TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jevgeni Sõritski
186025 IADB

Automaattestide arendamine meditsiinirakenduse näitel

Bakalaureusetöö

Juhendaja: Nadežda Furs
MBA

Tallinn 2021

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Jevgeni Sõritski

17.05.2021

Annotatsioon

Selle väitekirja eesmärk on arendada ja parandada automaatse testimise platvormi, mis võimaldaks kontrollida automaatselt Bait Partneri meditsiinilise rakenduse funktsionaalsust.

Töö käigus autor uuris erinevaid meetodeid ja tehnoloogiaid, mida kasutatakse automaatsete testide loomiseks: näiteks *mocking*, sõltuvuse sisestamine ja erinevad testimisraamistikud, et kasutada neid soovitud tulemuse saavutamiseks.

Testimisel kasutatavate erinevate tehnoloogiate ja meetodite analüüs võimaldas autoril paremini aru saada, kuidas ülesannet täita.

Lisaks tehti vajalikud muudatused ja täiendused rakenduste arhitektuuris, kirjutati automaatsed testid, mis kontrollivad rakenduse põhifunktsionalsust, eesmärgiga lihtsustada arendust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 26 leheküljel, 6 peatükki, 23 joonist, 0 tabelit.

Abstract

Development of Automation Testing by the Example of Medical Application

The testing processis is one of the most important stages in the lifecycle of software development, since it allows developers to eliminate errors and bugs. Indeed, this is through testing that it can be found out whether the code was written correctly, whether changes need to be made, and how it should be implemented in such a way that the final product does not contain errors and is convenient for users. Special attention to software testing should be paid by software companies working in the field of medicine, because mistakes in this area are unacceptable.

The aim of this thesis is to develop and improve an automatic testing platform, which would automatically control the functionality of a medical application developed by Bait Partner OÜ.

In this work, author studied various methods and technologies used to create automated tests, such as mocking, dependency injection, and various testing frameworks, in order to subsequently apply them to obtain the desired result.

In addition, the necessary changes and improvements were made to the architecture of the application, and the automatic tests were written to check the main functionality of the program to simplify development.

The thesis is in Estonian and contains 26 pages of text, 6 chapters, 23 figures, 0 tables.

Lühendite ja mõistete sõnastik

.NET	.NET Framework (.NET-raamistik) on Microsofti tarkvaraplatvorm
Back-end	Osa veebisaidist või rakendust, mida kasutajad ei näe, teiste sõnadega – rakenduse loogika
DAO	*Data Access Object*
IDE	Integrated development environment
IT	Information technology - infotehnoloogia
MVVM	*Model-View-ViewModel*
SDK	*Software development kit*
UWP	*Universal Windows Platform*
WPF	*Windows Presentation Foundation*
XAML	*eXtensible Application Markup Language*

Sisukord

1 Sissejuhatus 8
1.1 Lähtetingimused ja probleem 8
1.2 Ülesande püstitus 9
2 Tarkvara testimine 10
2.1 Visual Studio 11
2.2 Test Explorer 11
2.3 Dependency Injection 12
2.4 Mocking 15
3 Testitava rakenduse kirjeldus 19
3.1.1 Rakenduse arhitektuur 20
4 Testide kirjutamine 22
4.1 Testide planeerimine 22
4.1.1 *Back-end* loogika ja teenused 23
4.1.2 ViewModel 25
5 Testide käivitamine 31
6 Tulemused 33
Kokkuvõte 34
Kasutatud kirjandus 35
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 37

Jooniste loetelu

Joonis 1 *Test Explorer* 12
Joonis 2 A-klassis B-klassi uue eksemplari loomine [3] 13
Joonis 3 A-klass kasutades dependency injection [3] 13
Joonis 4 Car-klass dependency injection´ta 14
Joonis 5 Car-klass kasutades dependency injection 14
Joonis 6 *Fake* näide [5] 16
Joonis 7 Stub näide [5] 17
Joonis 8 Mock näide [5] 18
Joonis 9 Testitava rakenduse pealeht 19
Joonis 10 Testitava rakenduse uuringu seotud meedia leht 20
Joonis 11 Automaat testide kasutamine MVVM mallis [9] 21
Joonis 12 Arrange/Act/Assert näide 22
Joonis 13 Teenuse testimismeetod näide 23
Joonis 14 Failisüsteemiga töötava meetodi testi näide 24
Joonis 15 Test meetodi näide mis kasutab kasutajaliidese elemente 25
Joonis 16 Repositooriumi loomine meetodis 27
Joonis 17 Repositoorium konstruktoris 27
Joonis 18 Repositooriumi jaoks loodud liidesse näide 28
Joonis 19 Liidese kasutamine konstruktoris 28
Joonis 20 FakeItEasy raamastiku kasutamine testimismeetodis 29
Joonis 21 Staatilise meetodi kutsimine *FileWrapper*-is 30
Joonis 22 *FileHelper*-i imiteerimine testmeetodis 30
Joonis 23 Test Explorer rakenduse proektis 31

1 Sissejuhatus

Selle lõputöö eesmärk on automatiseeritud testide loomine Eesti ettevõte Bait Partner poolt välja töötatud UWP (*Universal Windows Platform*) rakenduse jaoks, mis kasutatakse meditsiini valdkonnas. Eeldatakse, et automatiseeritud testid aitavad lihtsustada programmi arendamist ning tuvastada ja vältida olulist osa vigadest enne toote kasutajani jõudmist.

1.1 Lähtetingimused ja probleem

Tänapäeval on testimisprotsess tarkvaraarenduse olelusringis üks olulisemaid etappe, kuna see võimaldab arendajatel vigu leida ja neid kõrvaldada. Testimise kaudu saavad arendajad teada, kas kood on õigesti kirjutatud, kas on vaja teha muudatusi ja kuidas neid tuleks rakendada nii, et lõpptoode oleks vigadeta ja kasutajasõbralik. Meditsiini valdkonna jaoks tarkvara arendatavad ettevõtted peaksid tarkvara testimisele erilist tähelepanu pöörama, kuna selles valdkonnas on vead vastuvõetamatud.

Bait Partner OÜ tegeleb IT lahenduste leidmise- ja väljatöötamisega meditsiini valdkonna jaoks. Ettevõtte peamised tooted on tarkvaralised ja riistvaralised lahendused patoloogia laboratooriumitele ja operatsioonisaalidesse. Hetkel on ettevõttel välja töötatud kaks lahendust *WPF* (*Windows Presentation Foundation*) platvormil. Nende lahenduste üks põhifunktsioone on meediafailide genereerimine, vaatamine ja märkuste lisamine. Kuid kuna need lahendused on liiga suured ja neid ei saa kasutada väljaspool tööruumi, otsustati luua uus toode meediafailidega töötamiseks. Kuna suur hulk meedia- ja patsiendiga seotud andmeid läbib selle toodet, on hädavajalik tagada, et pärast programmi loogikas muutuste sisseviimist, jäävad kõik andmed õigeks. Selle käsitsi kontrollimine on väga keeruline, ebaefektiivne ja aeganõudev tegevus, kuna inimlike eksimuste tõenäosus on suur. Selliste probleemide vältimiseks peaks kontrolli usaldama automatiseeritud testimissüsteemile (või programmile). Samuti tuleb märkida, et hetkel on väljatöötatud rakenduses testidega kaetud väga väike osa programmi funktsionaalsusest. Samuti ei sobi suure hulga programmi klasside ja mooduleid

automaatsete testide kasutamiseks, kuna neid ei saa eraldada andmebaasi ja failisüsteemi kihtidest. Seda aspekti tuleks automatiseeritud testide loomisel arvesse võtta.

1.2 Ülesande püstitus

Selle lõputöö eesmärk on automatiseeritud testide loomine Eesti ettevõte Bait Partneri poolt välja töötatud *UWP* rakenduse jaoks, mis kasutatakse meditsiini valdkonnas.

Selle eesmärgi saavutamiseks on vaja täita järgmised ülesanded:

- Automaatse testimise erinevate meetodite ja vahendite uurimine ja analüüs.
- Programmi üksikute osade refaktoreerimine, eesmärgiga tagada nende sobivus automaatsetele testidele
- Testide kirjutamine programmi põhifunktsionaalsuse testimiseks.

2 Tarkvara testimine

Tarkvara testimise eesmärk on kontrollida programmi tegeliku ja eeldatava käitumise kooskõla. Testimine on vajalik, sest me kõik teeme vigu ja kuigi mõned neist võivad olla tähtsusetud, võivad teistel olla kõige laastavamad tagajärjed. Sellepärast tuleb testida iga inimese poolt loodud tarkvara enne, kui see läheb laialdasse levikusse, et seda saaks tõhusalt ja ohutult kasutada.

Testimine on väga lai valdkond, kuid üldjuhul saab testimist jagada kaheks rühmaks: mittefunktsionaalne ja funktsionaalne testimine.

Tarkvara mittefunktsionaalne testimine on peamiselt mittefunktsionaalsete nõuete järgimise kontroll, näiteks:

- Kasutusmugavus
- Mastaapsus (kontrollitakse testitava rakenduse nii vertikaalset kui ka horisontaalset mastaapsust)
- Jõudlus (võime käivitada rakendust erinevate koormuste korral)
- Turvalisus (kasutajaandmete kaitse, rakenduse andmekaitse, häkkimistakistus).

Funktsionaalse testimise eesmärk on kontrollida, millised tarkvara funktsionaal on teostatud ning kui õigesti seda teostati. Funktsionaalsete testide seas võib eristada ühik- ja integreeritud testimist. Ühiktestimine seisneb rakenduse iga autonoomse funktsionaalsuse testimises eraldi, tehislikult loodud keskkonnas. Just vajadus luua konkreetse mooduli jaoks tehislik töökeskkond nõuab testijalt tarkvara testimise automatiseerimise teadmisi ja mõningaid programmeerimisoskusi. Integreeritud testimise puhul testitakse moodulite ühist tööd, s.t. kontrollitakse, kas omavahel ühendatud moodulid töötavad ja kas ilma vigadeta iseseisvalt töötavad moodulid tekitavad ühiseid vigu.

Automatiseeritud testimine aitab automatiseerida korduvaid, kuid testide katvuse maksimeerimiseks vajalikke ülesandeid. Automaatse testimise kasutamise peamised eesmärgid on järgmised:

- käsitsi testimise kulude vähendamine;
- silumise ja uue versiooni väljastamise aja vähendamine;
- vigade arvu vähenemine;
- riskide vähendamine;
- arhitektuuri täiustamine.

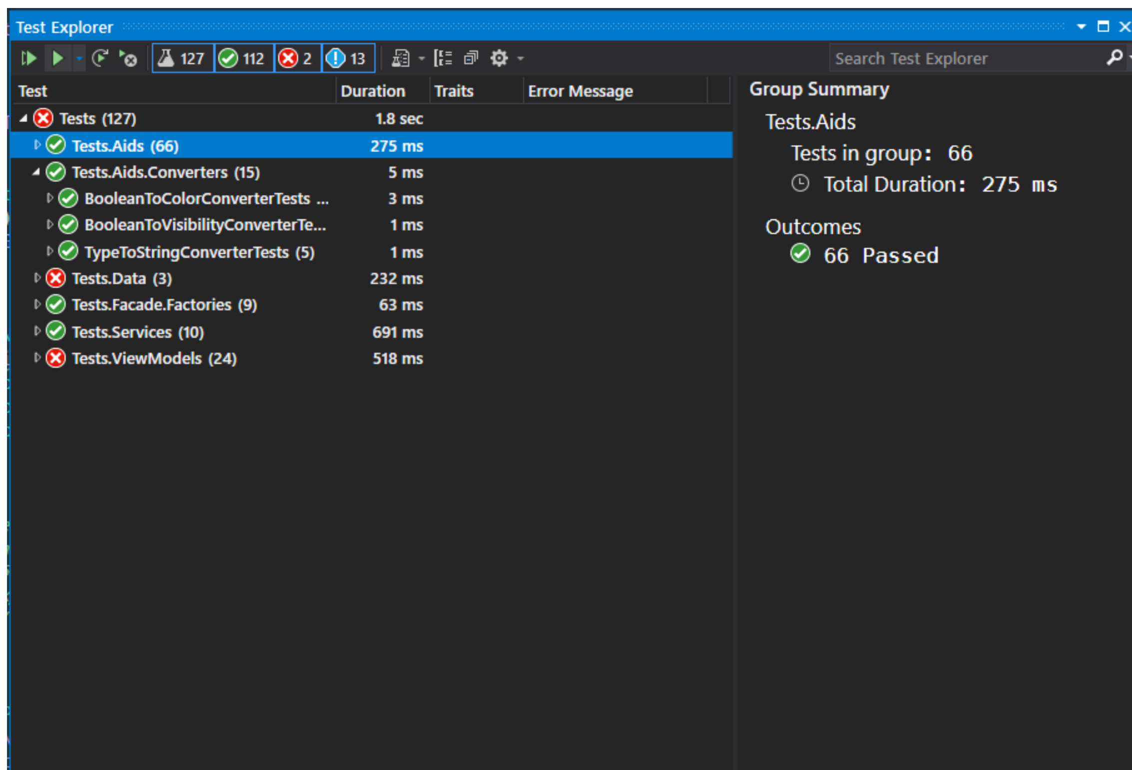
Automatiseeritud testimine hõlmab spetsiaalse tarkvara kasutamist (lisaks testitavale rakendusele) testide läbiviimise kontrollimiseks ja programmi eeldatava ja tegeliku tulemuse võrdlemiseks.

2.1 Visual Studio

Ettevõtes *UWP* -rakenduse arendamisel kasutatakse Microsoft-i *.NET*-platvormi. Visual Studio on *.NET*-rakenduste loomiseks kõige populaarsem, mugavam ja sagedamini kasutatav arenduskeskkond, mis on väljatöötanud Microsoft-i poolt. Visual Studio integreeritud arenduskeskkond (*IDE*) on rakenduste kirjutamise, silumise, koodide koostamise ja avaldamise lähtekoht. *IDE* on multifunktsionaalne programm, mida saab kasutada tarkvaraarenduse erinevate aspektide jaoks [1].

2.2 Test Explorer

Visual Studio eeliseks on *IDE* integreeritud testide vaatleja funktsioon – Test Explorer (Joonis 1). Test Explorer võimaldab käivitada üksuste teste Visual Studio või mõne muu osapoole üksuste testimisprojektidest. Lisaks võimaldab Test Explorer teste kategooriate järgi grupeerida, testide loendit filtreerida ning testide esitusloendeid luua, salvestada ja käivitada. Samuti on võimalik analüüsida testitud koodi mahtu ja siluda üksuste teste [2].



Joonis 1 *Test Explorer*

2.3 Dependency Injection

Dependency Injection printsiibi kohaselt iga objekti puhul kontrollitakse, mida tal oma tööks vaja läheb ja selle põhjal sisestatakse talle vajalikud sõltuvused.

Dependency Injection põhineb väga lihtsal kontseptsioonil: kui on olemas objekt, mis suhtleb teiste objektidega, liigub vastutus neile objektidele käitusajal viite leidmise eest väljaspool objekti ennast. Mida tähendab objekt "suhtleb" teiste objektidega? Tavaliselt, tähendab see objektide meetodite kutsumist või omaduste lugemist. Näiteks, kui meil on klass A, mis kutsub klassi B arvutama meetodi `Calculate`, võime öelda, et A suhtleb B-ga.

Samuti võib näidata, et A sõltub oma ülesannete täitmisel B-klassist. Sel juhul kutsub ta mitte ainult oma arvutusmeetodit, vaid loob ka selle klassi uue eksemplari, nagu on näidatud järgmises näites (Joonis 2).

```
class A
{
  private B _b;

  public A
  {
    _b = new B();
  }

  public int SomeMethod()
  {
    return (_b.Calculate() * 2);
  }
}
```

Joonis 2 A-klassis B-klassi uue eksemplari loomine [3]

Dependency Injection-i korral liigub vastutus B-tüüpi klassi rakendamise viite saamise eest väljaspool A-d (Joonis 3).

```
class A
{
  private _b = B;
  public A(B b)
  {
    _b = b;
  }
  public int SomeMethod()
  {
    return _(b.Calculate * 2);
  }
}
```

Joonis 3 A-klass kasutades dependency injection [3]

Sel juhul võib öelda, et sõltuvus (B) sisestati A-sse läbi konstruktori [3].

Lihtsamalt öeldes, kui funktsioon / objekt / moodul sõltub mõnest muust funktsioonist / objektist / moodulist, ei tohiks see luua ise seda mida ta vajab. Selle asemel anname ise talle kõik, mida vaja, tavaliselt edastades seda funktsiooni parameetrina.

Dependency injection-i kasutamisega on seotud järgmised eelised:

- Aitab üksuste testimisel;
- Rakenduse laiendamine muutub veelgi lihtsamaks;

- Aitab vähendada koodide sidusust, mis on oluline rakenduste arendamisel.

Dependency injection kasutatakse laialdaselt paljude kasutusjuhtumite toetamiseks, kuid kõige ilmsem on testimise lihtsustamine. See, kuidas sõltuvussisestamist testimisel kasutatakse, on järgmiste näidete abil hõlpsasti seletatav (Joonis 4 ja 5).

```
using Engine;
class Car
{
  private Engine _engine;
  public Car
  {
    _engine = new Engine();
  }
  Public void startEngine()
  {
       _engine.fireCylinders();
  }
}
```

Joonis 4 Car-klass dependency injection´ta

Joonisel 4 toodud näidisel, modelleerida `Engine,` osutub keeruliseks, sest klass `Car` loob sellest eksemplari, kuna alati kasutatakse tõelist `Engine`.

```
using Engine;
class Car
{
  private Engine _engine;
  public Car(Engine engine)
  {
    _engine = engine;
  }
  Public void startEngine()
  {
       _engine.fireCylinders();
  }
}
```

Joonis 5 Car-klass kasutades dependency injection

Kuid joonisel 5 toodud näitel on meil kontroll kasutatud mootori üle, mis tähendab, et testis saame luua alamklassi `Engine` ja selle ümber määrata meetodid. Näiteks kui on vaja näha, mida `Car.startEngine()` teeb, kui `engine.fireCylinders()` annab veateate, saab lihtsalt luua klassi `FakeEngine`, laiendada seda `Engine` klassist ja

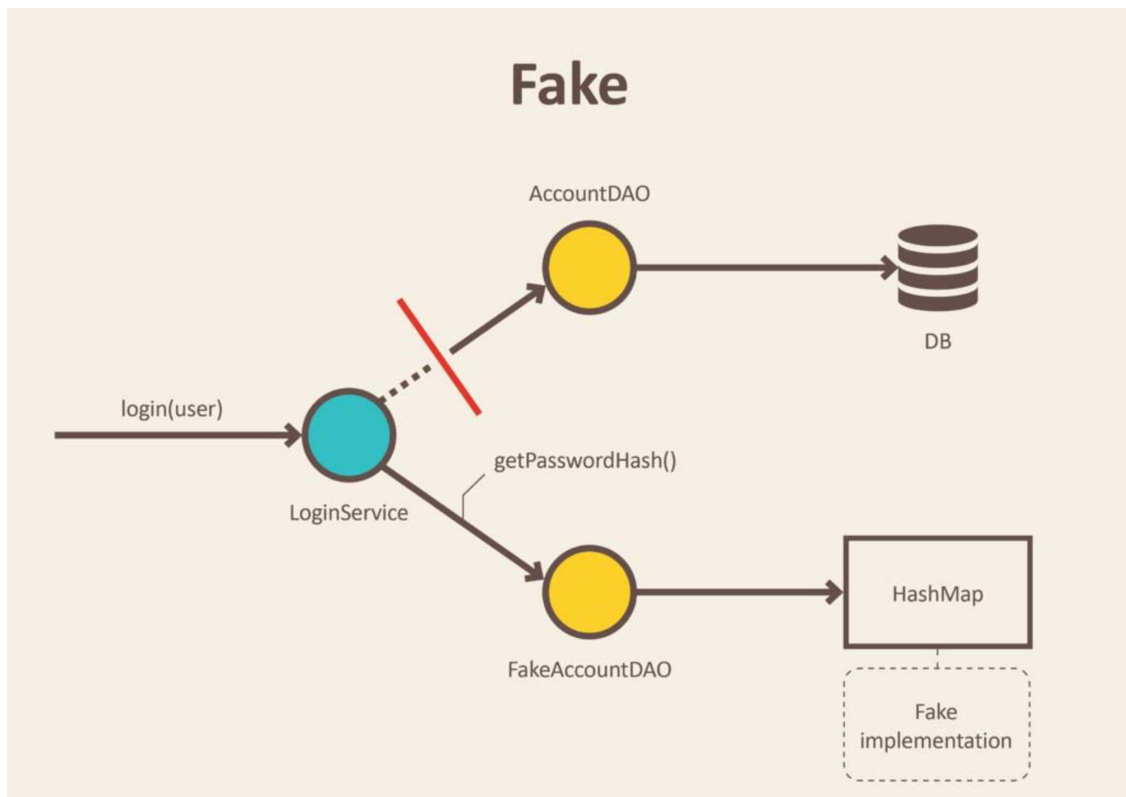
seejärel määrata ümber `fireCylinders` nii, et see annaks veateate. Testis saab selle `FakeEngine objekti` sisestada `Car` konstruktorisse [4].

2.4 Mocking

Dependency injection võimaldab hõlpsasti luua objekti koopiaid või imitatsioone. Programmeerimisel nimetatakse seda *mocking*. *Mocking*-i eesmärk on isoleerida testitav kood, et saaks keskenduda sellele, mitte aga väliste sõltuvuste käitumisele või olekule. *Mocking*-s asendatakse sõltuvused rangelt kontrollitavate asendusobjektidega, mis jäljendavad tegelike objektide käitumist. Asendusobjektidel on kolm peamist võimalikku tüüpi – *fake*-id, *stub*-id ja *mock*-id.

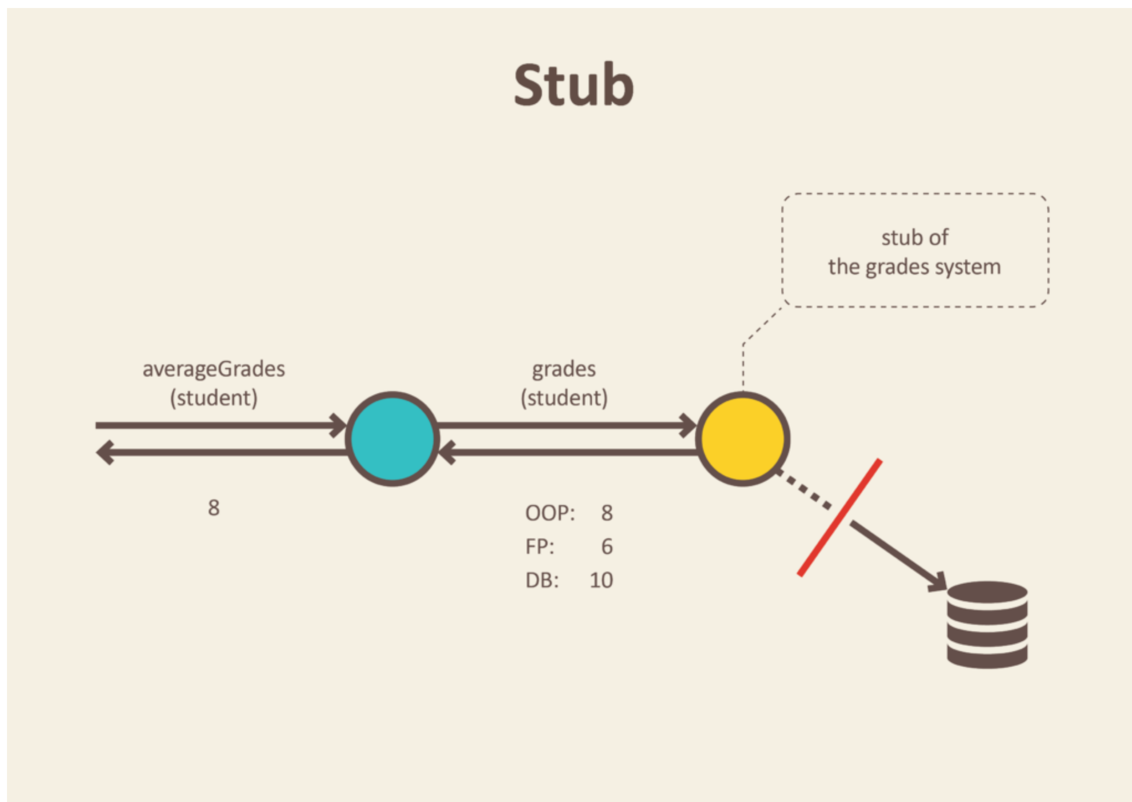
Fake ehk võltsing on objekt, mis asendab reaalse koodi, rakendades sama liidest, kuid ilma teiste objektidega suhtlemata. Tavaliselt võltsing on programmeeritud fikseeritud tulemuste tagasisaatmiseks. Erinevate kasutusjuhtumite testimiseks tuleb kasutusele võtta palju võltsinguid. Võltsingute kasutamisel on probleemiks see, et liidese muutmisel tuleb muuta kõiki võltsinguid, mis seda liidest rakendavad.

Näitena võib tuua *Data Access Object* (*DAO*) või repositooriumi teostamine ilma reaalse andmebaasi kasutamata (Joonis 6). Selle võltsingu teostus ei kasuta andmebaasi, vaid kasutab andmete salvestamiseks lihtsat listi. See võimaldab läbi viia teenuste integreeritud testimist ilma andmebaasi käivitamata ja pikki päringuid teostamata. [5].



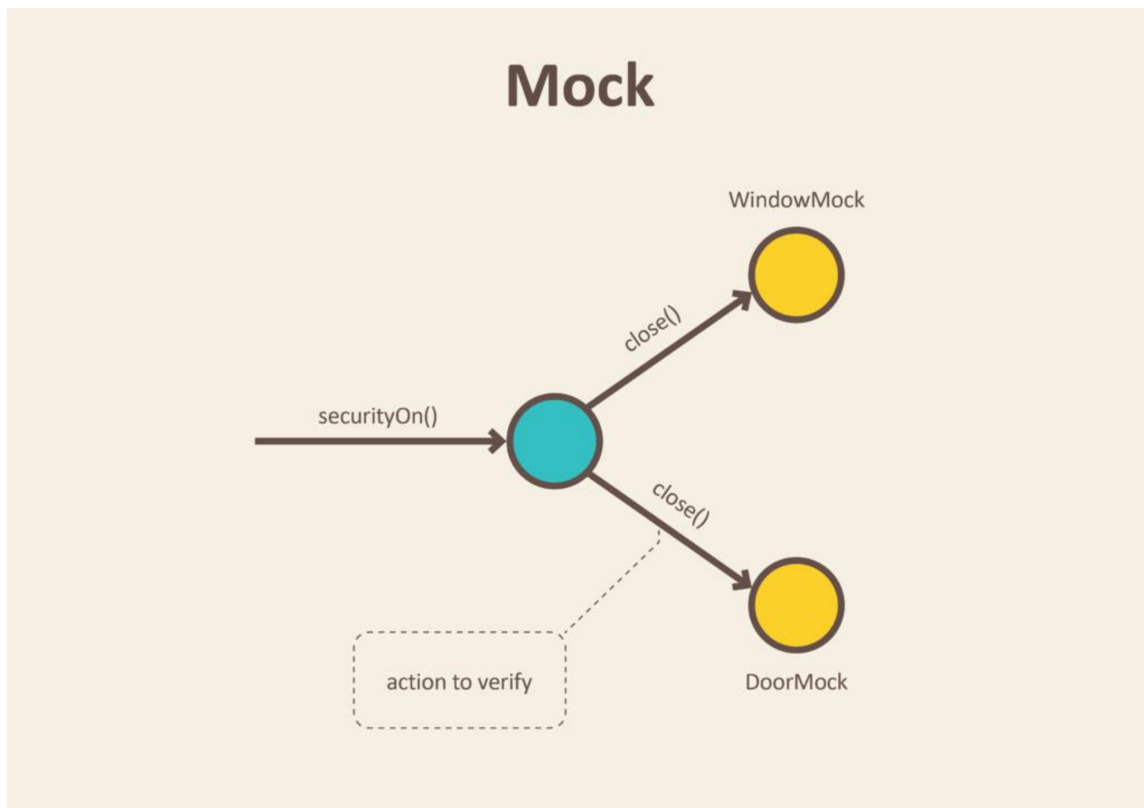
Joonis 6 *Fake* näide [5]

Stub on objekt, mis tagastab kindla sisendkomplekti põhjal saadud kindla tulemuse ja tavaliselt ei reageeri millelegi muule peale selle, mis on testi jaoks programmeeritud [1]. Näitena võib tuua objekti, mis peab meetodikutse vastamiseks, andmebaasist mõned andmed hankima. Pärisobjekti asemel sisestatakse *stub* ja määratakse kindlaks, millised andmed tuleks tagastada (Joonis 7) [5].



Joonis 7 Stub näide [5]

Mock on *stub*-i keerulisem versioon. See tagastab samuti sellised väärtused nagu *stub*, kuid seda saab programmeerida ka arvestades ootusi, näiteks selle kohta, mitu korda iga meetodit tuleks kutsuda, millises järjekorras ja milliste andmetega [6].



Joonis 8 Mock näide [5]

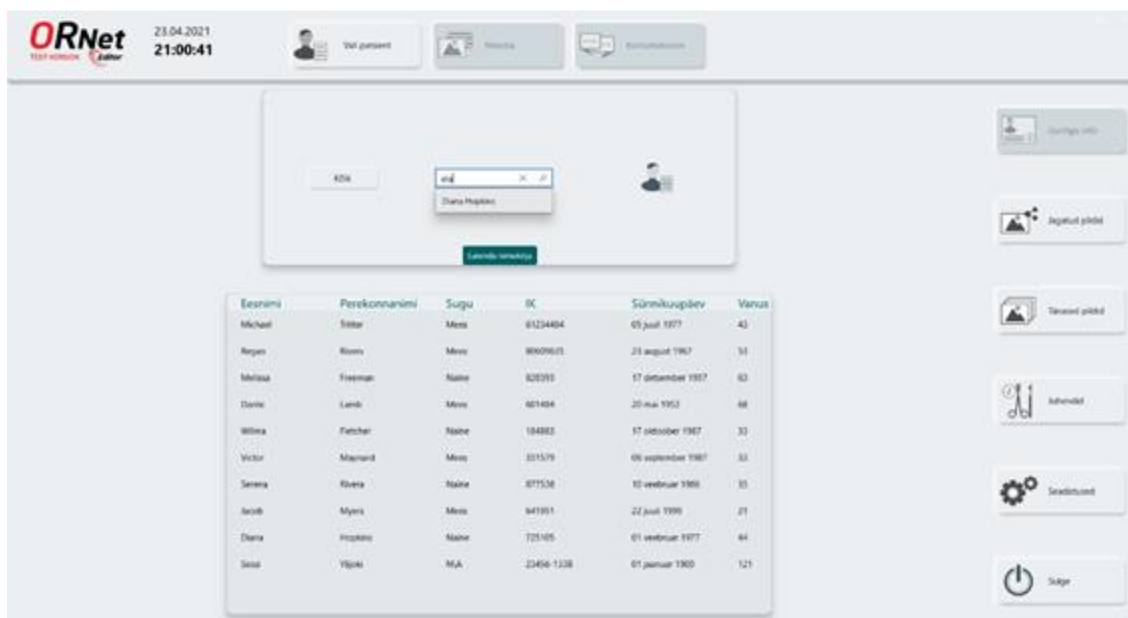
Näitena võib tuua funktsiooni, mis kutsub e-posti teenust. Iga kord, kui teeme testi, pole vaja e-kirju saata. Lisaks ei ole testides lihtne kontrollida, kas saadeti õige e-kiri. Ainuke asi, mida teha saab, on kontrollida testis rakendatud funktsionaalsuse tulemusi. Muudel juhtudel peab veenduma, et e-posti teenus on kutsutud [5].

Mugavama ja produktiivsema testi kirjutamise protsessi jaoks on palju erinevaid raamistikke, mis võimaldavad simuleerida või jäljendada mõnda funktsionaalsust või luua *mock* objekte. Sarnaseid raamistikke on palju ja mõned kõige populaarsemad, mis on loodud *.NET*-i keelega töötamiseks, on *Moq* või *Rhino Mocks*, neid kasutatakse ka ettevõtte teistes rakendustes. Kuid selles lõputöös kasutati *FakeItEasy* [7] raamistikku peamiselt selle kasutusmugavuse ning arusaadavama ja loogilisema semantika tõttu.

3 Testitava rakenduse kirjeldus

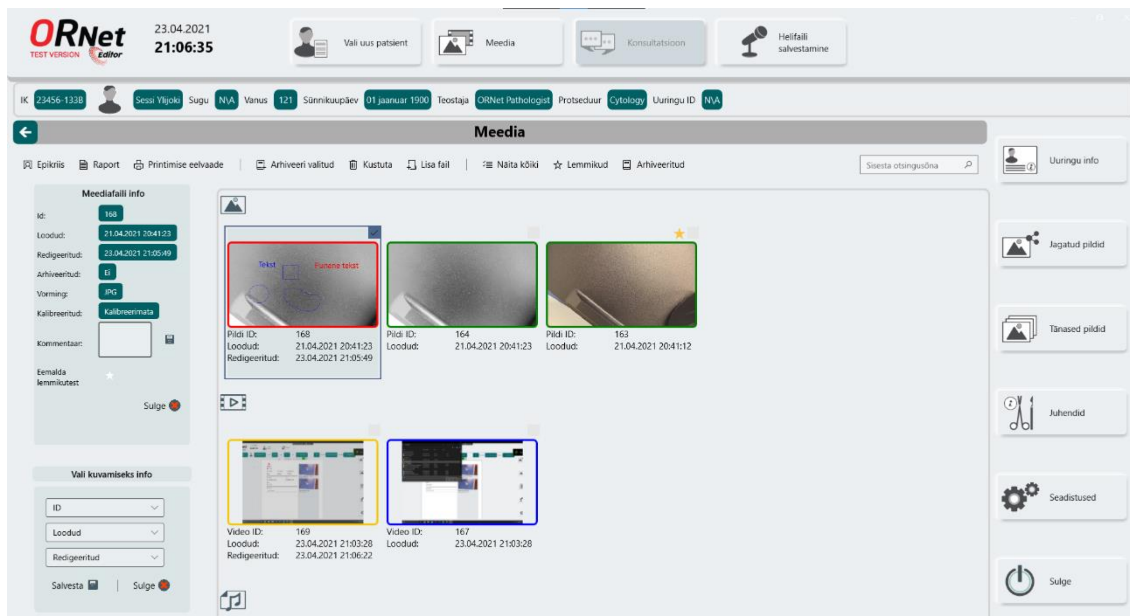
Ettevõtte Bait Partner rakendus arendatakse *UWP* platvormil ja see võib töötada kõikide Windows 10 versioonidega.

Rakenduse peamine eesmärk on anda meditsiinitöötajatetele (eriarstid, kirurgid, patoloogid) juurdepääs patsiendi meediaandmetele, mis asuvad meditsiiniasutuste andmebaasides (infosüsteemides). Seega on arstidel võimalus uurida varasemate protseduuride andmeid ja teha nende põhjal mõningaid järeldusi. Rakendus võimaldab erinevaid toiminguid meediafailidega - piltidele annotatsioonide lisamine, video-ja audiofailide töötlemine, failide salvestamine ja arhiveerimine. Rakenduse loomisel on arvesse võetud kasutajasõbralikkust ja turvalisust, erinevad filtrid võimaldavad sorteerida kas kuupäeva, protseduuri tüübi või patsiendi nime järgi. Joonisel 9 on toodud testitava rakenduse pealeht.



Joonis 9 Testitava rakenduse pealeht

Joonisel 10 on toodud testitava rakenduse uuringu meediafaillileht.



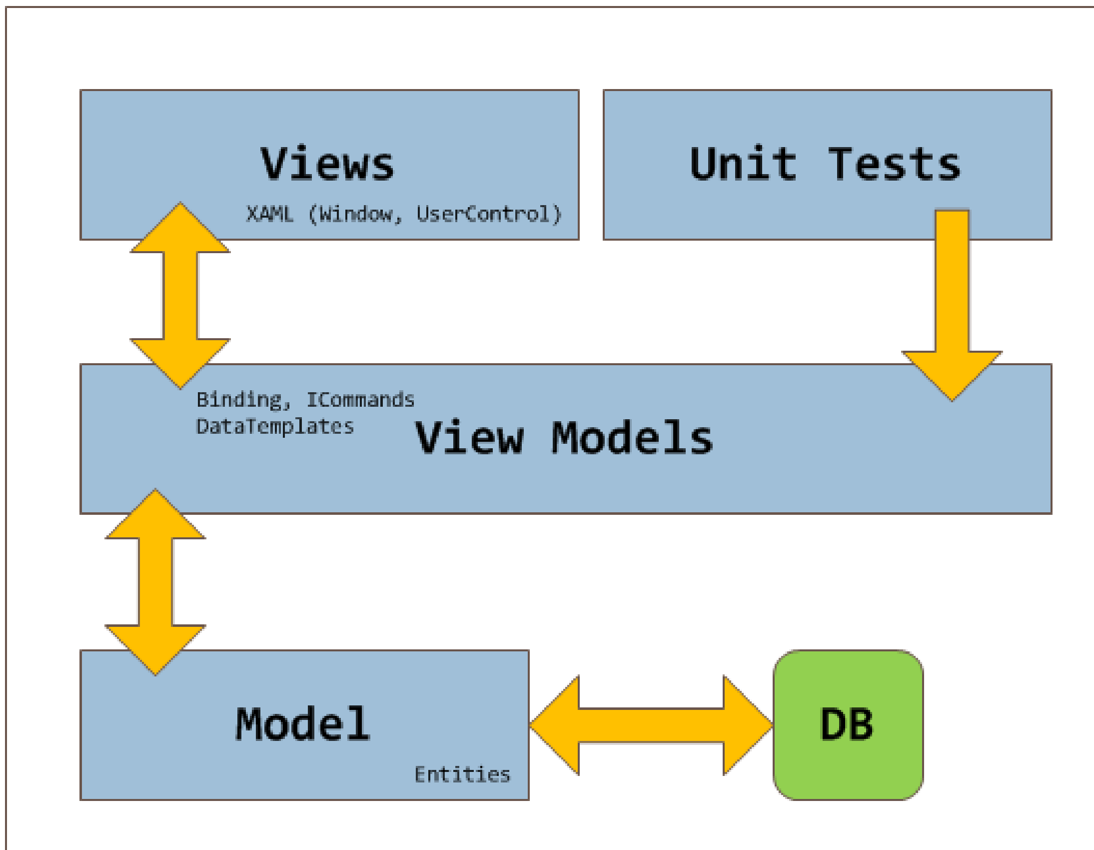
Joonis 10 Testitava rakenduse uuringu seotud meedia leht

Kuna kõik andmed paiknevad haigla üldandmebaasis, siis on kõik muudatused nähtavad sama andmebaasi kasutava ettevõtte teistes rakendustes.

3.1.1 Rakenduse arhitektuur

Selleks, et paremini mõista, kuidas saaks testida Bait Partner *UWP* rakendust, peab aru saama rakenduse arhitektuuri põhiprintsiipidest.

Rakendus arendatakse *Model-View-ViewModel* (*MVVM*) malliga. *MVVM* on kasutajaliidese arhitektuuri kujundusmall, mis on mõeldud kasutajaliidese koodi eraldamiseks ülejäänud koodist. *MVVM*-iga saate *XAML*-i kasutajaliidese (*eXtensible Application Markup Language*) deklaratiivselt tuvastada ja kasutada andmete sidumise märgistust, et linkida see teiste andmeid ja käske sisaldavate kihtidega. Andmete sidumise infrastruktuur loob nõrga ühenduse, mis sünkroonib kasutajaliidese ja sellega seotud andmed ning saadab kasutaja sisendi vastavatele käskudele. Kuna side on nõrk, vähendab andmete sidumise kasutamine jäiku korrelatsioone erinevat tüüpi koodide vahel. See lihtsustab üksikute koodiplokkide (meetodid, klassid, juhtelemendid jne) muutmist ilma tahtmatu kõrvalmõjuta teistele plokkidele [8].



Joonis 11 Automaat testide kasutamine MVVM mallis [9]

Tulenevalt asjaolust, et seda tüüpi arhitektuuris on programmi põhiloogika kasutajaliidesest eraldatud, on automaatsetes testides seda üsna lihtne kasutada, nagu on näidatud joonisel 11.

4 Testide kirjutamine

Testide kirjutamiseks kasutati malli *Arrange/Act/Assert* (*AAA*). See on mall koodi korrastamiseks ja vormindamiseks ühiku testimismeetodites. Soovitav on luua testid loomulikumal ja mugavamal viisil. Idee on töötada välja ühik test, tehes järgmised 3 lihtsat sammu:

1. *Arrange* – võimaldab seadistada katseobjekte ja valmistada ette testimise eeltingimused.
2. *Act* - teha testi tegelikku tööd.
3. *Assert* - kontrollida tulemust.

Joonisel 12 on toodud näide *AAA* malli kasutamisest testitavates rakendustes.

```
[TestMethod]
  public void ShouldCalculateLengthBetweenTwoPointsTest2() {
      //Arrange
      var p1 = new Point(5, 5);
      var p2 = new Point(1, 2);
      var expected = 5;
      //Act
      var actual =
MathHelper.CalculateLengthBetweenTwoPoints(p1, p2);
      //Assert
      Assert.AreEqual(expected, actual, $"Should be equal
to {expected}");
  }
```

Joonis 12 Arrange/Act/Assert näide

4.1 Testide planeerimine

Enne testimise alustamist on oluline kindlaks teha, millistel programmi osadel on kõrgeim prioriteet ja mis vajavad pidevat testimist. Peamiste kasutajate - arstide ja laboritöötajate seisukohast on väga oluline, et programmiloogika käigus jääksid muutumatuks meedifailidega seotud uuringute ja patsientide andmed ka peale meediafailide töötlust ja andmebaasi salvestamist. Kuna rakendus töötab meediafailidega ning nende eest vastutavatesse moodulitesse tehakse regulaarselt muudatusi ja parandusi, on oluline teha

ka nende kontrollimine mugavaks. Seega saab põhifunktsioonid, mida tuleb kontrollida, jagada kahte rühma:

- Erinev *back-end*-i loogika ja teenused.
- Rakenduse loogika, mis asub *ViewModel*-is.

4.1.1 *Back-end* loogika ja teenused

Programmi üks olulisemaid osi, mis nõuab suurt tähelepanu, on teenused, mis vastutavad andmete kogumise eest *ViewModel*-i jaoks. Seal kogutakse andmeid mitmelt eraldi lehelt ühte tervikusse, et neid oleks *ViewModel*-is lihtne kasutada. On väga oluline kontrollida, et pärast selliseid töötlusi andmetega miski pole muutunud ja pole tekkinud segadust. Joonisel 13 on toodud näide testist, mis kontrollib ühte neist teenustest.

```
       [TestMethod]
        public void ShouldReturnListWithCorrectDataTest()
        {
            setUpData();
            var patient = new Patient()
            {
                Birthdate = DateTime.Parse("1/1/2000"),
                FirstName = "Bob",
                Gender = "Male",
                Id = 1,
                LastName = "Bob",
                SSN = "QWERTY1234"
            };
            var vieModels = 
ExaminationViewModelsListFactory.GetList(
                _examinationList,
                _examinationTypes,
                patient,
                _examiners);
            Assert.IsTrue(vieModels.Count == 
_examinationList.Count);
            foreach (var viewModel in vieModels)
            {
                Assert.IsTrue(viewModel.Patient.FirstName == 
patient.FirstName
                              && viewModel.Patient.LastName == 
patient.LastName
                              && viewModel.Patient.SSN == 
patient.SSN);

                Assert.IsTrue(viewModel.ExaminerId == 
_examiners[0].Id);
            }

        }
```

Joonis 13 Teenuse testimismeetod näide

Faili süsteemiga seotud programmi loogika testimiseks saab kasutada ainult testprojekti katalooge, kuna *UWP*-l on väga piiratud juurdepääs operatsioonisüsteemi ülejäänud kataloogidele. All poolt on toodud näide: kus testitakse meetodit, mis töötab failisüsteemiga (Joonis 14).

```
[TestMethod]
 public async Task ShouldCopyFileTest()
 {
     var folder = await
StorageFolder.GetFolderFromPathAsync(ApplicationData.Current.LocalF
older.Path);
     var fileName = "test.txt";
     var newFileName = "test2.txt";

     await folder.CreateFileAsync(fileName);
     await FileHelper.CopyFile(Path.Combine(folder.Path,
fileName), folder.Path, newFileName);
     var fileExists = File.Exists(Path.Combine(folder.Path,
newFileName));
     File.Delete(Path.Combine(folder.Path, fileName));
     if (fileExists)
     {
         File.Delete(Path.Combine(folder.Path,
newFileName));
     }

     Assert.IsTrue(fileExists, "No copied file found");
 }
```

Joonis 14 Failisüsteemiga töötava meetodi testi näide

Meetodites, kus kasutati kasutajaliidese elemente oli vaja kasutada spetsiaalset silti (*tag*) "`UITestMethod`". See võimaldab kasutada kasutajaliidese voogu testmeetodis, mis ei ole lubatud meetodites, mis ei tööta otse kasutajaliidesega. Näide testidest, mis kasutab kasutajaliidese elemente on toodud joonisel 15.

```
[UITestMethod]
 public void ShouldGetRightToolByGeometryTest()
 {
     var path1 = new Path();
     var path2 = new Path();
     var path3 = new Path();

     path1.Data = new EllipseGeometry();
     path2.Data = new GeometryGroup();
     path3.Data = new RectangleGeometry();

     var tool1 = GeometryHelper.GetToolByGeometry(path1);
     var tool2 = GeometryHelper.GetToolByGeometry(path2);
     var tool3 = GeometryHelper.GetToolByGeometry(path3);

     Assert.AreEqual(tool1, Enums.Tools.Circle);
     Assert.AreEqual(tool2, Enums.Tools.Arrow);
```

```
        Assert.AreEqual(tool3, Enums.Tools.Rectangle);
    }
```

Joonis 15 Test meetodi näide mis kasutab kasutajaliidese elemente

4.1.2 ViewModel

*ViewModel-te t*estimisel kasutati punktides 2.3 ja 2.4 kirjeldatud erinevaid viise komponentide eraldamiseks välissõltuvustest. Alguses oli seda võimatu teha, kuna oli valesti teostatud ühendus *data access layer* programmi äriloogikaga, mis asub *ViewModel*-is. Andmebaas on ühendatud repositoriumi kaudu, mille meetodid saavad andmed andmebaasist ja teisendavad neis objektideks. Need repositoriumid kutsutakse uue objekti initsialiseerimisega otse klassi *ViewModel* sees. Seetõttu, kui proovida kutsuda meetod või luua klassi testmeetodis, see tekitab vea, mis on seotud asjaoluga, et repositoorium proovib andmebaasiga ühendust luua ja sealt andmeid saada. Seega ei saa selliseid klasse isoleerida. Näiteks `GetMediaList()` ühes *ViewModel*-is (Joonis 16).

```
public async void GetMediaList()
        {
            AuditLogger.LogInfo<MediaPageViewModel>($"Loading 
media for patient with id {Cache.GetPatientId()}.");
            var mediaRepository = new 
MediaRepository(_connectionString);
            var patientId = Cache.GetPatientId();
            var examId = Cache.GetExaminationId();
            var response = 
mediaRepository.GetNotSharedListByPatientId(patientId, examId);
            if (response == null) return;
            var mediaList = 
response.OrderByDescending(m=>m.Edited).ToList();
            mediaRepository.Dispose();
            var folders = new[] { _settings.StillPathServer, 
_settings.VideoServerPath, _settings.AudioPath };
            _logger.LogInfo<MediaPageViewModel>("Checking 
missing patient folders");
            await 
PatientFolderProvider.CheckMissingPatientFolders(folders);
            _logger.LogDebug<Media>(
                $"Media list found for patient (ID: 
{Cache.GetPatientId()}) " +
                $"and examination (ID: 
{Cache.GetExaminationId()}) contains {mediaList.Count} 
elements.");
            foreach (var media in mediaList)
            {
                if (_token.IsCancellationRequested)
                {
                    _logger.LogInfo<Media>("GetMediaList method 
was canceled");
                    return;
                }

                var path = 
_mediaLocatorService.GetFilePath(media);
                if (!await _fileHelper.IsFileExists(path)) 
continue;
                var collection = new 
ObservableCollection<Domain.Models.Media> { media };
                var mediaCollection = await 
_mediaViewModelsListFactory.SetThumbnails(collection, 
_mediaLocatorService);

                if (mediaCollection.Count < 1)
                {
                    continue;
                }

                var item = mediaCollection.First();
                _mediaItems.Add(item);
                AddMediaToCollection(item);
```

```
            }

            CheckIfNoContent();

            RaiseFiltersCanExecuteChanged();
        }
```

Joonis 16 Repositooriumi loomine meetodis

Selle probleemi lahendamiseks aitab *ViewModel*-i klassi refaktoreerimine, kasutades sõltuvuse sisestamist klassidesse, kus on oluline isoleeruda andmebaasist, failisüsteemist jne.

Kui objekti repositoorium kasutatakse regulaarselt praeguste andmete salvestamiseks või värskendamiseks, objekt saab repositooriumit parameetri kujul konstruktori kaudu (Joonis 17). Kui repositooriumit kasutatakse andmete saamiseks ainult üks kord, saab neid andmeid taotleda üks kord enne objekti loomist ja laadida need objekti parameetrina üle konstruktorisse.

```
public MediaRepository _mediaRepository;

public MediaPageViewModel(ILogManager logger,
IMediaLocatorService     mediaLocatorService, CancellationToken
token,
SystemSettings systemSettings, MediaRepository mediaRepository,
IFileHelper fileHelper, IMediaViewModelsListFactory
mediaViewModelsListFactory)
        {
            _mediaRepository = mediaRepository;
        }
```

Joonis 17 Repositoorium konstruktoris

Aga see ei ole veel piisav testimismeetodi kirjutamiseks. Objekti loomiseks testimismeetodis, siiski kasutatakse repositooriumit, mis suhtleb reaalse andmebaasiga, mida aga ei tohi lubada. Probleemi lahendamiseks aitab selle repositooriumi imiteerimine, kus saab vajadusel täpsustada, milliseid andmeid peab sellest repositooriumist saama. Imiteerimiseks on vaja luua liides ehk *interface* (Joonis 18).

```
public interface IMediaRepository
    {
        IEnumerable<Media> GetListByIds(List<int> ids);
        IEnumerable<Media>
GetStillListByPatientExaminationFromDate(int patientId, int
examinationId, DateTime date);
        IEnumerable<Media> GetStillListFromDate(DateTime date);
        IEnumerable<Media> GetListByPatientAndExaminationIds(int
patientId, int examinationId, Enums.MediaType type);
        IEnumerable<Media> GetSharedMediaList();
        IEnumerable<Media> GetNotSharedListByPatientId(int
patientId, int examinationId);
        Media GetById(int id);
        void UpdateMediaContainerId(int id, int containerId);
        void UpdateMediaEditedTime(int id, DateTime
editedDateTime);
        void UpdateMediaDuration(int id, double duration);
        void UpdateMediaDicomStatus(int id, int dicomStatus,
DateTime date);
        void UpdateMediaIsCompletedToFalse(int id);
        void UpdateMediaIsFavorite(int id, int value);
        void UpdateMediaIsCompletedToTrue(int id, DateTime
completed);
        void UpdateMediaNotes(int id, string notes);
        void InsertMedia(Media media);
        void DeleteMediaById(int id);
        void Dispose();
    }
```

Joonis 18 Repositooriumi jaoks loodud liidesse näide

Liideses on määratud meetodid, mis tuleb luua sellest päritud klassist. Pärast liidese loomist on vaja määrata seda konstruktoris (Joonis 19).

```
public IMediaRepository _mediaRepository;

public MediaPageViewModel(ILogManager logger,
IMediaLocatorService     mediaLocatorService, CancellationToken
token,
SystemSettings systemSettings, IMediaRepository mediaRepository,
IFileHelper fileHelper, IMediaViewModelsListFactory
mediaViewModelsListFactory)
        {
            _mediaRepository = mediaRepository;
        }
```

Joonis 19 Liidese kasutamine konstruktoris

Nüüd saab luua repositooriumi imitatsiooni, kasutades eraldi klassi loomist või imiteerida spetsiaalse raamistikuga. *FakeItEasy* raamastiku kasutamine *ViewModel*-i meetodi testimiseks on kujutatud joonisel 20.

```
        [TestMethod]
         public void ShouldRemoveMediaFromMediaCollectionTest()
         {
             setData();
             var fakeMediaRep = A.Fake<IMediaRepository>();
             var fakefileHelper = A.Fake<IFileHelper>();
             var fakeMediaLocator =
A.Fake<IMediaLocatorService>();
             var fakelistFactory =
A.Fake<IMediaViewModelsListFactory>();
             A.CallTo(() => fakefileHelper.IsFileExists("path"))
.Returns(true);
             A.CallTo(() =>
fakeMediaRep.GetNotSharedListByPatientId(A<int>.Ignored,
A<int>.Ignored))
.Returns(_mediaList);
             A.CallTo(() =>
fakeMediaLocator.GetFilePath(A<Media>.Ignored)).Returns("path");
             A.CallTo(() =>

fakelistFactory.SetThumbnails(A<ObservableCollection<Media>>.Ign
ored,
                         A<MediaLocatorService>.Ignored))
                 .ReturnsNextFromSequence(_list[0], _list[1],
_list[2]);

             var viewModel = new MediaPageViewModel(new
LogManager(),
                 fakeMediaLocator,
                 new CancellationToken(false),
                 new SystemSettings(), fakeMediaRep,
fakefileHelper, fakelistFactory);

             viewModel.GetMediaList();
             Assert.IsTrue(viewModel._mediaItems.Count == 3);
             Assert.IsTrue(viewModel.StillCollection.Count == 2);
             Assert.IsTrue(viewModel.AudioCollection.Count == 1);
             Assert.IsTrue(viewModel.VideoCollection.Count == 0);
         }
```

Joonis 20 FakeItEasy raamastiku kasutamine testimismeetodis

Lisaks repositooriumitele testimismeetodite kirjutamise protsessi raskendasid mõned teenused, millised olid tehtud staatiliste klassidena. Staatilistel klass ei saa päranduda liideselt. Seetõttu, selleks et sellest klassist isoleeruda, ei piisa ainult objekti

konstruktorisse määramisest, sest hiljem pole võimalik imiteerida tema funktsioone testimismeetodis. Staatiliste klasside imiteerimiseks kasutati nn *wrapper* klassid. *Wrapper* on klass, mis kordab kõiki staatilise klassi meetodeid. Kõik need meetodid kutsuvad staatilisest klassist välja vastava meetodi ja tagastavad selle väärtuse. Kuna *wrapper* pole staatiline klass, saab selle jaoks luua liidese ja imiteerida *wrapper*-it testimismeetodis.

Näidisena võib tuua staatilise klassi *FileHelper*, mida kasutatakse üsna tihti kõigis programmi komponentides. Testimise ajal võib tekkida probleeme failisüsteemiga, kus *FileHelper* proovib leida olematu faili. Samuti pole soovitav, et testimeetodid toimiksid failisüsteemiga, välja arvatud testid, kus kontrollitakse *FileHelper*-i funktsionaalsust. *FileHelper*-st isoleerimiseks loodi klass *FileWrapper*. Joonisel 21 toodud näide, kus *FileWrapper* kutsub staatilist meetodit *FileHelper*-i klassist.

```
public class FileWrapper : IFileHelper

    {
        public Task<bool> IsFileExists(string path)
        {
            return FileHelper.IsFileExists(path);
        }
    }
```

Joonis 21 Staatilise meetodi kutsimine *FileWrapper*-is

Sel juhul on see meetod nimega `isFileExists`. Kuna *FileWrapper* pärandub liideselt, testimeetodis saab määrata `isFileExists` meetodi tagastatava väärtust ilma faile kontrollimata (Joonis 22).

```
var fakeFileHelper = A.Fake<IFileHelper>();

            A.CallTo(() =>
fakeFileHelper.IsFileExists(A<string>.Ignored)).Returns(true);
```

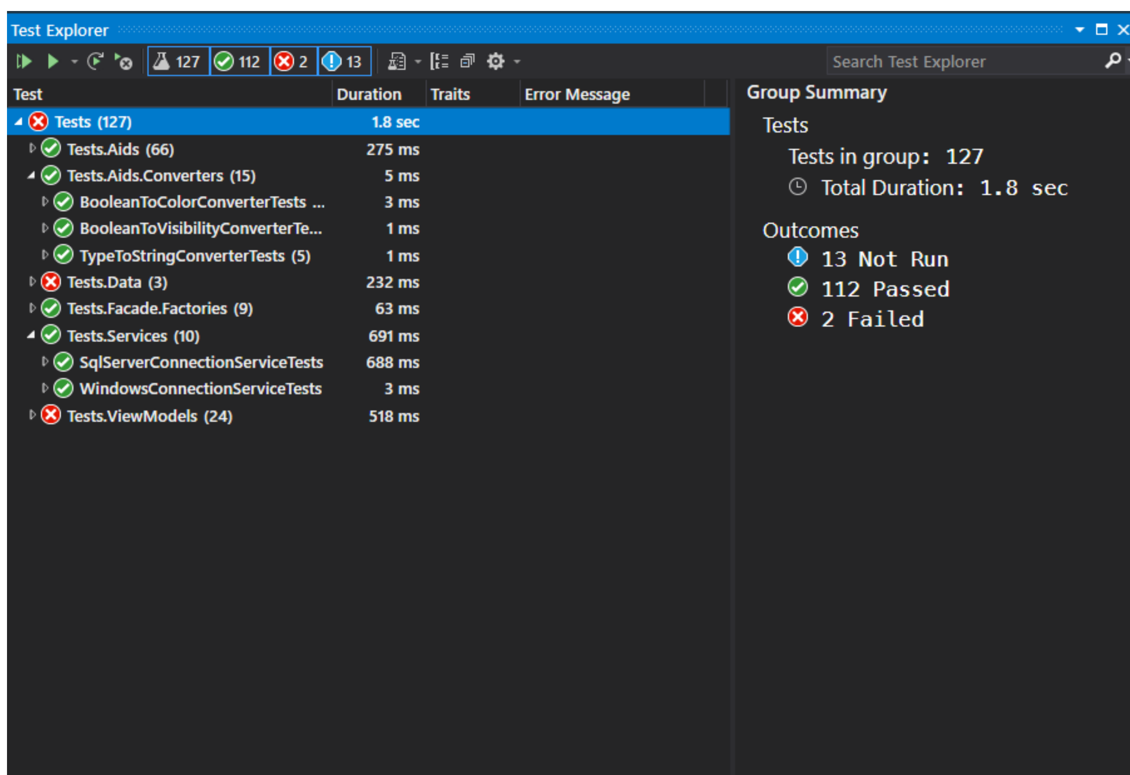
Joonis 22 *FileHelper*-i imiteerimine testmeetodis

Töö käigus olid refaktoreeritud juba olemasolevad *ViewModel*-id , et parandada eespool kirjeldatud viga. Uued *ViewModel*-id loodi kasutades uut ja õiget skeemi. *ViewModel*-i ja teiste välissõltuvusega klasside testid olid kirjutatud sarnaste mallide abil, mis on toodud joonistel 13, 14 ja 15.

5 Testide käivitamine

Projekti avamisel on igal arendajal võimalus käivitada teste ja kontrollida programmi peamiste komponentide töövõimet. Samuti võivad testid töötada eraldi, näiteks kui vaja kiiresti kontrollida koodi toimingute õigsust pärast muudatusi, ilma rakenduse käivitamiseta, mis mõnikord võtab üsna palju aega. Kui test teatab veast, saab kiiresti minna testimismeetodile ja vaadata, mis oli valesti. Bait Partner ettevõttes käivitatakse testprojekt kindlasti enne rakenduse väljalaskeversiooni väljaandmist, mis pärast mitmeid visuaalseid käsitsi kontrolle saab alles klientidele kättesaadavaks.

Aeganõudvaid protsesse lihtsustatakse uue testkeskkonna abil. Praeguse lähenemisviisi korral vähendatakse raskesti ligipääsetavate andmete kontrollimist, mida on raske kontrollida, sekunditele. Pealegi pole vaja ajakohast suhtlust kolmandate osapoolte teenustega, mis ei pruugi alati saadaval olla. Joonisel 23 on toodud Test Exploreri rakenduse vaade, kus on näha, et testimine võtab vaid paar sekundit aega, samas kui ühe testistsenaariumi käsitsi läbiviimine võib võtta mitu minutit.



Joonis 23 Test Explorer rakenduse projektis

Teste saab käivitada ja tulemusi vaadata vajutades vastavat nuppu Test Explorer-is või kasutades spetsiaalset käsku käsureal. Testide käivitamiseks käsurealt peab kasutama spetsiaalset tööriista vstest.console.exe, mis asub Windows *Software development kit*-s (SDK). Selleks on vaja määrata faili projekti nimega ja laiendiga *.build.appxrecipe* või *.dll*. Need failid luuakse projekti juurkataloogis pärast projekti edukat ehitamist. On olemas võimalus kasutada erinevaid atribuute, et määrata täiendavaid sätteid. Näiteks, saab määrata nimed üksikutele testimismeetoditele, mis on vaja käivitada [10]. Testide käivitamine käsurealt on kasulik siis, kui on vaja teostada lõpliku kontrolli rakenduse installija automatiseeritud paketi loomise protsessis.

6 Tulemused

Töö käigus oli loodud automaatsed testid Bait Partneri UWP rakenduses, mis võimaldavad kiiresti kontrollida programmi peamist funktsionaalsust.

Praegune rakenduse versioon lahendab arendamise ja testimise probleeme. Loodud automaattestid katavad suurt osa programmi funktsionaalsusest ja võimaldavad kiiresti kontrollida konkreetse koodijupi või sellega omavahel seotud komponentide töövõimet pärast muudatusi, mis lihtsustab ja kiirendab arendamist ning annab arendajale kindlustunde, et muudatused ei ole midagi programmi teistes osades rikkunud. Samuti on parandatud rakenduse arhitektuur ja selle puudused eemaldatud, nii et programmi mooduleid saab kasutada väga paindlikult ja neid on lihtsam testimiseks kasutada.

Uue testkeskkonnaga on kõik testimisse investeeritud ressursid uskumatult optimeeritud. Praegune lähenemine vähendab drastiliselt rakenduse väljatöötamise ja testimise aega. Ülejäänud ressursse saab suunata rakenduse kvaliteedi parandamiseks, koodi ümberehitamiseks ja uute funktsioonide arendamiseks.

Kuna rakenduse arendus ei ole veel lõpuni viidud ja ettevõttel on juba plaanid rakenduse edasiseks arendamiseks ja teostamiseks, aitavad selle lõputöö tulemused kaasa edasisele arengule. Praegusse testimiskeskkonda saab integreerida kõik programmis ilmuvad uued teenused.

Kokkuvõte

Ettevõttel Bait Partner OÜ on arendamisel UWP rakendus, mille abil arstid saavad patsiendiga seotud meediafaile vaadata ja annoteerida. Arendatav rakendus vajab automaattestimise võimalust, mis aitab lihtsustada programmi arendamist ning tuvastada ja vältida olulist osa vigadest enne toote kasutajani jõudmist

Selle lõputöö eesmärk oli automaatsete testide loomine ja arhitektuuri parandamine Eesti ettevõte Bait Partneri poolt välja töötatud UWP rakenduse jaoks.

Probleemid, mis oluliselt raskendavad arendusprotsessi on halvasti teostatud rakenduse arhitektuur ja väga väike osa testidega kaetud programmi funktsionaalsusest. Seetõttu esines ebamugavusi arendamisel ning esines rakenduse kvaliteedi ja turvalisuse kahanemist. Töös kirjeldati rakenduse arhitektuuri probleeme ja selle lahendamise viise. Autor analüüsis erinevaid automaatse testimise meetodeid ja tööriistu, mida hiljem kasutas töö praktilise osa tegemiseks.

Töö tulemusena suurendati programmi põhifunktsioonide kaetust võrreldes algse lahendusega ja parandati rakenduse arhitektuuri vead, rakendades korrektselt MVVM-i rakenduse malli ja kasutades sõltuvuse sisestamise (*dependency injection*) põhimõtteid moodulite vaheliste linkide teostamisel.

Selles töös arendatud automatiseeritud testid aitasid lihtsustada rakenduse arengut, ning aitasid tuvastada ja vältida olulist osa vigadest.

Kasutatud kirjandus

[1] „Overview of Visual Studio | Microsoft Docs,“ Microsoft, [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019. [Kasutatud 21 04 2021].

[2] „Run unit tests with Test Explorer,“ Microsoft, [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/visualstudio/test/run-unit-tests-with-test-explorer?view=vs-2019. [Kasutatud 20 04 2021].

[3] „Dependency Injection for Dummies - DZone,“ DZone, [Võrgumaterjal]. Available: https://dzone.com/articles/dependency-injection-dummies#:~:text=Dependency%20injection%20is%20a%20very,outside%20of%20the%20object%20itself.. [Kasutatud 21 04 2021].

[4] „Практическое введение во внедрение зависимостей (Dependency Injection),“ [Võrgumaterjal]. Available: https://webdevblog.ru/prakticheskoe-vvedenie-vo-vnedrenie-zavisimostej-dependency-injection/. [Kasutatud 21 04 2021].

[5] „Test Doubles — Fakes, Mocks and Stubs. | by Michal Lipski | Pragmatists,“ Pragmatists, [Võrgumaterjal]. Available: https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da. [Kasutatud 21 04 2021].

[6] „Mocking Solution with Unit Testing,“ Telerik, [Võrgumaterjal]. Available: https://www.telerik.com/products/mocking/unit-testing.aspx. [Kasutatud 23 04 2021].

[7] „FakeItEasy - It's faking amazing,“ FakeItEasy , [Võrgumaterjal]. Available: https://fakeiteasy.github.io/. [Kasutatud 03 02 2021].

[8] „Data binding and MVVM - UWP applications,“ Microsoft, [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/windows/uwp/data-binding/data-binding-and-mvvm. [Kasutatud 23 04 2021].

[9] „The big MVVM Template,“ CodeProject, [Võrgumaterjal]. Available: https://www.codeproject.com/Articles/768427/The-big-MVVM-Template. [Kasutatud 22 04 2021].

[10] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

[11] „Implementation of Dependency Injection Pattern in C#,“ [Võrgumaterjal]. Available: https://www.dotnettricks.com/learn/dependencyinjection/implementation-of-dependency-injection-pattern-in-csharp. [Kasutatud 21 04 2021].

[12] „Dependency injection in .NET | Microsoft Docs,“ Microsoft , [Võrgumaterjal]. Available: https://docs.microsoft.com/en-Us/dotnet/core/extensions/dependency-injection. [Kasutatud 21 04 2021].

[13] „Arrange Act Assert | JustMock Documentation,“ Telerik, [Võrgumaterjal]. Available: https://docs.telerik.com/devtools/justmock/basic-usage/arrange-act-assert. [Kasutatud 22 04 2021].

[14] D. T. Andrew Hunt, Pragmatic Unit Testing: In Java with JUnit, 2003.

[15] M. S. Steven van Deursen, Dependency Injection Principles, Practices, and Patterns, 2019.

[16] „VSTest.Console.exe command-line options - Visual Studio | Microsoft Docs,“ Microsioft, [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/visualstudio/test/vstest-console-options?view=vs-2019. [Kasutatud 02 05 2021].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Jevgeni Sõritski

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Automaattestide arendamine meditsiinirakenduse näitel", mille juhendaja on Nadežda Furs.
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

17.05.2021

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.