alustaminen sai muutosta C# 3.0 -versiossa. Olioitten alustaminen pyrittiin saamaan yksinkertaisemmaksi ja helpommaksi, kuten koodiesimerkissä 5 näkyy. C# 2.0:n olion alustamisessa koodi oli useammalla kuin yhdellä rivillä, kun määritteli arvot oliolle, jossa kaikki alustettavat arvo eivät olleet konstruktorissa. C# 3.0:ssa ohjelmoija pystyi alustamaan olion ominaisuudet, jotka eivät olleet konstruktorissa, samalle riville konstruktoriarvojen alustamisen kanssa. Koodiesimerkissä 5 alustetaan arvot aaltosulkujen sisällä, jotka eivät ole konstruktorissa.

```
class ObjectInitialize
    {
        static void Main()
        {
           //Old way to initialize object
           Contact contact1 = new Contact("Joonas");
           contact1.Email = "joonas.virtanen@gmail.com";

           //New way to initialize object
            Contact contact2 = new Contact("Sanna") { Email =
           "sanna.lehtonen@gmail.com" };
        }

    }
    class Contact
    {
        public string Name { get; set; }
        public string Email { get; set; }
        public Contact(string name)
        {
            Name = name;
        }
    }
```

Koodiesimerkki 5. Vanha ja uusi tapa alustaa olio.

Olioiden uusi alustamisominaisuus näkyi LINQ-hakuarkkitehtuurissa hakujen Select-kohdassa. Alla olevassa koodiesimerkissä 6 palautetaan lista Customer-olioita. Ilman kohtaa ”{ Age=customers.Age}” haku palauttaa kokoelmaan oldCustomerAges Customers -olioita, joissa on nimi mutta ikä on jokaisessa arvoa 0. Olioiden uusi alustamistapa

koodiesimerkin 6 aaltosuluissa mahdollistaa iän arvon lisäämisen hakuun, jos kyseistä arvoa ei ole luokan konstruktorissa.[3, s. 66-67.]

```
var oldCustomerAges = from customers in Customers
                      where customers.Age > 24
                      orderby customers.Name
                      select new Customer(customers.Name) { Age=custom-
ers.Age};
```

Koodiesimerkki 6. C# 3.0:n olioitten uusi alustaminen LINQ-haussa.

2.1.4 Lisäosametodit

Lisäosametodit antoivat C# 3.0:lle mahdollisuuden laajentaa olemassa olevia tyyppejä uusilla toiminnoilla ilman, että pitää käyttää perintää. Esimerkiksi, jos halusi käyttää Reverse-metodia System.String-tyyppiä kohden, niin piti tehdä erillinen staattinen auttaja metodi. Metodin kutsu ei kuitenkaan ollut kovin helposti luettava, kuten koodiesimerkissä 7 voi nähdä.

Lisäosametodeilla pystyttiin saamaan koodi paljon yksinkertaisemmaksi ja helpommin luettavammaksi. Kuten koodiesimerkistä 7 voi huomata, lisäosametodia kutsutaan kuin se olisi metodin instanssi. Vanha MyHelpers-luokan Reverse-metodi tarvitsee vain pienen muutoksen sen parametreihin, sanan ”this”, että sen voi muuttaa lisäosametodiksi. [3, s. 68-69.]

```
//new extension way
static class Extensions
 {
     public static string Reverse(this string s)
     {
         char[] characters = s.ToCharArray();
         Array.Reverse(characters);
         return new string(characters);
     }
 }
  //old helper way
 static class MyHelpers
 {
     public static string Reverse(string s)
     {
         char[] characters = s.ToCharArray();
         Array.Reverse(characters);
         return new string(characters);
     }
 }
```

```
static class Program
{
    static void Main()
    {
      string name = "Joonas";

        //old way
      string tra = MyHelpers.Reverse(
      name.ToUpper().Trim()).Substring(1);
      Console.WriteLine(tra); // Prints out:ANOOJ

        //New way
      string art = name.ToUpper().Trim().Reverse().Substring(1);
      Console.WriteLine(art); // Prints out:ANOOJ
    }
  }
```

Koodiesimerkki 7. Uusi lisäosametodi ja vanha tapa.

Lisäosametodeja voi myös käyttää LINQ-hauissa. Koodiesimerkissä 8 tehdään LINQ-metodihaku listasta, jossa haetaan sen asiakkaan nimi, jonka ikä on 24. FirstOrDefault valitsee hausta ensimmäisen elementin tai oletusarvon, jos tuloksia on enemmän kuin yksi. Lopuksi voidaan käyttää koodiesimerkissä 7 tehtyä ”Reverse”-lisäosametodia asiakkaan nimen kääntämiseen. LINQ-lisäosametodeja käydään lisää luvussa 5.1.

```
Customer customer1 = new Customer() { Name = "Joonas", Age = 24 };
Customer customer2 = new Customer() { Name = "Sanna", Age = 27 };
List<Customer> customers = new List<Customer> { customer1, customer2
};
string youngCustomerNameInReverse = customers.Where(customer => cus-
tomer.Age == 24).Select(x=>x.Name).FirstOrDefault().Reverse();
//result is "sanooJ"
```

Koodiesimerkki 8. Lisäosametodit LINQ-metodihaussa.

2.1.5 Lambda-lauseet

Lambda-lauseet ovat metodien määritelmiä, joita voi esimerkiksi käyttää matemaattisissa Func-laskuissa tai LINQ-metodihauissa. Lambda-lauseiden tarkoitus oli käyttää niitä delegaattien sijaan. C#-kääntäjä muuttaa lambda-lauseen, joko delegaatiksi tai lausepuuksi. Koodiesimerkissä 9 näkyy, miten jakojäännöslaskun voi tehdä uudella ja vanhalla tavalla Func:in avulla. Kummassakin laskussa koodin yhtä suuri kuin -merkin vasemmalla puolella oleva koodi on samanlainen, mutta oikealla puolella koodi on erilainen. Func-delegaatin kaksi ensimmäistä parametria ovat funktiolle välitettävien arvojen tyypit ja kolmas on palautettavan arvon tyyppi. [2, s. 135.]

```
// Old way
Func<int, int, int> remainder2 = delegate(int a, int b) {
return a % b; };

// New way
Func<int, int, int> remainder3 = (a, b) => a % b;

int result2 = remainder2(5, 3); // result is 2
int result3 = remainder3(5, 3); // result is 2
```

Koodiesimerkki 9. Uusi ja vanha tapa laskea jakojäännös käyttäen Func-delegaattia.

Tarkastellaan lambdan-lausetta tarkemmin. Koodiesimerkissä 10 on lambda-lause, jossa lasketaan jakojäännös, joka on sama kuin koodiesimerkissä 9. Suluissa olevat parametrit on erotettu pilkulla eikä niiden tyyppejä tarvitse ilmoittaa. Seuraavaksi koodissa on => lambda-operaattori ja itse jakojäännöslasku, jossa % on jakojäännös-operaattori.

```
(a, b) => a % b;
```

Koodiesimerkki 10. Lambda-lause, jossa lasketaan jakojäännös.

Lambda-lausetta voi nähdä myös useasti LINQ-metodihauissa. Koodiesimerkissä 11 tehdään haku customers-nimiseen listaan. Haussa ”cityQuery” haetaan ne asiakasoliot, joissa kaupunki on ”London”. Luvussa 5.1 kerrotaan tarkemmin, miten lambda-lauseet toimivat LINQ-hauissa. [3, s. 69-71; 5.]

```
var cityQuery = customers.Where(c => c.City == "London");
```

Koodiesimerkki 11. Lambda-lause LINQ-kyselyssä.

2.1.6 Lausepuut

Lausepuissa on tapa näyttää koodi datana, jota voidaan tutkia runtimessa. Alla on lambda koodiesimerkki 12, josta voi tehdä lausepuun.

```
(int a, int b) => a + b;
```

Koodiesimerkki 12. Lambda-lause, jossa lasketaan summa

Koodiesimerkistä 12 on kaksi mahdollista käännöstä. Yksi näistä käännöksistä on koodiesimerkissä 13, jossa käytetään delegaatteja:

```
Func<int, int, int> add = (a, b) => a + b;
```

Koodiesimerkki 13. Lambda-lauseen delegaatti koodiesimerkistä 12.

```
Expression<Func<int, int, int>> add = (a, b) => a + b;
```

Koodiesimerkki 14. Lausepuu koodiesimerkistä 12.

Toinen tapa on kääntää lambda-lauseen lausepuuksi, jossa kääntäjä käsittelee koodin datana, joka näkyy koodiesimerkissä 14. Ohjelmoijan ei välttämättä tarvitse tietää, mitä lambda-lauseiden taustalla tarkasti tapahtuu, mutta on mielenkiintoista tietää, että on vaihtoehtoisia tapoja lambda-lauseen suorittamiseen. [3, s. 71-73.]

2.1.7 Automaattiset ominaisuudet

Automaattiset ominaisuudet on C# 3.0:n ainoa julkaisu, millä ei ole mitään tekemistä LINQ-hakuarkkitehtuurin kanssa, mutta käydään se läpi kuitenkin, koska se julkaistiin C# 3.0:n kanssa. Tämän ominaisuuden tarkoitus on auttaa ohjelmoijia ominaisuuksien hakemisessa ja asettamisessa laittamalla ne paljon yksinkertaisempaan muotoon.

Koodiesimerkissä 15 näkyy, miten automaattisesti ominaisuudet on toteutettu ennen C# 3.0 -versiota ja sen jälkeen. Koodiesimerkistä huomaa, että AutoImplementedProperties1-luokan tapa tehdä get ja set on paljon yksinkertaisemman näköisempi kuin AutoImplementedProperties2-luokan tapa tehdä get ja set. Kääntäjä syntetisoi piilotetun private-kentän ohjelmoijalle, mikä helpottaa koodin lukemista. [3, s. 73-74.]

```
//New way
class AutoImplementedProperties1
{
    // Auto-Implement Properties for easy get and set
    public int Price { get; set;}
    public void InitializePrice()
    {
        Price = 10;
    }
}

//Old way
class AutoImplementedProperties2
{
    private int _price;
    public int Price
    {
```

```
        get {return _price;}
        set { _price=value;}
    }
    public void InitializePrice()
    {
        Price = 10;
    }
}
```

Koodiesimerkki 15. Uusi ja vanha tapa tehdä luokan ominaisuudet

3 Windows 10

Microsoft julkaisi Windows 10 -käyttöjärjestelmän tietokoneille 29. heinäkuuta 2015, joka oli seuraaja Windows 8 -käyttöjärjestelmälle. Windows 10 Mobile julkaistiin 20. päivä marraskuuta 2015. Windows 10 Mobile julkaistaan vanhoille Lumia-malleille aalloissa. Ensimmäisessä aallossa 13 Lumia-mallia on saanut ilmaisen päivityksen Windows 10 Mobile –versioon [6]. Windows 10 -mobiiliversiosta on julkaistu ohjelmoijille 11 eri emulaattoria, jolla voi kehittää ohjelmia Windows 10 Mobile -alustalle.

Windows 10 -alustaa voi ohjelmoida monella ohjelmointikielellä, kuten JavaScript, C#, Visual Basic, tai C++. Windows 10 -mobiilialustalta löytyy myös mahdollisuus kirjoittaa komponentteja yhdellä ohjelmointikielellä ja käyttää niitä ohjelmassa, joka on tehty toisella ohjelmointikielellä. Windows 10 -ohjelmoija tarvitsee myös kehittäjälisenssin, jotta voi julkaista ohjelmiston. [8.]

3.1 Model-View-ViewModel

Windows 10 -ohjelmia on mahdollista ohjelmoida MVVM-suunnittelumallin mukaisesti. MVVM on lyhenne Model-View-ViewModel-suunnittelumallista, jota Microsoft suosittelee käyttämään eri .NET-alustoilla. MVVM-suunnittelumalli helpottaa kehittäjiä erottamalla eri komponentit toisistaan. MVVM-mallissa on kolme eri mallikäsitettä:

- Näkymässä (View) on ohjelman käyttöliittymä, joka näkyy käyttäjälle ja jonka koodissa ei ole logiikkaa.

- Malli (Model) sisältää ohjelman logiikan ja datan.

- Näkymämalli (ViewModel) tarjoaa näkymälle datan sidonnan ja hoitaa tietokantaoperaatiot.

MVVM-malli on hyödyllinen, koska se auttaa erittäin paljon testaamisessa ja koodin pienentämisessä. Esimerkiksi yhden ohjelman pivot sivut voi jakaa moneen näkymään, joka helpottaa ison ohjelman hallinnassa. Näkymämalli on paljon helpompi testata, koska sillä ei ole omaa käyttöliittymää. Malli on myös helposti testattava, koska se täysin itsenäinen näkymästä ja näkymämallista.

Ohjelmien suunnitteluvaiheessa kannattaa miettiä, mitä projektimallia tulee käyttämään. Pienimmissä projekteissa MVVM-malli ei välttämättä ole niin hyödyllinen kuin isommissa projekteissa. Uuden mallin opiskelu vaatii aina paljon oppimista eikä MVVM-malli ole välttämättä tehokkain pienimmissä projekteissa, joten kannattaa miettiä projektin alussa kannattaako siirtyä MVVM-malliin. MVVM-malli on vain yksi monista suunnittelumalleista miten ohjelma voidaan tehdä. [9.]

Omassa insinöörityössä tein ohjelmani osittain MVVM-mallilla. Näkymämalli hoitaa korttien tietojen sidonnan korttisivulla ja tallennetun kortin sivulla. Malleista löytyy jokainen taulu, eli luokat, ja niiden ominaisuudet. Poikkeuksina MVVM-malliin projektissa ovat tietyt tietokantaoperaatiot ja tallennettujen korttien selaussivu. Olen jakanut yleisimmät tietokantaoperaatiot omille .cs-tiedostoihin, kuten DatabaseDelete.cs, DatabaseUpdate.cs. Tallennettujen korttien selaussivulla ei ole datan sidontaa ollenkaan, joten MVVM-mallia ei noudateta koko insinöörityössä.

Käydään seuraavaksi läpi, miten MVVM-mallia on noudatettu korttisivulla, jossa generoidaan käyttäjälle uusi kortti. Samalla esitellään, miten datan sidonta toimii Windows 10 -alustalla.

```
<TextBlock x:Name="VerbTextBlock"  Text="{x:Bind vm.currentModel.Verb, Mode=OneWay}" />
<TextBlock x:Name="AdjectivesTextBlock" Text="{x:Bind vm.currentModel.Adjective, Mode=OneWay}" />
<TextBlock x:Name="NounTextBlock" Text="{x:Bind vm.currentModel.Noun, Mode=OneWay}" />
<Image x:Name="Image" Source="{x:Bind vm.currentModel.Source, Mode=OneWay}"  />
```

Koodiesimerkki 16. Card.xaml-sivun kolme teksti elementtiä ja yksi kuva elementti.

Koodiesimerkissä 16 on korttisivun kolme TextBlock-elementtiä ja Image-elementti. Luvussa 6 on korttisivusta kuvankaappaus, mistä koodiesimerkki 16 on otettu. Koodiesimerkin 16 jokaisen TextBlock-elementin teksti ja Image-elementin lähde on sidottu näkymämallin currentModel-olion ominaisuuksiin. Sidonta on tehty koodiesimerkissä 16 TextBlock-elementtien Text-ominaisuudessa ja Image-elementin Source-ominaisuudessa. Jotta näkymälle voidaan kertoa, että CardViewModel-luokka on kyseisen kortin näkymämalli, joka hoitaa datan sidonnan, pitää se merkitä Card.xaml.cs näkymä-tiedostossa. Seuraava koodinpätkä hoitaa juuri tämän:

```
private CardViewModel vm;
        public Card()
        {
            vm = new CardViewModel();
            this.DataContext = vm;
```

Koodiesimerkki 17. Määritellään CardViewModel-luokka datan sidonnan lähteeksi.

Koodiesimerkin 17 kohta ”this.DataContext = vm” tarkoittaa sitä, että koodiesimerkin 16 datansidonta otetaan CardViewModel-luokasta. CardViewModel-luokassa luodaan get ja set currentModel-oliolle, eli datan asettaminen ja hakeminen.

Koodiesimerkissä 18 määritellään currentModel-olio CardViewModel-luokassa. Koodiesimerkissä 18 määritelty currentModel-oliossa on nykyisen kortin arvot, johon käyttöliittymän data on sidottu. Olio currentModel on siis CardCurrent-tyyppinen olio, johon koodiesimerkin 16 arvot on sidottu. Aina kun uusi CardCurrent-tyyppinen olio asetetaan currentModel-olioon, niin kutsutaan OnpropertyChanged-metodia. OnpropertyChanged-metodi kertoo käyttöliittymälle, että currentModel-olion arvot on muutettu ja käyttöliittymän pitää päivittää tekstikenttiensä ja kuvansa tiedot vastaamaan currentModel-olion arvoja. Käydään OnpropertyChanged-metodin toiminnot myöhemmin läpi.

```
private CardCurrent _currentModel;

    public CardCurrent currentModel
    {
      get { return _currentModel; }
      set
      {
        _currentModel = value;
        OnpropertyChanged();
      }
    }
```

Koodiesimerkki 18. currentModel-olion määrittely

CardCurrent-luokassa on kolme string-tyyppistä arvoa, josta tekstikenttiin noudetaan arvot. CardCurrent luokassa on myös int-tyyppinen ominaisuus, joka on nimeltään Image. Image-ominaisuus on siis kuvan nimi. Kun Image nimiseen ominaisuuteen asetetaan arvo, niin se automaattisesti asettaa viidenteen Source-ominaisuuteen kuvan polun. Source-ominaisuus on tyypiltänsä string.

3.2 Käyttöliittymään sidottujen arvojen päivittäminen

Kun käyttöliittymän arvot on sidottu dataan, ohjelma tarvitsee tietää, kun tieto on päivittynyt, että se voi päivittää käyttöliittymän arvot dynaamisesti. Tämä on välttämätöntä MVVM-suunnittelumallissa. Jos käyttäjä muokkaa tai lisää oliota kokoelmassa, niin UI-elementti ei välttämättä päivity vastaamaan päivitettyä tietoa. Esimerkiksi ObservableCollection<T>-kokoelma on erittäin hyödyllinen kun tarvitsee ilmoittaa UI-elementeille, että kokoelma on päivittynyt. ObservableCollection<T> on dynaaminen datakokoelma, joka käyttää INotifyCollectionChanged ja INotifyPropertyChanged -rajapintoja. Käydään seuraavaksi läpi, miten IHaveIdeas-ohjelmassa on tehty datansidonta UI-elementteihin.

```
public string Verb
        {
            get
            {
                return this._Verb;
            }
            set
            {
                if (_Verb != value)
                {
                    _Verb = value;
                    NotifyPropertyChanged("Verb");
                }
            }
        }
```

Koodiesimerkki 19. CardCurrent-luokan Verb-ominaisuus, joka kutsuu tiedon asettamisessa NotifyPropertyChanged-metodia.

Koodiesimerkissä 19 esitellään ominaisuus Verb, joka on osa CardCurrent-luokkaa. CardCurrent-luokasta kerrottiin edellisessä luvussa. CardCurrent-luokka on MVVM-

suunnittelumallissa osa mallia. Metodi NotifyPropertyChanged ilmoittaa käyttöliittymäelementeille PropertyChanged-tapahtuman avulla, että ominaisuuden Verb arvo on vaihtunut.

```
        public event PropertyChangedEventHandler PropertyChanged;

         // Used to notify the app that a property has changed.
         private void NotifyPropertyChanged(string propertyName)
         {
             if (PropertyChanged != null)
             {
                 PropertyChanged(this, new PropertyChangedEven-
tArgs(propertyName));
             }
         }
```

Koodiesimerkki 20. NotifyPropertyChanged-metodi kutsuu PropertyChanged-tapahtumaa.

Koodiesimerkissä 20 esitellään metodi NotifyPropertyChanged, jota kutsutaan edellisessä koodiesimerkissä. Metodi kutsuu PropertyChanged-tapahtumaa, joka päivittää sen sidotun käyttöliittymäelementin, joka vastaa NotifyPropertyChanged-metodin parametria ”propertyName”. Jos koodiesimerkin 19 ominaisuuteen Verb asettaisi jonkin arvon, niin propertyName-parametri koodiesimerkissä 20 olisi ”Verb”. Koodiesimerkin 20 käyttöliittymäelementin päivitysmetodia käytetään silloin, kun IHaveIdeas-ohjelmassa käyttäjä vaihtaa sanan tai kuvan Card-sivulla. IHaveIdeas-ohjelma esitellään luvussa 6.

```
public event PropertyChangedEventHandler PropertyChanged;

        public void OnpropertyChanged([CallerMemberName] string prop-
ertyname = null)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEven-
tArgs(propertyname));
            }
        }
```

Koodiesimerkki 21. OnpropertyChanged-metodin implementointi

OnpropertyChanged on metodi, jota kutsuttiin luvussa 3.1 koodiesimerkissä 18. Metodissa oleva CallerMemberName laittaa propertyname-parametrin joko kutsujan metodin nimeksi tai kutsujan muuttujan nimeksi. Luvussa 3.1 tapauksessa se on currentModel eli CardCurrent-tyyppinen olio. OnpropertyChanged-metodi on hyödyllinen, jos halutaan tietää, onko olio korvattu uusilla arvoilla. OnpropertyChanged-metodi ei siis kerro, mikä olion arvoista on muutettu, vaan se kertoo, että currentModel-olioon on asetettu uusi

currentModel-olio uusilla arvoilla. If-lauseessa varmistetaan, että ei päivitetä käyttöliittymäelementtejä, jos olioon ei ole vielä asetettu arvoja. OnpropertyChanged-metodia käytetään IHaveIdeas-ohjelmassa, kun koko kortti vaihdetaan.

Luokat, jotka käyttävät INotifyPropertyChanged-rajapintaa, pystyvät kommunikoimaan tiedonlähteen ja Windows Mobile 10 -käyttöliittymäelementtien kanssa. Tämä on hyödyllistä, koska ohjelmoija voi luoda dynaamisen käyttöliittymän. Dynaamisessa käyttöliittymässä esimerkiksi listat ja tekstikentät päivittyvät automaattisesti vastaamaan uusia arvoja. Luokat, jotka käyttävät INotifyCollectionChanged-rajapintaa, huomauttaa käyttöliittymää kun kokoelmaa on muutettu.[10, s. 34 – 36.]

3.3 Työkalut

Windows 10 -alustan ohjelmoinnissa on kuitenkin rajoitteita työkaluissa. Kehittäjä tarvitsee vähintään Windows 8 -käyttöjärjestelmän ja Visual Studio 2015 [11]. Tein ideageneraattoriohjelmani ja muut koodiesimerkit Visual Studio 2015 Community -versiolla.

Visual Studio on Microsoftin oma ohjelmistokehitysympäristö. Ohjelmointikielinä voi käyttää muun muassa: C#, Visual Basic, C++, F#, Python, Noe.js ja HTML/Javascript.

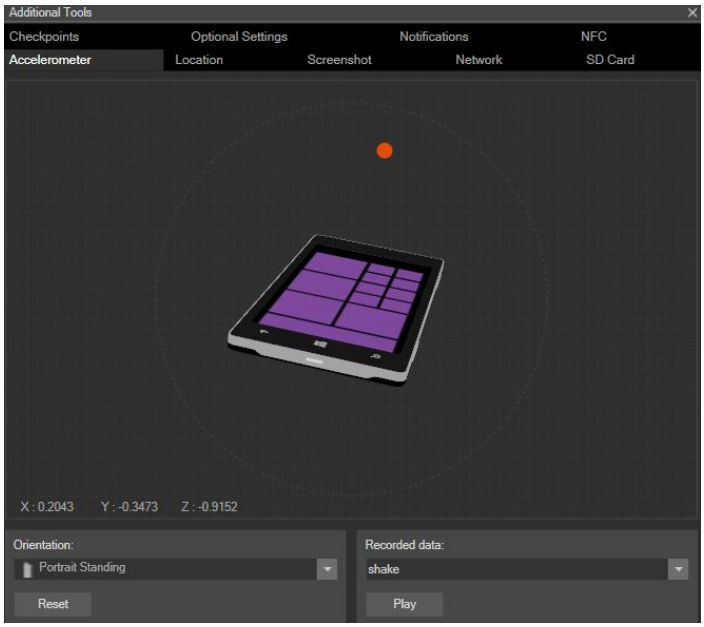
Visual Studio 2015 -versiossa on 11 eri Windows 10 mobiiliemulaattoria, joissa RAM-muisti ja näytön koko vaihtelevat. Windows 10 -emulaattorien lisäksi Visual Studio 2015 -kehitysympäristössä on mahdollista käyttää Android- ja iOS-emulaattoreita. [12.]

3.4 Emulaattori

Windows 10 -ohjelmoija voi testata oman koodin toimivuutta emulaattorilla, ilman että tarvitsee Windows 10 -puhelinta. Insinöörityön aloitusvaiheessa Windows 10 Mobile -käyttöjärjestelmä ei ollut vielä saapunut markkinoille, joten emulaattori on ainoa tapa testata Windows 10 Mobile -ohjelmaa. Tehdessäni insinöörityötä huomasin, että Windows 10 Mobile -emulaattorit ovat erittäin hyödyllisiä ja toimivia työkaluja ohjelman testaamisessa. Kuvassa 1 näkyy Windows 10 Mobile -emulaattori.

Kuva 1. Emulaattorin aloitusnäyttö.

Emulaattorissa on myös erilaisia työkaluja. Kiihtyvyysanturityökalu tarjoaa ohjelmoijalle mahdollisuuden testata, miten ohjelma reagoi puhelimen kääntämiseen X-, Y- ja Z -akseleilla. Asetuksista voi vaihtaa missä asennossa puhelin on ja sillä voi myös testata mitä tapahtuu, kun puhelimeen kohdistaa pientä ravistusta. Kuvassa 2 on kiihtyvyysanturityökalu esiteltynä.



Kuva 2. Emulaattorin kiihtyvyysanturityökalu

Emulaattorilla voi myös simuloida, missä puhelin olisi fyysisesti maailmassa paikkatyökalulla. Paikkatyökalu on erittäin hyödyllinen ohjelman testaamisessa, jos ohjelma käyttää vähänkin paikantamista.

3.5 Universaali projekti

Universaaliprojekti tutustettiin Windows 8 -käyttöjärjestelmän mukana ja joka tunnettiin nimellä Windows Runtime. Universaalin projektin pääydin on se, että kehittäjä voi kehittää ohjelmansa kaikille alustoille yhdellä projektilla. Ohjelmia voi luoda helposti halutuille Windows 10 -alustoille universaalin projektin avulla. Kuvassa 3 voi nähdä eri alustat, joille universaalilla projektilla voi tehdä ohjelmia. [8.]

Kuva 3. Universaalin projektin alustat.

Universaalilla projektilla voi tehdä ohjelmia tietokoneille, tableteille, mobiililaitteille, Xboxille, Microsoft Surface Hubillle, Microsoftin HoloLens:ille ja IoT laitteistoille kuten Raspberry ja Pi 2. [7]

Omassa ohjelmassani keskityin vain Windows Mobile 10 -alustan käyttöön. Windows 10 Universal -projekti on kuitenkin erittäin hyödyllinen tapa saada oma ohjelma monelle Windows 10 -laitteelle ja voisin mahdollisesti tulevaisuudessa laajentaa IHaveIdeas-ohjelmaa työpöydälle.

3.6 XAML

XAML tai “Extensible Application Markup Language” on merkintäkieli, jota käytetään Windows 10 -ohjelmien käyttöliittymän rakentamiseen. XAML-merkintäkielellä voi luoda käyttöliittymäkomponentteja ja hallinnoida niiden tapahtumia erillisessä tiedostossa. Kun Visual Studiossa tekee uuden XAML-sivun, esimerkiksi oman ohjelmani About.xaml, niin Visual Studio automaattisesti luo XAML-tiedoston taakse About.xaml.cs-kooditiedoston, jossa voidaan käsitellä käyttöliittymän tapahtumia ja muita toimintoja.

Koodiesimerkissä 24 on pääsivun painike, josta voi generoida uusia kortteja. Määritellään ensin, mikä komponentti on, eli painike. ”x:Name” -ominaisuus määrittelee nimensä mukaan painikkeen nimen. Nimi ei ole pakollinen, mutta on välttämätön, jos esimerkiksi

haluaa muokata painikeen ominaisuuksia kooditiedostosta ohjelman ajon aikana. Etuliitteen ”x:” tarkoittaa nimiavaruuden konstruktoria, joista kolme yleisintä on:

- x:Key, joka asettaa käyttäjän määritellyn uniikin avaimen resurssille ResourceDictionary. Koodiesimerkissä 22 annetaan esimerkki sen määrittelystä ja käytöstä painikkeessa. Painikkeen tekstiksi tulee alapuolella olevassa koodiesimerkissä ”Hello, world!”.

```
//In a xaml file
<Page.Resources>
    <x:String x:Key="strHelloWorld">Hello, world!</x:String>
</Page.Resources>

//Resource used in a button
 <Button Content="{StaticResource strHelloWorld}"  other code.../>
```

Koodiesimerkki 22. X:Key määrittely XAML-sivun resursseissa ja sen käyttö.

- X:Class määrittelee koodin nimiavaruuden ja sen luokan nimen, joka toimii XAML-sivun takana. Koodiesimerkin 23 tapauksessa tämä on Card-niminen luokka, joka käsittelee Card.xaml-sivun toiminnot. Card-luokka sijaitsee IHaveIdeas.Card-nimiavaruuden alla.

```
x:Class="IHaveIdeas.Card.Card”
```

Koodiesimerkki 23. IHaveIdeas Card.xaml-sivun x:Class määrittely.

- x:Name määrittelee elementin nimen. Nimeä voi käyttää sivun takan olevassa kooditiedostossa ominaisuuksien, kuten esimerkiksi paikan, värin, koon tai tekstin vaihtamiseen.

```
<Button x:Name="Pick_a_Card_Button" Content="I have an idea!" Horizon-talAlign-
ment="Left" Margin="91,478,0,0" VerticalAlignment="Top" Width="305"
Click="Pick_a_Card_Button_Click" BorderThickness="2"/>
```

Koodiesimerkki 24. Ideageneraattorin pääsivun painikkeen määrittely.

Koodiesimerkissä 24 on Content-ominaisuus, joka määrittelee painikkeen tekstin. Tekstiä voi muokata ohjelman suorituksen aikana taustalla olevan koodin kautta, käyttäen painikkeen nimeä. Omassa ohjelmassani kaikki Content-tekstien määrittelyt on hoidettu

dynaamisesti, joka esiteltiin luvussa 3.2. Koodiesimerkissä 25 esitellään, miten painikkeen teksti voidaan muokata taustalla olevassa kooditiedossa.

```
Pick_a_Card_Button.Content = "Take a card already!";
```

Koodiesimerkki 25. Pick_a_Card_Button -painikkeen tekstinvaihto.

Seuraavat ominaisuudet koodiesimerkissä 24 ovat paikan-, koon- ja reunanpaksuudenmäärittelyitä. Click-ominaisuus määrittelee kooditiedostossa olevan metodin nimen, jota kutsutaan, jos painiketta painetaan. [13.]

4 LINQ-ohjelmointikieleen integroitu hakuarkkitehtuuri

Ennen C# 3.0 -version julkaisua monilla ohjelmoijilla oli iso ongelma päästä käsiksi tietoon, joka voi sijaita monella eri tallennusalustalla. Yhden ohjelman tietokanta saattoi olla relaatiotietokannassa, XML-tiedostossa tai muussa kohteessa. Ongelma oli, että jokaisella tietolähteellä oli oma API, jotka olivat erilaisia toisistaan ja joiden opetteleminen lisäsi kehittäjien oppimiskäyrää merkittävästi. Ratkaisu tähän ongelmaan oli ”Project Clarity”, mikä uudelleennimettiin Lanquage Integrated Query -nimiseksi ja jonka lyhenne on LINQ. LINQ integroitiin C# 3.0 -versiossa ohjelmointikieleen, ja se mahdollistaa haut jokaiseen tietolähteeseen, joissa on LINQ-tuki.

LINQ-esimerkkinä käytetään asiakaslistaa, joka on tallennettu olioina listakokoelmaan. Seuraavana on koodiesimerkki 26, jossa on edellä mainittu lista.

```
var Customers = new List<Customer>{
                new Customer { Name="Joonas Virtanen" ,Age=23},
                new Customer { Name="Petrus Laine" ,Age=33},
                new Customer { Name="Juha Kokkonen" ,Age=21},
                new Customer { Name="Juha Jäntti" ,Age=25},
                new Customer { Name="Antti Valkeinen" ,Age=29},
                new Customer { Name="Pekko Lainiala" ,Age=28},
                new Customer { Name="Teemu Hynninen" ,Age=35},
                new Customer { Name="Kristian Vättö" ,Age=30}
            };
```

Koodiesimerkki 26. Asiakasolioita listakokoelmassa.

Ajatellaan, että ohjelmassa pitäisi filtteröidä ja uudelleen järjestää tieto toiseen kokoelmaan. Esitellään ensin, miten tieto haetaan kahdella eri tavalla ja lopuksi käydään läpi

LINQ-esimerkki. Seuraavaksi tehdään toinen kokoelma, missä pitäisi olla nuoret asiakkaat, jonka ikä olisi alle 24.

```
var yCustomers = new List<Customer>();
foreach (var customer in Customers)
 {
     if (customer.Age < 24)
     {
         yCustomers.Add(customer);
     }
 }
```

Koodiesimerkki 27. Alle 24-vuotiaiden hakeminen käyttäen if-lausetta foreach-lauseen sisällä.

Koodiesimerkin 27 koodi on melko helppo ymmärtää. Filteröinti on erittäin yksinkertainen, mutta se tarvitsee aika monta riviä koodia sen suorittamiseen. Mitäs jos halutaan vaihtaa tallennuskohdetta? Halutaan järjestää asiakkaat, jotka ovat alle 24 ja yli 24, vaikkapa Dictonary-kokoelmaan.

```
var ageGroups = new Dictionary<int, List<Customer>>();
foreach (var customer in Customers)
{
    int group = (customer.Age/24);
    List<Customer> customerInGroup;
    if (!ageGroups.TryGetValue(group, out customerInGroup))
    {
        ageGroups[group] = customerInGroup = new List<Customer>();
    }
        customerInGroup.Add(customer);
}
```

Koodiesimerkki 28. Alle 24-vuotiaiden hakeminen Dictonary-kokoelmaan.

Koodiesimerkki 28 ei ole niin helposti luettavana kuin koodiesimerkki 27. Koodin haittapuolina on koodin vaikealukuinen rakenne ja koodin rakentaminen kuluttaa turhan paljon aikaa pieneen tehtävään. Koodiesimerkissä 28 ageGroups-avaimelle nolla asetetaan kaikki alle 24-vuotiaat asiakkaat ja avaimelle 1 kaikki 24-vuotiaat ja vanhemmat asiakkaat.

Tehdään seuraavaksi SQL-kysely relaatiotietokannasta käyttäen apuna Microsoftin Northwind-tietokantaa. Koodiesimerkissä 29 haetaan kaikki tuotteet, joiden yksikköhinta on pienempi kuin 15.

```
SELECT * FROM Products WHERE UnitPrice < 15
```

Koodiesimerkki 29. Yksinkertainen SQL-kysely.

Edellä esitelty haku on erittäin yksinkertainen SQL-kysely, mutta tehdään sen suorittaminen .Net-alustalla käyttäen C#-ohjelmointikieltä. Vaikka itse SQL-haku näyttää olevan yksinkertainen, sen suorittaminen tarvitsee paljon ylimääräistä koodia. Koodiesimerkissä 30 esitetään, miten tämä tehtäisiin käytännössä. Koodiesimerkin 30 koodin tekemiseen menee turhan paljon aikaa, joka johtaa myös siihen, että koodi on virhealttista.

```
var products = new List<Product>();
using(var conn = new SqlConnection(connectionString)){
    string sql = "SELECT * FROM Products WHERE UnitPrice < 15";
    using(var cmd = new SqlCommand(sql,conn)){
        conn.Open();
        using (var reader = cmd.ExecuteReader()){
            while(reader.Read()){
                string name = (string)reader["ProductName"];
                decimal price (decimal)reader["UnitPrice"];
                products.Add(new Product {Name = name, Price =
price});
            }
        }
    }
}
```

Koodiesimerkki 30. SQL-haku .Net-alustalla C#-ohjelmointikielellä.

Nyt on käyty kaksi eri tapaa tehdä kysely tietolähteeseen. Foreach-lausetta ja ehtolauseita käyttäen koodi muuttuu helposti monimutkaiseksi ja vaikeasti luettavaksi, jos siihen lisätään lisätoimintoja. SQL-hakujen kanssa tulee paljon ylimääräistä koodia, jos haluaa tehdä yksinkertaisenkin haun, mikä ei ole suositeltavaa koodin selkeyden näkökulmasta. Kummassakin tiedonhakumenetelmässä on ongelmia koodin luettavuudessa, ja ne ovat kummatkin helposti alttiita virheille.

Tässä kohdassa LINQ-hakuarkkitehtuuri tulee erittäin käteväksi. Koodiesimerkissä 31 tehdään haku Customers-kokoelmaan, jossa haetaan taas kaikki asiakkaat, joiden ikä on alle 24 ja järjestetään ne iän mukaan. Ensimmäinen haku alla olevassa koodiesimerkissä on deklaratiivinen LINQ-haku ja jälkimmäinen on LINQ-metodihaku. Luvussa 5.1 käydään läpi, mitä eroja näillä kahdella haulla on ja esitellään kummatkin haut tarkem-

min. Koodiesimerkki 31 on erittäin helppolukuinen ja lisätoimintojen lisääminen on helppoa, eikä vaadi edellisten esimerkkikoodien tapaista runsasta vaikealukuista koodia. [3, s. 914 – 917.]

```
//LINQ query syntax
var yCustomers = from customer in Customers
                                where customer.Age < 24
                                orderby customer.Age
                                select customer;
//LINQ extension method syntax
var yCustomers  = Customers.Where(x => x.Age > 24).OrderBy(y =>
y.Name);
```

Koodiesimerkki 31. Kaksi LINQ-hakua, jossa on sama lopputulos.

5 Hakujen toiminta LINQ:ssä

5.1 Deklaratiivisien ja lisäosametodien syntaksi LINQ:ssä

Seuraavaksi esitellään, miten LINQ-deklaratiivisten ja lisäosametodien hakusyntaksi toimii. LINQ deklaratiiviset haut pitää ensin kääntää metodikutsuiksi .NET CLR:lle, kun kääntäjä kääntää koodin. Metodikutsut ovat standardihakuoperaattoreita, jotka voivat olla esimerkiksi Select, Where, Average, Join, Max, OrderBy ja GroupBy. LINQ-haut voi tehdä myös suoraan metodihakuilla, joka esitellään seuraavaksi.

```
//Query syntax:
IEnumerable<string> yCustomers2 = from customer in Customers
                                    where customer.Age < 24
                                    orderby customer.Age
                                    select customer.Name;
//Method syntax:
IEnumerable<string> yCustomers3 = Customers.Where(customer => cus-
tomer.Age < 24).OrderBy(customer => customer.Age).Select(customer =>
customer.Name);
```

Koodiesimerkki 32. Deklaratiivinen haku ja metodihaku.

Koodiesimerkissä 32 on tehty sama haku deklaratiivisella ja metodityyppisellä LINQ-haulla. Kummankin haun tulos on täysin sama. Haku tehdään kokoelmasta, mikä on tehty koodiesimerkissä 26. Kummassakin haussa lopputulos on samaa tyyppiä, eli IEnumerable<string>.

Metodihaut toimivat lambda-lauseilla. Lambda-lauseet esiteltiin ensin luvussa 2.1.5, mutta käydään ne tarkemmin läpi. Koodiesimerkin 32 ensimmäisen lambda-operaattorin ”=>” vasemmalla puolella oleva ”customer” on sisään tuleva arvo, joka on sama kuin deklaratiivisten haun ”customer”. Kääntäjä voi päätellä mitä tyyppiä customer on, koska se tietää ”Customers” on geneerinen IEnumerable<T>-tyyppi. Tässä tapauksessa customer on lambda-lauseessa Customer-tyyppinen.

Seuraavaksi on lambda operaattori => Where-standardihakuoperaattorissa. Kohdassa ”`customer.Age < 24`” määritellään filtteri haulle, eli haetaan ne Customer-oliot kokoelmasta, joiden ikä on alle 24. Toisessa standardihakuoperaattorissa OrderBy järjestellään haku iän mukaan. OrderBy:n sisäinen lambda-lause on samanlainen rakenteeltaan kuin Where-standardihakuoperaattorin lambda-lause mutta ilman ehtoa. Lopuksi on standardihakuoperaattori Select. Select-standardihakuoperaattorilla voi valita mitä halutaan palauttaa hausta. Koodiesimerkissä 32 Select-standardihakuoperaattorilla palautetaan yCustomers3 kokoelma, jossa on asiakkaiden nimiä, jotka on järjestetty nimensä mukaan ja joiden ikä on alle 24 vuotta.

Deklaratiivinen LINQ-haku on helppo ymmärtää, jos SQL on ennestään tuttu. Koodiesimerkin 32 haussa määritellään ensin customer, joka on tyypiltään Customer kokoelmasta Customers, joka määritellään operaatiolla in. Seuraavaksi luodaan filtteröinti operaatiolla where. Haetaan ne asiakkaat, joiden ikä on alle 24. Haun palautettavat arvot järjestetään iän mukaan operaatiolla orderby. Hausta palautetaan asiakkaan string-tyyppiset nimet kohdassa ”select customer.Name”. [14.]

5.2 IEnumerable<T> rajapinta

LINQ:ssä on kaksi datan perusyksikköä, jotka ovat sekvenssit ja elementit. Sekvenssi on mikä tahansa olio, joka käyttää IEnumerable<T>-rajapintaa. Elementti on jokainen alkio sekvenssissä. Seuraavana on koodiesimerkki 33, jossa esitellään tämä käytännössä. Int tyyppinen kokoelma ”wages” on sekvenssi ja elementti on jokainen luku sekvenssissä.

```
int[] wages =  { 2500, 2550, 2750 };
```

Koodiesimerkki 33. Segmentti wages, jossa on kolme elementtiä

Koodiesimerkkiä 33 kutsutaan paikalliseksi sekvenssiksi, koska se sisältää kokoelman paikallisia olioita muistissa. Hakuoperaattori on metodi, joka muuttaa sekvenssin hakuoperaatioilla. Hakuoperaatio ottaa sisään sekvenssin ja päästää ulos muutetun sekvenssin. Yksinkertaisin haku sisältää yhden sisääntulevan sekvenssin ja yhden operaattorin. Luokassa Enumerable, joka on System.Linq-nimiavaruuden sisällä, on noin 40 hakuoperaattoria. Näitä kutsutaan standardeiksi hakuoperaattoreiksi. Hakuoperaattorit on kaikki toteutettu staattisilla lisäosametodeilla, joita käytiin luvussa 2.1.5 ja edellisessä luvussa 5.1.

Jotta voidaan ymmärtää IEnumerable-toimintaa, niin käydään edellisen luvussa koodiesimerkki askelein. Seuraavana on koodiesimerkki 34, jossa finalQuery:ssa tulos on sama kuin koodiesimerkissä 32.

```
IEnumerable<Customer> filtered = Customers.Where(customer => cus-
tomer.Age < 24);
IEnumerable<Customer> sorted = filtered.OrderBy(customer => cus-
tomer.Age);
IEnumerable<string> finalQuery = sorted.Select(customer => cus-
tomer.Name);
```

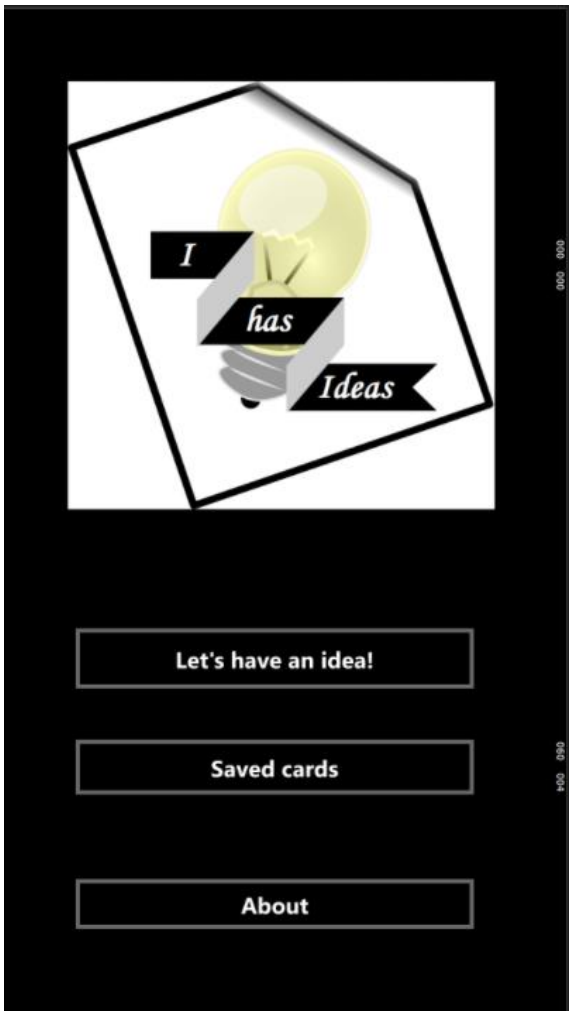
Koodiesimerkki 34. Koodiesimerkin 32 lisäosametodihaku tehty kolmessa osassa.

Koodiesimerkissä 34 voi huomata, että IEnumerable filtteröinti on kuin liukuhihna. Liukuhihnalle laitetaan alussa koko Customers-kokoelma. Ensimmäinen filtteröinti ottaa liukuhihnalta ne asiakkaat, joiden ikä on yli 23 ja liukuhihna vie kokoelman toiselle filtterille. Haussa sorted, liukuhihnalle saapuvat ne asiakasoliot, jotka pääsivät ensimmäisen filteröinnin läpi. Kohdassa sorted laitetaan liukuhihnalta tulevat asiakasoliot järjestykseen nimen mukaan. Liukuhihnan lopussa tehdään finalQuery, jossa otetaan liukuhihnalla olevien olioiden ominaisuuksista pois kaikki paitsi nimi. Liukuhihnalta tulee siis vain asiakkaiden nimet ulos, kuten koodiesimerkin 32 hauissa. [2, s. 319–323.]

6 IhaveIdeas-ohjelma

Insinöörityötä varten olen tehnyt Windows Mobile 10 -ohjelman, joka on nimeltään IHaveIdeas. Päätavoite ohjelman teossa oli oppia Windows Mobile 10 -ohjelmoinnista ja SQLitestä käyttäen LINQ-hakuarkkitehtuuria. Aloitin ohjelman teon 2014 syksyllä osana Mobile Application Design -kurssia Windows Phone 8.1 Silverlight -alustalle käyttäen LINQ To SQL -tietokantaa. Insinöörityötä tehdessäni julkaistiin Windows 10 ja päätinkin

tehdä ohjelmani Windows Mobile 10 -alustalle. Aloitin ohjelman teon ilman mitään esikokemusta mobiilialustoihin. IHaveIdeas-ohjelman lähdekoodi on GitHubissa, jonka linkki löytyy liitteistä. Ohjelman pääsivu näkyy kuvassa 4.



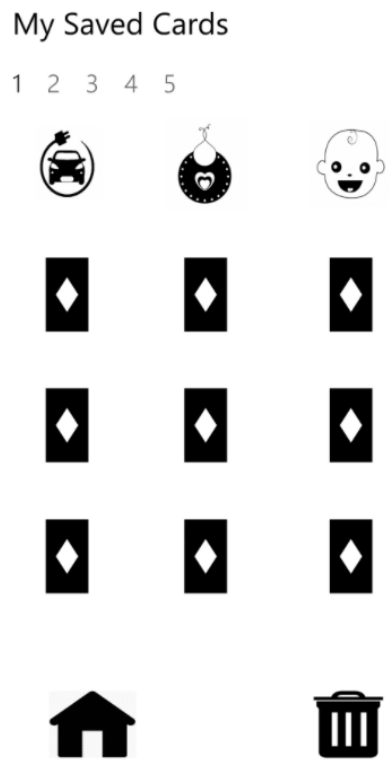
Kuva 4. IHaveIdeas-ohjelman pääsivu emulaattorissa.

IHaveIdeas on pienimuotoinen ideageneraattoriohjelma. Ohjelma on suunnattu niille ihmisille, joilla on ongelmia keksiä ideoita. Ideageneraattori voi esimerkiksi auttaa koululaista esseen kirjoittamisessa tai auttaa taiteilijaa inspiraation saamisessa. Ohjelma generoi käyttäjälle kortin, jossa on kuva ja kolme sanaa ja joista käyttäjän pitäisi saada inspiraatio. Kolme sanaa, jotka ideageneraattori generoi, ovat verbi, adjektiivi ja substantiivi.

Kuva 5. Esimerkki kortista.

Kuvassa 5 on esimerkki kortista. Jos käyttäjä ei pidä kuvasta tai yhdestä sanasta, niin hän voi vaihtaa sen pyyhkäisemällä sen oikeaan tai vasempaan reunaan, joka generoi sitten uuden sanan tai kuvan. Jos käyttäjä pitää saamastaan kortista, hän voi tallentaa sen korttipakkaan.

Kuvassa 6 on tallennettujen korttien pivotsivu, jossa on kaksi korttia tallennettu korttipakkaan 1. Korttien kuvat toimivat pikakuvakkeina tallennetuille korteille. Pivotsivussa on viisi korttipakkaa, johon käyttäjä voi tallentaa korttinsa. Jokaisen korttipakan alapuolella ovat painikkeet pääsivulle ja koko korttipakan poistamisen painike.



Kuva 6. Tallennetut kortit -pivotsivu.

7 SQLite tietokanta

SQLite on avoimen lähdekoodin suosittu relaatiotietokantajärjestelmä. D. Richard Hipp kehitti SQLiten, kun Hipp työskenteli Yhdysvallan merivoimissa vuonna 2000. Yhdysvaltojen merivoimilla oli ohjattavia ohjuksia, jotka käyttivät Informix-tietokantaa. Informix-tietokanta oli kuitenkin liian tehokas järjestelmä merivoimille, koska se esimerkiksi vaati tietokanta-ammattilaiselta noin päivän työpanoksen asentaa tai päivittää tietokanta. SQLite oli täydellinen ratkaisu tähän, koska ei vaatinut minkäänlaista asentamista.

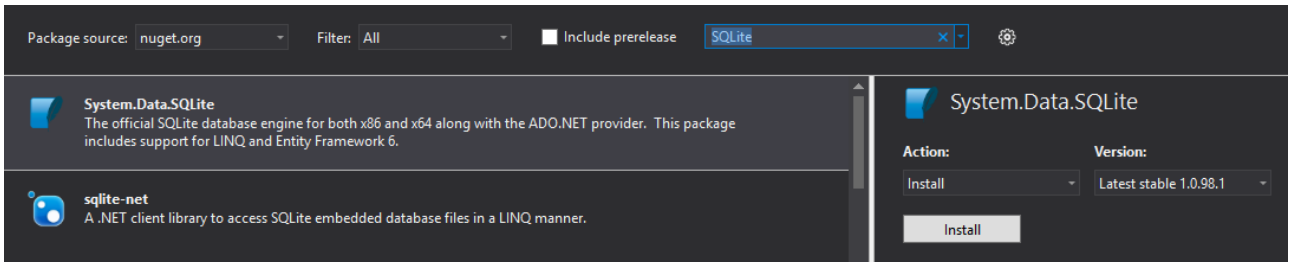
SQLite ei ole samanlainen kuin esimerkiksi Microsoft SQL Server -tyyppinen tietokanta, joka toimii palvelimella, vaan se on ohjelman sisäinen tietokanta. Sisäisellä tietokannalla tarkoitetaan sitä, että se on osa itse ohjelmaa. Tietokanta luodaan ohjelman sisällä, eikä se ota yhteyttä johonkin ulkomailla olevaan serveriin, kuten Microsoft SQL Serverissä voi tehdä. SQLite-tietokannan voidaan luoda kun ohjelma ajetaan ensimmäistä kertaa tai lataa tietokannan tiedostosta, jossa on tietokanta valmiiksi rakennettu.

SQLite on rakennettu siirrettäväksi tietokantamoottoriksi. SQLite tukee Windowsia, Linuxia, BSB, Mac OS X, sulautettuja alustoja kuten Symbian, Windows CE, QNX, Palm OS, ja VxWorks. Lisäksi SQLite tukee myös muutamaa kaupallista Unix-käyttöjärjestelmää. SQLite-tietokantatiedostot ovat myös binäärisesti yhteensopivia kaikilla tuetuilla käyttöjärjestelmillä. Kehittäjä voi luoda SQLite-tietokannan vaikkapa Sun SPARC -työasemalla ja käyttää sitä Windows-koneella ilman mitään muutoksia. SQLite-tietokannat pystyvät pitämään jopa 2 teratavua dataa. [15, s. 3-4]

Insinöörityössä haluttiin käyttää LINQ-hakuarkkitehtuuria SQLite-tietokantakyselyissä IHaveIdeas-ohjelmassa. SQLite.Net PCL on kääre, joka mahdollistaa juuri tämän. SQLite.Net PCL on kopio toisesta kääreestä, mikä on nimeltään SQLite-net. SQLite.Net PCL on tarkoitettu parantaa SQLite-net koodin laatua käyttäen moderneja teknologioita, kuten PCL eli siirrettäviä luokkakirjastoja [16].

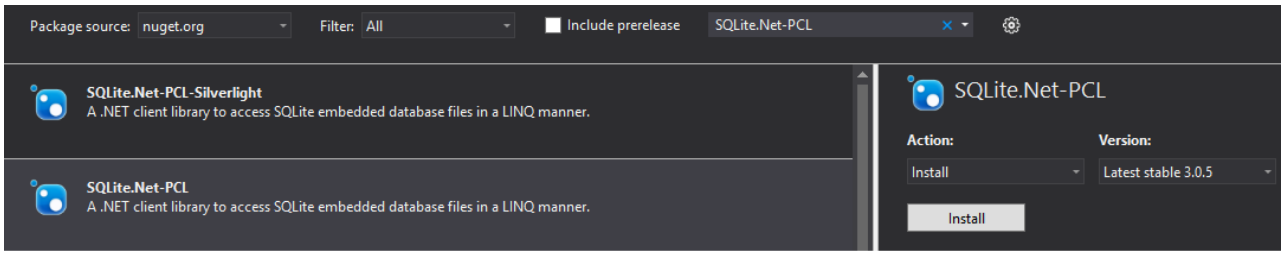
7.1 Asennus

SQLite.Net PCL -asennus on erittäin suoraviivaista. Kaikki tarvittavat osat saa helposti ladattua Visual Studion NuGet -paketteina. SQLiten voi hakea hakusanoilla ”SQLite”, kuten kuvassa 7 näkyy.



Kuva 7. SQLite-NuGet-paketti.

Seuraavaksi haetaan SQLite.Net PCL hakusanalla ”SQLite.Net PCL”, kuten kuvassa 8. Lopuksi pitää vielä muistaa lisätä SQLite projektiin refereissä. SQLite löytyy nimellä ”SQLite for Universal App Platform” Universal Windows ja sen alla olevan Extension-kohdan alta. Näiden toimintojen jälkeen voi alkaa tehdä SQLite-tietokantaa.



Kuva 8. SQLite.Net PCL-NuGet-paketti.

7.2 Tietokannan luonti

Jotta SQLite-tietokannan voi tehdä koodiesimerkin 35 mukaisesti, niin koodiin pitää lisätä alla olevat nimiavaruudet:

```
using SQLite.Net;
using SQLite.Net.Platform.WinRT;
```

Koodiesimerkki 35. Nimiavaruudet, joita tarvitsee tietokannan luomiseen.

```
public static string path = Path.Combine(ApplicationData.Current.LocalFolder.Path, "SQLit-
edb.sqlite");

using (SQLiteConnection conn = new SQLiteConnection(new SQLitePlatformWinRT(), path))
        {
          conn.CreateTable<Adjectives>();
          conn.CreateTable<Verbs>();
          conn.CreateTable<Nouns>();
          conn.CreateTable<SavedCards>();
          conn.CreateTable<GeneratedCards>();
          conn.CreateTable<Images>();
        }
```

Koodiesimerkki 36. IHaveIdeas-ohjelman tietokannan luonti ja taulujen määrittely.

Yläpuolella on koodiesimerkki 36, jossa esitellään IHaveIdeas-ohjelman SQLiten-tietokannan luominen. Tietokanta luodaan ”using” kohdassa, missä määritellään tietokannan alusta ja sijainti. Sijainti on määritelty koodiesimerkin 36 alussa, joka on nimeltä path. Muuttujassa path on määritelty polku tietokantaan ja sen nimi.

Seuraavaksi on taulujen määritys. Taulut määritellään SQLiteConnection-luokan CreateTable<T>()-metodilla. Luokat, jotka on määritelty tauluiksi, ovat yksinkertaisia luokkia. Koodiesimerkissä 37 on Adjectives-luokan määrittely, jota käytetään tauluna tietokan-

nassa. Ominaisuus Id on määritelty attribuutilla ”[PrimaryKey, AutoIncrement]” pääavaimeksi ja automaattisesti kasvavaksi arvoksi. AdjectiveWord-ominaisuus ei tarvitse mitään attribuutteja. AdjectiveWord-ominaisuuteen tallennetaan itse adjektiivi sana.

```
public class Adjectives
  {
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    public string AdjectiveWord { get; set; }
  }
```

Koodiesimerkki 37. Adjectives-luokan määrittely.

7.3 Tietokantaoperaatiot

Luvussa 5.2 käytiin läpi LINQ-hakuarkkitehtuurin toimintaa tarkemmin IEnumerable-rajapinnassa. Tämän luvun alaotsikoissa esitellään, miten tehdä yksinkertaisia SQLite.Net PCL-tietokantaoperaatioita. Koodiesimerkissä 38 esitellään SQLite-tietokantayhteyden tekeminen, joka nähtiin jo edellisen kappaleen koodiesimerkissä 36. Kaikista koodiesimerkeistä paitsi haku- ja päivitysoperaatioista tämä on otettu pois, jotta esimerkit olisivat yksinkertaisimpia. Tietokantaoperaatioita tehdessä pitää aina avata yhteys tietokantaan koodiesimerkin 38 tapaisella koodilla. Yhteyden luomisessa määritellään tietokannan alusta ja tietokantatiedoston sijainti.

```
using (SQLiteConnection db = new SQLiteConnection(new SQLitePlatformWinRT(), path))
        {
        //Code
        }
```

Koodiesimerkki 38. SQLite-tietokannan yhteyden avaaminen.

7.3.1 Lisäysoperaatio

Koodiesimerkissä 39 näytetään, miten lisäysoperaatio Verbs-tauluun toimii. Seuraava koodiesimerkki on heti tietokannan luomisen jälkeen. Ensin luetaan Verbs-niminen tekstitiedosto, jossa on noin 600 verbiä. Verbit luetaan rivi riviltä, kunnes tiedosto loppuu. Verbi sana asetetaan verbs-olioon ja lisätään tietokantaan yksinkertaisella Insert-metodilla, jossa parametrina on verb-olio.

```
StorageFile file = await StorageFile.GetFileFromApplicationUriAsync(new Uri(@"ms-appx:///Data-
baseWords/Verbs.txt"));
            using (StreamReader Read = new StreamReader(await file.OpenStreamForRea-
dAsync()))
            {
              while (!Read.EndOfStream)
              {
                Verbs verb = new Verbs { VerbWord = (Convert.ToString(Read.ReadLine())) };
                db.Insert(verb);
              }
            }
```

Koodiesimerkki 39. Tekstitiedoston datan lisääminen verbi-tauluun.

7.3.2 Haku- ja päivitysoperaatio

Koodiesimerkissä 40 näytetään, miten hakuoperaatio ja päivitysoperaatio toimivat SQLite.Net PCL -kääressä. Koodiesimerkkiä 40 kutsutaan, kun tallennetuissa korteissa yksi kortti poistetaan pakasta. Koodiesimerkkiä 40 päivitysoperaatio on tarpeellinen, jotta kortit pysyvät oikeilla paikoillaan tallennettujen korttien pakassa.

```
/// <summary>
/// Reorders saved cards pivot page cards after delete.
/// </summary>
/// <param name="pack">The pack number where a card was
removed</param>
/// <param name="cardPosition">Position number of the card where card
was removed</param>
        public void ReOrderSavedCards(int pack, int cardPosition)
        {
          int newCardPosition = cardPosition;
           using (SQLite.Net.SQLiteConnection db = new
           SQLite.Net.SQLiteConnection(new
           SQLite.Net.Platform.WinRT.SQLitePlatformWinRT(), App.path))
            {
                IEnumerable<SavedCards> reOrderQuery =
                db.Table<SavedCards>()
               .Where(card => card.PackNumber == pack &&
               card.CardNumber >  cardPosition);
                foreach (SavedCards savedCard in reOrderQuery)
                {
                    savedCard.CardNumber = newCardPosition;
                    db.Update(savedCard);
                    newCardPosition++;
                }
            }
        }
```

Koodiesimerkki 40. Uudelleen järjestellään tallennettujen korttien pakka poisto-operaation jälkeen.

Koodiesimerkissä 40 tehdään IEnumerable-haku reOrderQuery. Haut ovat taulupohjaisia ja alkavat koodipätkällä ”db.Table<SavedCards>()”, josta tehdään LINQ-haku. Table<T> on SQLite-tietokannan tauluhakumetodi, jossa T, eli tyyppi, merkin paikalle asetetaan taulun luokka josta hakuoperaatio halutaan tehdä. Taulusta tehdään LINQ-haku, jossa haetaan vain sen pakan kortit, josta on tehty poisto-operaatio, ja ne kortit, joiden paikkanumero on suurempi kuin poistetun kortin.

Päivitysoperaatio tehdään foreach-lauseen sisällä. Päivitettävän kortin arvoon CardNumber asetetaan uusi arvo newCardPosition, joka määrittelee kortin uuden paikan pakassa. Lopuksi kortti päivitetään metodilla Update, jonka parametriksi tulee päivitettävä kortti.

7.3.3 Poisto-operaatio

Koodiesimerkissä 41 tehdään haku tallennettuihin kortteihin, missä haetaan yhden tallennetun pakan kaikki kortit. Foreach-lauseessa tehdään poisto-operaatio Delete-metodilla, jonka parametrissa on tallennettu kortti, joka halutaan poistaa tietokannasta. Hakuoperaatio on melkein sama kuin koodiesimerkissä 40, mutta ilman CardNumber-ehtoa. Koodiesimerkkiä 41 käytetään korttien selaussivulla, kun halutaan poistaa tallennettujen korttien pakka.

```
IEnumerable<SavedCards> packQuery = db.Table<SavedCards>().Where(card =>
card.PackNumber == pack);
foreach (SavedCards savedCard in packQuery)
  {
    db.Delete(savedCard);
  }
```

Koodiesimerkki 41. Haetaan kaikki pakan kortit ja poistetaan ne tietokannasta.

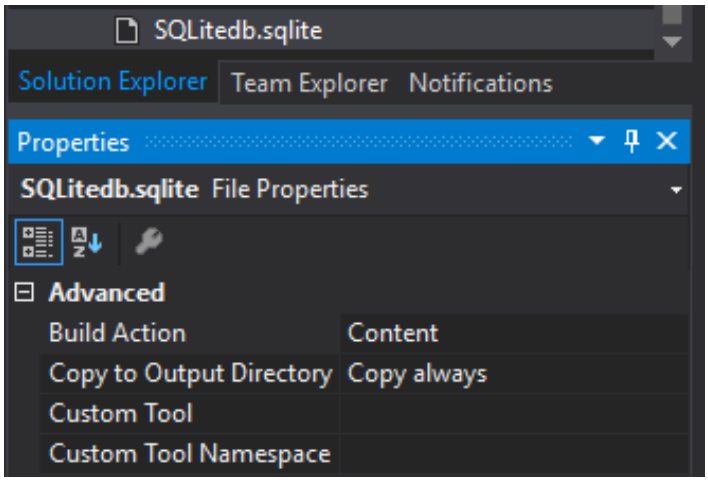
7.4 Eristetty tietokanta ja sen käyttöönotto

Insinöörityön ideageneraattoriohjelman olen tehnyt kahdella eri tavalla. Toinen on paikallinen SQLite-tietokanta ja toinen on eristetty SQLite-tietokanta. Eristetyssä tietokantaversiossa tietokanta on ”SQLitedb.sqlite” nimisessä tiedostossa. Tiedostossa on tietokanta valmiina, eikä tietokantaan tarvitse lisätä kuvia ja sanoja tai määritellä tauluja uu-

destaan. Eristetyn tietokannan käsittely on kuitenkin samanlainen kuin paikallisen tietokannan. Paikallinen tietokanta voidaan muuttaa eristyneeksi tietokannaksi käyttäen muun muassa Windows Phone Power Tools -työkalua.

Windows Phone Power Tools -työkalulla voidaan hakea .sqlite-tiedosto, joka syntyy paikallisessa tietokannassa. Ohjelma pitää käynnistää kerran ja sitten hakea .sqlite-tiedosto Windows Phone Power Tools -työkalulla. SQLite-tietokanta tiedosto pitää luomisen jälkeen kopioida Visual Studion -projektiin. Seuraavaksi käydään läpi, mitä toimintoja tarvitsee tehdä, jotta eristettyä tietokantaa voi käyttää.

Tietokantatiedostossa on valmiina kaikki taulut ja niihin sidotut sanat ja kuvat. Jotta tietokantaa voidaan käyttää Windows 10 Mobile -ohjelmassa, niin tietokanta-tiedosto pitää kopioida ohjelman paikalliseen juurikansioon. Windows 10 -projektissa pitää muistaa laittaa .sqlite-tiedoston ominaisuuksista ”Build Action” -kohtaan arvo Content, kuten kuvassa 9 näkyy.



Kuva 9. SQLite-tietokantatiedoston ominaisuudet.

App.xaml.cs-tiedostossa pitää poistaa taulujen määrittely ja tietokannan datan lisääminen. Koodiesimerkissä 42 on CopyDatabase-metodi, jossa haetaan .sqlite-tiedosto projektista ja kopioidaan se ohjelmaan. Olemassa olevat tietokantaoperaatiot eivät tarvitse minkäänlaisia muutoksia eristetyssä tietokannassa.

```
private async Task CopyDatabase()
        {
            try
            {
```

```
        StorageFile file = await
        StorageFile.GetFileFromApplicationUriAsync(new Uri(@"ms-
        appx:///SQLitedb.sqlite"));
        await file.CopyAsync(ApplicationData.Current.LocalFolder);
         }
         catch (Exception error)
         {
             Debug.WriteLine(error.Message);
             Debug.WriteLine(error.StackTrace);
         }
    }
```

Koodiesimerkki 42. Metodi kopioi SQLitedb.sqlite-tiedoston puhelimen paikalliseen juurikansioon.

8 Yhteenveto

Tässä insinöörityössä on tarkasteltu C#-ohjelmointikieltä, Windows 10 Mobile -alustaa, LINQ-hakuarkkitehtuuria ja sen käyttämistä SQLite-tietokannassa. Tämän työn alussa oli tarkoitus tehdä insinöörityö Windows Phone 8.1 Silverlight -mobiilialustalla ja käyttäen LINQ To SQL -tietokantaa. Teknologia kehittyy niin nopeasti, että oli järkevämpää siirtyä SQLite-tietokantaan, jotta aihe pysyy uutena ja tuoreena. LINQ To SQL ei valitettavasti päässyt Windows Phone 8.1 Silverlightista Windows 10 Mobile -alustalle, joten aiheen vaihto oli tarpeellinen.

Työssä opin erittäin paljon ei pelkästään Windows Phone 8.1 Silverlight -alustasta vaan myös Windows 10 Mobile -ohjelmien kehityksestä. Windows 10 -ohjelmointi on kiinnostavaa varsinkin, koska sillä voi ohjelmoida monelle alustalle yhdellä Visual Studio -projektilla. En ennen ollut niin suuresti kiinnostunut erilaisista tietokannoista, mutta tämä insinöörityö herätti kasvavan mielenkiintoni tietokantoja ja niiden käsittelyä kohtaan.

Pääpaino insinöörityössä oli kuitenkin LINQ-hakuarkkitehtuuri ja sen käyttö. IHaveIdeas-ohjelman kehityksessä huomasin, että LINQ oli erittäin mukava käyttää. LINQ on suurin syy jonka takia C#-ohjelmointikieli on oma suosikkini ohjelmointikielistä. Työmaailmassa LINQ-hakuarkkitehtuurin osaaminen on ollut itselleni myös suureksi hyödyksi. Insinöörityössä tehty Windows 10 Mobile -ohjelmaa voisi myös tulevaisuudessa kehittää kaikille Windows 10 -alustoille. Omasta mielestäni olen saavuttanut insinöörityön tavoitteet oppien paljon LINQ-hakuarkkitehtuurin toiminnasta sekä Windows 10 -alustalle kehityksestä.

Lähteet

1 LINQ (Language-Integrated Query). Verkkodokumentti. <https://msdn.microsoft.com/fi-fi/library/bb397897.aspx> Luettu 1.6.2015.

2 Albahari, Joseph, Albahari Ben. C# 5.0 in a Nutshell, 5th Edition. Yhdysvallat: O'Reilly Media.

3 De Smet, Bart. 2013. C# 5.0 Unleashed. Yhdysvallat: Sams Publishing.

4 Anonymous Types (C# Programming Guide). Verkkodokumentti.<https://msdn.microsoft.com/en-us/library/bb397696.aspx> Luettu 1.9.2015.

5 Lambda Expressions (C# Programming Guide). Verkkodokumentti.<https://msdn.microsoft.com/en-us/library/bb397687.aspx> Luettu 15.7.2015.

6 Windows 10. Tee asiat paremmin. Verkkodokumentti. <https://www.microsoft.com/fi-fi/laitteet/windows10/#> Luettu 27.12.2015.

7 A first look at the Windows 10 universal app platform. Verkkodokumentti. <https://blogs.windows.com/buildingapps/2015/03/02/a-first-look-at-the-windows-10-universal-app-platform/> Luettu 24.01.2016.

8 What's a Universal Windows app? Verkkodokumentti. <https://msdn.microsoft.com/library/windows/apps/dn726767.aspx> Luettu 15.9.2015.

9 The MVVM Pattern. Verkkodokumentti. <https://msdn.microsoft.com/en-us/library/hh848246.aspx> Luettu 1.10.2015.

10 Liberty, Jesse, Japikse, Philip, Galloway, Jon. Pro Windows 8.1 Development with XAML and C#. Yhdysvallat: Apress.

11 Test with the Microsoft Emulator for Windows 10 Mobile. Verkkodokumentti. <https://msdn.microsoft.com/fi-fi/library/windows/apps/mt188754.aspx> Luettu 10.10.2015.

12 Visual Studio Community. Verkkodokumentti. <https://www.visualstudio.com/products/visual-studio-community-vs> Luettu 18.10.2015.

13 XAML overview. Verkkodokumentti. <https://msdn.microsoft.com/fi-fi/library/windows/apps/xaml/mt185595.aspx> Luettu 01.01.2016.

14 Query Syntax and Method Syntax in LINQ (C#). Verkkodokumentti. <https://msdn.microsoft.com/en-us/library/vstudio/bb397947.aspx> Luettu 10.9.2015.

15 Owens, Michael. The Definitive Guide to SQLite. Yhdysvallat: Apress.

16 Krog, Øystein. SQLite.Net-PCL. GitHub repositio <https://github.com/oysteinkrog/SQLite.Net-PCL>. Luettu 3.10.2015.

IhaveIdeas lähdekoodi

https://github.com/Joonas-Virtanen/IHaveIdeas