Tampereen yliopisto

Heidi Poels

LEGACY-OHJELMISTON MODERNISOINTI

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2022

TIIVISTELMÄ

Heidi Poels: Legacy-ohjelmiston modernisointi
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Joulukuu 2022

Tänä päivänä suurimmassa osassa organisaatioita on käytössä organisaation toiminnan kannalta tärkeä ohjelmisto. Jos organisaatio on perustettu vuosikymmeniä sitten, ohjelmisto on jo ikääntynyt ja sisältää tarpeettomia osia. Digitalisaation myötä ohjelmistojen määrän on kasvanut räjähdysmäisesti, ja ohjelmiston ylläpito on tärkeämpää kuin koskaan. Legacy-ohjelmisto on vanhahko ohjelmisto, joka ei enää suoriudu tarkoitetusta tehtävästä toivotulla tavalla. Legacy-ohjelmiston tunnusmerkkejä ovat ikä, laajuus ja hitaus. Iältään legacy-ohjelmisto on 20–30 vuotta vanha, ja koodikannasta osa koodista on jo poistunut käytöstä. Toinen tunnistava tekijä on koodikannan laajuus. Laajuuteen vaikuttaa esimerkiksi vanhentunut koodi, jota ei olla poistettu koodikannasta. Viimeisenä tunnusmerkkinä on legacy-ohjelmiston hitaus. Koska koodikanta on laaja ja ylimääräistä koodia on paljon, ohjelmiston käyttö hidastuu.

Tutkielma on kirjallisuuskatsaus ja tutkin legacy-ohjelmiston tunnistavia tekijöitä, sekä miten legacy-ohjelmistoa voi modernisoida. Tutkimuskysymykseni on ”Miten legacy-ohjelmisto tunnistetaan ja modernisoidaan?” Tutkielmassa käytetty aineisto on haettu Andor-tietokannasta ja ScienceDirect-tietokannasta. Mukana on myös muutama ei-tieteellinen lähde terminologian selittämistä varten.

Lehman kirjoitti 1990-luvulla kahdeksan ohjelmiston evoluutioon liittyvää sääntöä. Sääntöjen avulla ohjelmiston parissa työskentelevä voi tunnistaa ohjelmiston käyttäytymistä, ja miten evoluutio näyttäytyy ohjelmistossa. Nämä säännöt ovat nykypäivänäkin ajankohtaisia. Useat työn lähteistä viittaavat Lehmanin ohjelmiston evoluution sääntöihin. Lehmanin sääntöjen perusteella todetaan ohjelmiston olevan jatkuvasti muuttuva entiteetti. Ohjelmistoa tulee modernisoida sen elinkaaren aikana niin käyttäjätyytyväisyyden kuin ylläpidettävyyden kannalta. Legacy-ohjelmistoa ei myöskään voi poistaa, koska legacy-ohjelmistossa on usein liiketoiminnan kannalta tärkeää tietoa ja logiikkaa. Ilman kyseistä logiikkaa liiketoiminta ei toimi.

Legacy-ohjelmiston modernisointi on täten organisaatiolle tärkeä askel. Legacy-ohjelmiston modernisointi tapahtuu valitsemalla organisaatiolle ja kehitystiimille sopivat menetelmät. Tämän tutkielman perusteella suosittelen koodin refaktorointia ja ketterän työtavan omaksumista organisaatiossa. Tutkielma esittelee kaikkiaan neljä menetelmää, joista toimivin menetelmä saadaan yhdistelemällä kahta tai useampaa menetelmää organisaatiolle sopivaksi.

Avainsanat: legacy-ohjelmisto, legacy, perinnejärjestelmä, ohjelmiston modernisaatio, agile, ketteryys.

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck – ohjelmalla.

SAMMANFATTNING

Heidi Poels: Modernisering av legacy-mjukvara
Kandidatavhandling
Tampereen yliopisto
Program i datavetenskap
December 2022

Idag använder en stor del organisationer mjukvara som är viktig för organisationens verksamhet. I organisationer som grundades för årtionden sedan, är mjukvaran redan föråldrad och innehåller onödiga delar. Med digitaliseringen har mängden mjukvara växt exponentiellt, och underhåll av mjukvara är viktigare än någonsin. En legacy-mjukvara är en föråldrad mjukvara, som inte längre utför de uppgifter mjukvaran var avsedd att utföra. Utmärkande för legacy-mjukvara är ålder, tröghet och omfattning. Legacy-mjukvaran är 20–30 år gammal och en del, eller all, kod i databasen är föråldrad. En annan identifierande faktor är kodbasens omfattning. Omfattningen påverkas till exempel av föråldrad kod, som inte har tagits bort från kodbasen. En äldre mjukvara är också långsam, vilket är ytterligare en identifierande egenskap. Eftersom kodbasen är omfattande och det finns mycket extra kod, är användningen av mjukvaran trög.

Denna avhandling är en litteraturgenomgång och undersöker identifierande faktorer för äldre mjukvara, och hur äldre mjukvara kan moderniseras. Forskningsfrågan är "Vad identifierar en legacy-mjukvara och hur moderniseras det?” Materialet som används i avhandlingen har hämtats från hämtats från databaserna Andor och ScineceDirect. I avhandlingen har även några icke-vetenskapliga källor använts som stöd för att definiera terminologin.

På 1990-talet skrev Lehman åtta regler relaterade till utvecklingen av mjukvara. Med hjälp av dessa regler kan personer, som arbetar med legacy-mjukvara känna igen programvarans beteende, och hur utvecklingen har skett. Dessa regler är fortfarande relevanta idag. Flera källor hänvisar till Lehmans regler för mjukvarans utveckling. Enligt Lehmans regler är mjukvara en ständigt ändrande enhet. Programvaran måste moderniseras under sin livscykel, både på grund av tillfredställelse hos användare och underhållbarhet.

Legacy-mjukvara kan inte heller raderas bort, eftersom legacy-mjukvara ofta innehåller information och logik som är viktig för organisationens verksamhet. Utan den logiken fungerar inte företaget. Att modernisera legacy-mjukvaran är alltså ett viktigt steg för organisationen. Legacy-mjukvarans modernisering sker genom val av metoder som passar organisationen och utvecklingsteamet. Baserat på denna avhandling rekommenderas en refaktorering av koden och ett agilt arbetssätt i organisationen. Avhandlingen presenterar totalt fyra metoder, från vilka den mest effektiva nås genom att kombinera två eller flera metoder för att passa organisationen.

Nyckelord: legacy mjukvara, legacy, legacy system, modernisering av mjukvara, agil, agility.

Originaliteten i denna publikation har kontrollerats med Turnits OriginalityCheck-program.

SISÄLLYSLUETTELO

1 Johdanto **1**

2 Tutkimusmenetelmä **2**

3 Ohjelmiston kehitysvaiheet **3**

4 Legacy-ohjelmiston tunnistaminen **5**

5 Legacy-ohjelmiston modernisointi **7**

5.1 Käärintäteknologiat 8

5.2 Muutosvetoinen kehitys 9

5.3 Iteratiivinen prosessi 10

5.4 Prosessikeskeinen kehitys 10

6 Keskustelu ja yhteenveto **13**

Lähdeluettelo **16**

1 Johdanto

Tämän tutkielman avulla haluan tutkia legacy-ohjelmiston tunnusmerkkejä ja ehdottaa ratkaisua legacy-ohjelmiston modernisointiprosessiin. Tutkielman menetelmä on kirjallisuuskatsaus. Legacy-ohjelmistolla tarkoitetaan ohjelmistoa, jonka alkuperäisestä kehityksestä on kulunut vuosia (Merriam-Webster, 2022). Ajan saatossa ohjelmistoon on lisätty toiminnallisuuksia ja tehty korjauksia (Rajlich, 1997). Tämä johtaa koodikannan paisumiseen tilanteeseen, jolloin koodikannan ylläpito on työlästä, jopa mahdotonta (Rajlich, 1997). Tämän perusteella voisi päätellä, että legacy-ohjelmistolla tarkoitetaan siis ohjelmistoa, jonka alkuperäisestä kehityksestä on kulunut aikaa, eikä ohjelmisto täytä enää nykyisiä vaatimuksia ohjelmiston toiminnasta. Ohjelmisto kehitetään organisaation ydintoiminnan ympärille (Colosimo, 2009). Tämän takia legacy- ohjelmistot ovat organisaation selkäranka, ja niissä yhdistyy kaikki organisaation tieto yhteen paikkaan (Colosimo, 2009). Jos ohjelmisto lakkaa toimimasta, koko organisaatio kärsii tästä (Colosimo, 2009). Tyypillinen ratkaisu koodin modernisointiin voi olla vanhan ohjelmiston täydellinen hylkääminen, mutta tämä on harvoin mielekäs ratkaisu juuri legacy-ohjelmiston tärkeyden takia (Colosimo, 2009).

Suunnitellaan siis toinen ratkaisu muutoksen muodossa, toisin sanoen ohjelmiston modernisointia. Modernisoinnilla tarkoitetaan koodin muuntamista nykyaikaisempaan muotoon. Modernisoinnin mukana tulevat koodimuutokset vaihtelevat paljon. Pienimmillään muutos voi olla yksittäisen metodin refaktorointi. Suurimmillaan muutos voi olla laajamittainen uudelleenrakentaminen. Legacy-ohjelmiston laajuuden takia yhdentyyppinen muutostapa harvoin riittää. Koodiin kohtaan A voi riittää nopeahko refaktorointi, kun taas kohtaan B pitää suunnitella arkkitehtuuri ja toiminta kokonaan uudestaan. Usein modernisaation päätöksien takan on tarve siirtää legacy-ohjelmiston nykypäiväisempään infrastruktuuriin kilpailukyvyn säilyttämiseksi. (Colosimo, 2009)

Yksi vaikeuttava tekijä legacy-ohjelmien modernisoinnissa on sanasto. Kaksi yleistä termiä, joita käytetään usein vaihtovuoroisesti, on ylläpito ja evoluutio. Nämä eivät kuitenkaan tarkoita samaa asiaa ohjelmiston kehityksessä. Ohjelmiston ylläpidolla tarkoitetaan pienien virheiden (ns. bugien) korjaamista, kun evoluutiolla puolestaan tarkoitetaan ohjelmiston jatkuvaa kehitystä yksinkertaisemmasta ja huonommasta tilasta parempaan tilaan. Ylläpitoon liittyy kaikki tukitoimet, joita suoritetaan ohjelmistoon toimituksen jälkeen. Evoluutioon liitetään kaikki toimet, jotka muuttavat ohjelmiston rakennetta tai toiminnallisuutta parempaan tai toimivampaan suuntaan, esimerkiksi käytettävyyden lisääminen tai uuden toiminnallisuuden lisäys. (Tripathy, 2014) Tämän tutkielman puitteissa tutkin ohjelmiston kehitystä ohjelmiston evoluution kannalta. En sisällytä ylläpitoon liittyviä kohtia, koska mielestäni ylläpito kuuluu ohjelmiston huoltoon riippumatta sen iästä.

Toinen vaikeuttava tekijä legacy-ohjelmiston modernisoinnissa on muistaa katsoa kokonaisuutta. Ohjelmisto on ainiaan muuttuva kokonaisuus, eikä ohjelmiston muuttuminen lopu modernisaatioprosessin jälkeen. Lehman (1996/2005) on tutkinut ohjelmiston evoluutiota ja kehittänyt tutkimuksen perusteella kahdeksan sääntöä, jotka kuvaavat ohjelmiston evoluutioprosessia. Nämä kahdeksan sääntöä ovat tärkeitä vielä tänäkin päivänä, koska ohjelmistoja on valtava määrä, ja on tärkeä ymmärtää, miten ohjelmisto kehittyy alusta loppuun. Ymmärryksen avulla kehittäjät voivat tehdä tietoon perustuvia päätöksiä ohjelmiston kehityksen suhteen, arvailun sijaan. (Lehman, 1996/2005)

Erilaisia töitä legacy-ohjelmistojen modernisoinnista on olemassa runsaasti, mukaan lukien tutkimuspapereja, teknisiä raportteja ja kirjoja. Olemassa oleva kirjallisuus keskittyy usein tiettyyn yksityiskohtaan, eikä kokonaisvaltaisia teoksia ole kovinkaan paljon. Mielestäni on tärkeä ymmärtää ohjelmisto kokonaisuutena. Kokonaisuuden ymmärrettyä modernisointiprosessi on helpompi pilkkoa eri vaiheisiin, jolloin lopputulos on kokonaisuus erillisien palojen sijaan.

Tämän tutkielman tavoitteena on tutkia legacy-ohjelmistojen tunnusmerkkejä, sekä miten legacy-ohjelmistoja voidaan lähteä modernisoimaan. Tutkimuskysymykseni on ”Miten legacy-ohjelmisto tunnistetaan ja modernisoidaan?”. Päätin sisällyttää neljä eri prosessia koodin modernisoimiseen. Tämän tutkielman ulkopuolelle jäi päivityskeinot, kuten infrastruktuurin päivitys ja järjestelmän siirtäminen pilvipohjaiseen ympäristöön. Tämän tutkielman puitteissa keskityn ohjelmiston modernisaatioon. Tutkielman lopputulos on, että yhdistelemällä useita modernisaatiotapoja, riippuen ohjelmiston tilasta ja organisaation resursseista, saadaan paras prosessi ohjelmiston modernisaatiota varten. Mielestäni paras yhdistelmä on refaktorointi scrum-työskentelytavan kanssa.

Luvussa 2 esittelen tutkielman kirjoittamiseen käytetyt tutkimusmenetelmät ja avaan hakusanoja ja käytettyjä tietokantoja. Luvussa 3 esittelen ohjelmiston evoluutioon liittyviä vaiheita. Luvussa 4 teen katsauksen legacyn tunnusmerkkeihin. Luvussa 5 tutkin eri modernisaatiotapoja. Luvussa 6 esittelen ratkaisuehdotuksen tutkimuskysymykseen.

2 Tutkimusmenetelmä

Tämä tutkielma on kirjallisuuskatsaus. Olen hakenut kirjallisuutta kahdesta tietokannasta, Andor ja ScienceDirect.

Tein ensimmäiset haut Andor-tietokantaan hakusanoilla ”legacy software”, ”software evolution”, ”software lifecycle AND legacy” ja lifecycle management”. Andorista löytyi paljon lähteitä ja jouduin varhain suorittamaan karsintaa lähteiden joukosta. Löydettyäni ensimmäiset lähteet siirryin ScienceDirect-tietokantaan. Andorin hakujen kokemuksella osasin jo arvioida mitkä hakusanat tuottivat lähimmät tulokset. Hain hakusanoilla ”legacy software”, ”idea to legacy”, ”software lifecycle”, ”software lifecycle

management” ja “lifecycle management” AND legacy. Näiden alustavien hakujen perusteella sain selkeän kuvan siitä, mitä aiheesta on jo kirjoitettu. Lähteen sisällyttämiskriteeri oli ensimmäisen haun aikana otsikon ja tiivistelmän sisällön yhdistelmä. Jos nämä vaikuttivat lupaavilta, sisällytin lähteen.

Alustavan haun jälkeen aloitin lähteiden karsimisen. Karsinta tapahtui johdannon ja yhteenvedon lukemisella, ja jos lähde ei sopinut aihepiiriini, karsin sen lähteen pois. ScienceDirect-tietokannasta löydettyjen lähteiden perusteella kävin uudemman kerran Andorissa hakemassa täsmähaulla lähteitä, joita oli käytetty muissa artikkeleissa. Hain tällöin kirjoittajan nimellä tai tutkimusartikkelin otsikolla. Yksi esimerkki on Lehmanin (2005) ohjelmiston evoluutioon liittyvä tutkielma, johon viitattiin toisessa tutkielmassa.

Ensimmäisessä haussa en katsonut vuosilukuja kovin tarkkaan. Aloin kiinnittää huomiota kirjoitelmien vuosilukuihin vasta karsintavaiheessa. Pyrin sisällyttämään ainoastaan lähteitä, jotka on kirjoitettu viimeisen 7–10 vuoden sisällä. Ainoat poikkeukset ovat taustatietoon liittyvää lähdettä, joissa mielestäni on selkeästi kuvattu aihepiiriin liittyvää perustietoa.

Löydettyäni relevantin lähteen, joka sopii sisällyttämiskriteereihin, aloitin lähteen tarkemman lukemisen. Sisällyttämiskriteerit olivat 1) lähde olisi tieteellinen (poissulkien sanojen ja käsitteiden selitykset), 2) lähde käsittelee legacy-ohjelmiston modernisointiprosessia, 3) lähde kuvaa legacy-ohjelmiston tunnusmerkkejä. Lopullisista lähteistä 17/25 lähteestä sopi sisällyttämiskriteereihin.

Luin kolmannella kerralla artikkelin otsikot ja jokaisen otsikon sisällön ensimmäiset lauseet. Halusin tässä vaiheessa varmistaa artikkelin sisällön relevanttiuden tutkimuskysymykseeni, mutten vielä halunnut syventyä artikkelin yksityiskohtiin. Tässä vaiheessa alleviivasin tärkeitä kohtia, jos sellainen tuli vastaan. Jos huomasin alleviivaavani useampia kohtia jo alussa, sain käsityksen, että lähteestä on aiheeseeni hyötyä ja lähde oli virallisesti sisällytetty. Tässä vaiheessa viisi lähdettä karsiutui pois, ja minulle jäi 12 lähdettä käteen. Vasta neljännellä lukukerralla tutustuin syvällisesti tekstiin, kopion alleviivatut kohdat tekstitiedostoon ja aloitin referoinnin. Syvällisen tutustumisen jälkeen hain vielä täydentäviä lähteitä. Käytin samoja sisällyttämiskriteereitä, ja samaa lukutyyliä täydentäviin lähteisiin.

3 Ohjelmiston kehitysvaiheet

Lehman (2005) on kirjoittanut 1970–1990 välisenä aikana kahdeksan sääntöä ohjelmiston evoluutioon liittyen ja miten nämä säännöt vaikuttavat ohjelmistoon. Lehmanin säännöt liittyvät ohjelmiston modernisaation, koska ohjelmisto ei ole staattinen entiteetti. Kun ohjelmistoa modernisoidaan, pitää muistaa ohjelmiston evoluution myös tämän modernisaation jälkeen. Lehmanin säännöt ovat aiheellisia vielä tänäkin päivänä koska ne kuvaavat ohjelmistoa kokonaisuutena. Sääntöjen kehityksen jälkeen niitä on tutkittu ja

validoitu ohjelmistoalalla, ja ovat osoittaneet pitävänsä vielä pakkansa. Tämän tutkielman puitteissa esimerkiksi Tripathy (2014) ja Bennett & Rajlich (2000) ovat viitanneet Lehmanin sääntöihin.

Ensimmäinen sääntö on **jatkuva kehitys:** ohjelmisto kehittyy jatkuvasti. Jos ohjelmisto ei kehity muuttuu se vähitellen vähemmän tyydyttäväksi. Lehmanin ensimmäinen sääntö mukautuu orgaanisen olion kehitykseen. Ainoana erona on ohjelmiston kehityksen jatkuva palautesilmukka. Palaute ohjelmistossa tulee ulkopuolelta, kun organismin muutostarve tulee organismin sisältä. (Lehman, 1996/2005)

Toinen sääntö, **kasvava monimutkaisuus**, mukailee Lehmanin mukaan termodynamiikan toista lakia. Kun ohjelmistoa muutetaan, ja muutoksen päälle tulee muutos, jonka päälle tulee muutos, johtaa tämä ohjelmiston entropiaan. (Lehman, 1996/2005) Entropia tarkoittaa tilaa, jossa ohjelmisto on epäjärjestyksessä, kaaoksessa (Merriam-Webster, 2022).

Kolmas sääntö, **itsesäätely**, viittaa kehitystiimiin, joka kehittää organisaation ohjelmistoa. Tiimin edut ja tavoitteet eivät ainoastaan koske ohjelmiston valmistumista, vaan ulottuu paljon pidemmälle. Esimerkkinä yrityksen johto on elin, jota kiinnostaa kehitystiimin edistymien. Johto on tämän takia luonut eri mittareita, joilla kehityksen edistymistä voidaan pitää silmällä. (Lehman, 1996/2005)

Neljännessä säännössä, **vakauden säilyttäminen organisaatiossa**, otetaan ajankäyttö huomioon, ja kuinka paljon aikaa kehitykseen käytettään. Yleinen olettamus on, että ohjelmiston kehitykseen käytetty aika määrätään johdon toimesta. Vaikka johto valvoo kehityksen toimintaa, kehitys on usein ulkoisten voimien rajoittama, kuten ammattitaidollisen henkilöstön saaminen projektiin. (Lehman, 1996/2005)

Viides sääntö, **perehtyneisyys**, ottaa huomioon ohjelmiston tarpeelliset muutokset. Mitä enemmän muutoksia ja lisäyksiä ohjelmistoon tulee, sitä vaikeampi asianomaisten on pysyä mukana ohjelmiston sen hetkisen toiminnan tilasta, sekä tietää mitä heiltä vaaditaan. Työn edistymisen nopeuteen ja laatuun vaikuttaa kuinka nopeasti osallistujat saavat tarvittavan tiedon ohjelmiston toiminnasta. (Lehman, 1996/2005)

Vaikkakin kuudes, **jatkuva kasvu**, sääntö vaikuttaa päältä päin samanlaiselta kuin ensimmäinen sääntö, on näillä kahdella merkittävä ero. Ensimmäinen sääntö liittyy kehittyneen ohjelmiston toimintaan ja käyttäjän käytön väliseen kuilun. Kun ohjelmisto ei enää toimi käyttäjän haluamalla tavalla syntyy kehityksen ja muutoksen tarve. Kuudes sääntö puolestaan koskee ohjelmiston kehityksen alussa saatujen kuvausten sisällyttämistä, tai poisjättämistä, ohjelmistoon. Eri rajoitusten takia osa vaatimuksista jätetään pois ja osia muutetaan. Kun ohjelmisto on päässyt käyttäjille asti käyttöön ovat juuri nämä poisjääneet tai muutetut toiminnallisuudet usein turhautumisen takana. (Lehman, 1996/2005)

Seitsemäs sääntö, **laadun heikkeneminen**, koskee ohjelmisto laadun takaamista. Ohjelmisto, joka on aikaisemmin toiminut halutulla ja tyydyttävällä tavalla, alkaa jossain vaiheessa käyttäytymään odottamattomalla tavalla. Heikentyminen liittyy jatkuvan kehityksen vaatimaan rajallisiin resursseihin. Koska resursseja on aina rajallinen pitää valita mitä sisällytetään ohjelmistoon ja mitä ei, mitä ongelmia ratkotaan ja mitä ei. Myös reaalimaailman jatkuva kehitys vaikuttaa ohjelmiston väistämättömään heikkenemiseen. (Lehman, 1996/2005)

Lehmanin kahdeksas ja viimeinen sääntö on **palaute**. Vaikkakin lisätty myöhemmin kuin muut säännöt, se ei ole löydöksenä uusi. Palautteen laki täsmentää miten mahdollisia ongelmia ja ilmiöitä voi löytää, täsmentää ja tutkia palautteen avulla. Kun palautetta kerätään isommaksi massaksi, voidaan muodostaa käyttäytymisteoria ja käyttäytymisteorian avulla ilmiötä voidaan tarkentaa, testata ja parantaa lisähavainnoilla. (Lehman, 1996/2005)

4 Legacy-ohjelmiston tunnistaminen

Legacy-ohjelmiston ymmärtäminen vaatii yli puolet tarvittavasta työpanoksesta ohjelmiston kehittämisessä, jonka takia legacy-ohjelmiston tunnistaminen on tärkeää. Tavallisin syy ohjelmiston siirtyminen legacy-ohjelmistoksi on ohjelmiston vaatimuksien muuttuminen. Jotta legacy-ohjelmistoa voi kehittää, vaatii se ensin ymmärrystä. Vain ohjelmistoa, joka on täsmällisesti ymmärretty, voidaan ylläpitää, kehittää ja uudelleen käyttää. (Rajlich, 1997) Kun tarkastelemme Lehmanin (1996/2005) sääntöjä pitäen näkökulman legacy-ohjelmistoissa on selvä, että legacy-ohjelmiston kanssa työskentely on aikaa vievää ja työlästä. Vaikkakin Rajlichin (1997) tutkimus on iäkäs, toistuu Rajlichin tulos esimerkiksi Abdellatif (2021) tutkimuksessa.

Ajatus vanhentuneen ohjelmiston poistamisesta kuulostanee alkuun mielekkäältä askeleelta, jolloin uudet tarpeet voidaan alusta asti saada mukaan. Valitettavasti tämä on harvoin mahdollista. Legacy-ohjelmisto sisältää usein piilossa olevaa hiljaista tietoa (Abdellatif, et.al., 2021). Liiketoiminta ei toimi ilman tätä tietoa ja legacy-ohjelmisto jää elämään (Abdellatif, et.al., 2021). Vaikka legacy-ohjelmisto ei toimi optimaalisesti ja vaatii paljon kustannuksia ylläpitoon, pitää sen kanssa kuitenkin työskennellä (Abdellatif, et.al., 2021).

Riederer (2020) yhtyy Abdellatifin kantaan. Kun ohjelmistoa on käytetty usean vuoden, joskus jopa vuosikymmenen ajan, ohjelmisto on hidastunut ja jäykistynyt. Tällöin se ei enää ole yhteensopiva uusien teknologioiden kanssa, kuten käyttökelpoisuus nettisovelluksessa. Houkutus on suuri korvata vanha ohjelmisto täysin uudella, mutta tähän liittyy suuria kuluja, niin taloudellisia kuin resursointikuluja. Kuluja ei myöskään koidu ainoastaan uuden ohjelmiston kehitysvaiheessa, vaan myös myöhemmin, kun työntekijöitä pitää uudelleen kouluttaa uuden ohjelmiston käytössä. Uuden ohjelmiston opettelussa voi

syntyä myös muutosvastarintaa, koska vanha ohjelmisto oli jo hyväksytty sellaisenaan ja käyttöön oli totuttu. (Riederer, 2020)

Seuraava kysymys nousee pintaan: miksi ohjelmisto ei voi vain pysyä samanlaisena? Edellisessä osiossa esitetty Lehmanin (1996/2005) ensimmäisen säännön mukaan kaikki muuttuu jatkuvasti. Jos ohjelmisto ei seuraa muutoksen virtaa ohjelmistosta tulee progressiivisesti vähemmän tyydyttäviä (Lehman, 1996/2005). Esimerkiksi Ahmad ja muut (2021) on verrannut ohjelmiston kehitystä biologisen organismin kehitykseen. Biologisessa organismissa uusi ominaisuus tulee esille uuden ympäristön asettaman vaatimuksen takia (Ahmad, et al., 2021). Näin tapahtuu myös ohjelmistokehityksessä, jossa uusien vaatimusten takia ohjelmisto kehittyy uudeksi versioksi (Ahmad, et al., 2021).

Toinen syy ohjelmiston modernisoinnin pakottavaan tarpeeseen on käyttäjätyytyväisyys. Jos ohjelmistoa ei modernisoida, eikä ajan kuluessa modernisoida, jää ohjelmisto unholaan ja käyttäjät siirtyvät muitten, nykypäiväisempien, palvelujen pariin. Tämän huomaa varsinkin kuluttajaohjelmistoissa. Kun kuluttajaohjelmistoa modernisoidaan, onnistuneen modernisoinnin edellytys on käyttäjäpalautteen ottaminen huomioon ja käyttäminen. Kehittäjät eivät tiedä mitä loppukäyttäjät loppupeleissä haluavat tai tarvitsevat, jolloin palaute on elintärkeää. (Aljannan, 2020).

Voimme edellisen perusteella todeta, että legacy-ohjelmiston tunnistaminen on yrityksen edun mukaista, jottei resursseja tuhlata. Rajlich (1997) on koonnut lista tunnusmerkeistä, joista legacy-ohjelmiston tunnistaa:

- ohjelmisto on otettu käyttöön useita vuosia sitten
- ohjelmisto tekniikka on vanhentunut
- ohjelmisto rakenne on heikentynyt
- ohjelmisto on investoitu paljon, niin rahaa kuin aikaa
- ohjelmisto on liiketoimintasääntöjä, joita ei ole saatavilla muualta
- ohjelmisto on vaikeasti korvattavissa
- ohjelmiston alkuperäisiä kirjoittajat eivät ole tavoitettavissa

Iso osa olemassa olevista ohjelmistoista ovat verkko- tai työasemapohjaisilla alustoilla. Kannettavien laitteiden, kuten puhelimien ja mobiililaitteiden, räjähdysmäinen kasvu on vaatinut legacy-ohjelmistojen uudelleenkäyttöä ja modernisointia. Legacy-ohjelmistoa ei aina ole mahdollista, tai järkevää, korvata uudella. Legacy-ohjelmistossa voi olla esimerkiksi liiketoiminnalle tärkeää logiikka, jota ei voi helposti korvata. Sen sijaan, että vanha ohjelmisto korvataan uudella, voidaan ohjelmisto modernisoida sisällyttämällä ydinlogiikka ja lisäämällä tarvittavat uudet ominaisuudet. Ahmadin (2021) tutkimuksen tulosten perusteella, tämä tuottaisi kontekstiherkän, helposti päivitettävän ja interaktiivisen koodin. (Ahmad et al., 2021)

Kuten mainitsin johdannossa, ylläpitoa ja evoluutio on perinteisesti nähty samana asiana (Tripathy, 2014). Ylläpito on kuitenkin osa ohjelmiston elinkaareen kuuluvaa toimintaa (Pohjalainen, 2014). Ylläpidon myötä varmistetaan ohjelmiston toimivuus nykyhetken käytössä (Pohjalainen, 2014). Ylläpitoon kuuluu pienien virheiden korjaus, testien kirjoittamista tai parantamista ja riippuvuuksien vähentäminen mahdollisuuksien mukaan (Pohjalainen, 2014). Bennett & Rajlich (2000) lisäävät, että ylläpidon voi jakaa suurpiirteisesti neljään vaiheeseen: mukautuva (muutokset ohjelmiston ympäristössä), täydentävä (uudet ohjelmiston vaatimukset), korjauksellinen (ongelmien korjaus) ja ennaltaehkäisevä (tulevien ongelmien ennaltaehkäisevä toiminta). Evoluutio puolestaan on toiminta, joka edistää ohjelmiston toimintaa parempaan suuntaan, jossa on uusia toiminnallisuuksia ja vastaa uusiin käyttäjien tarpeisiin (Tripathy, 2014). Bennett & Rajlich (2000) löydösten perusteella evoluution toteuttaminen on suuri ongelmakohta. Kehitystiimin työmäärästä vain 21 % käytetään ylläpitoon, kuten bugien korjaukseen, ja loput ajasta menee evoluution hallintaan.

Jos ylläpito siis virheellisesti mielletään evoluutioon voi tämä mielestäni johtaa tilanteeseen, jossa teknologioita muutetaan liian usean ja refaktorointeja suoritetaan loputtomasti. Kuten Tripathy (2014) kirjoittaa, ylläpito mielletään vähemmän haluttavaksi työtehtäväksi, jolloin nämä tehtävät annetaan uusille ohjelmoijille. Tämä voi puolestaan johtaa uusien bugien syntymiseen, kun kokematon ohjelmoija yrittää ratkoa olemassa olevaa virhettä. Voimme siis päätellä, että evoluutioon liittyvät tehtävät ovat miellyttävämpiä kehittyneemmälle ohjelmoijalle. Evoluutioon liittyvissä tehtävissä voidaan vaikuttaa käytettäviin teknologioihin, tuleviin ominaisuuksiin ja teknisiin ratkaisuihin. (Tripathy, 2014)

Monet nykyiset ohjelmistot ovat myös viimeisten vuosikymmenien aikana laajentuneet suuriksi kokonaisuuksiksi, joiden kanssa on vaikea tehdä töitä. Suuressa organisaatiossa saattaa olla satojakin kehittäjiä, jotka työskentelevät ohjelmiston kehityksen kanssa ja yhteistyö eri tiimien kesken voi olla vaikeaa. Suuren tiimin suurimmat haasteet ovat hidas vasteaika ilmoitettuun ongelmaan, ajanhallinta ja suunnittelu sekä etääntyminen asiakkaasta. Vasteena näihin ongelmakohtiin organisaatiot kiristävät prosessejaan, jonka seurauksena ylirasitteet lisääntyvät. (Bosch & Bosch-Sijtsema, 2011)
Näiden perusteella voidaan olettaa legacy-ohjelmiston tunnistamisen olevan organisaation edun mukaista, jopa elintärkeää. Seuraavaksi avaan miten legacy-ohjelmistoa voidaan modernisoida.

5 Legacy-ohjelmiston modernisointi

Legacy-ohjelmiston modernisointi vaatii yritykseltä paljon resursseja. Vaatimusten muuttuminen vaatii uuden analyysin tekemistä, jotta uuden vaatimukset löydetään. Koska

legacy-ohjelmisto on vanha ja laaja on vaarana olla huomaamatta tärkeitä toiminnallisuuksia, joka johtaa puutteelliseen toiminnallisuuteen. (Riederer, 2020)

Kun huomataan modernisoinnin tarve, aloitetaan suunnittelulla ja päädytään ideaaliseen kuvaan työprosessista ja lopputuloksesta. Valitettavasti suunnitellut työtavat ja lopputulokset eivät aina saa tukea operatiivisista prosesseista. Lisäksi jo olemassa olevan datan siirto uuteen ohjelmistoon on monimutkainen hanke. (Riederer, 2020)

Kuten mainitsin edellisessä kohdassa, uuden toiminnallisuuden lisääminen on hankalin askel kohti modernisoitua ohjelmistoa. Jos voisimme välttää tämän kompastuskiven, modernisaatioprosessi helpottuisi huomattavasti. Miksi emme siis varmista uuden ohjelmiston suunnitteluvaiheessa, että saamme kaikki tulevaisuudessakin tarvittavat toiminnallisuudet (Bennett, & Rajlich, 2000)? Koska ohjelmointitiimi ei voi sisällyttää ohjelaman jotain, mitä eivät voi kuvitella (Bennett, & Rajlich, 2000). Tämän takia modernisointi on tärkeä vaihe. Uudet toiminnallisuudet voidaan lisätä ilman aiempaa tietoa uuden toiminnallisuuden tarpeesta, mutta tähän tarvitaan vanhempia kehittäjiä, jotka ovat kokeneet ohjelmiston aiemmat versiot (Bennett, & Rajlich, 2000).

Kuten Abdellatif (2021) ja Riederer (2020) kirjoittavat, legacy-ohjelmistoa ei kannata lähteä kokonaan uudistamaan alusta alkaen. Riederer (2020) jatkaa, että ohjelmiston modernisaatio voi usein olla riittävä askel eteenpäin, eikä pyörää tarvitse keksi uudelleen. Yksi Riedererin ehdottamista tavoista on refaktorointi. Refaktorointi on kohdennettu uudistus johonkin tiettyyn kohtaan ohjelmistossa, ja tämä sopii erityisen hyvin ohjelmistoihin, jota pitää muuntaa useasti. Pelkistetysti ajateltuna refaktorointi on lähdekoodin siivoamista. Siivoamisen aikana lähdekoodia yksinkertaistetaan, mutta ei muuteta sen toiminnallisuutta. Refaktoroinnin ansiosta koodia on helpompi lukea ja ymmärtää tulevaisuudessa, ja uusien toiminnallisuuksien lisääminen on suorasukaisempaa. Koodin refaktorointi lisää myös suorituskykyä, kun ylimääräinen koodi poistetaan. (Riederer, 2020)

Toinen tapa ajatella refaktorointa on muuttaa koodin ulkonäköä. Refaktoroinnin tarkoitus ei siis ole lisätä toiminnallisuutta tai muuttaa käyttöliittymän ulkonäköä. Refaktoroinnin avulla voidaan säilyttää ohjelmiston arvo, tai jopa lisätä arvoa. Käyttäjän näkökulmasta arvo nousee, kun käyttäjän kaikki tarpeet ovat täytetty. Käyttäjälle arvokas ohjelmiston on ohjelmisto, joka on virhevapaa ja vasteaika muutokseen on nopea. (Tinetti & Méndez, 2012) Alla tutkin refaktoroinnin lisäksi neljää muuta modernisointitapaa.

5.1 Käärintäteknologiat

Kuten aiemmin on todettu, vanhan ohjelmiston pois heittäminen tuo mukanaan suuria riskejä organisaatiolle, jonka takia se on harvoin varteen otettava vaihtoehto. Aiempien investointien säilyttämiseksi, sekä riskien ja kehityskustannusten vähentämiseksi, vanhojen ohjelmistojen kapselointi käärintäteknologian (eng. ”wrapping technologies”) avulla on varteenotettava vaihtoehto. (Colosimo et.al., 2009)

Käärintäteknologia on käytäntö, jonka avulla eri ohjelmiston osia voidaan kääriä omaan kapseliin. Kääre mahdollistaa erillisten osien toimimisen yhdessä, kun ne omillaan eivät olisi yhteensopivia. Esimerkiksi, on olemassa vanha ohjelmisto, joka on kirjoitettu ohjelmointikielellä, johon ei nykyään ole saatavilla osaamista, mutta vanhalla kielellä kirjoitettu koodi on organisaation toiminnan kannalta elintärkeä. Tämän takia vanhaa koodia ei voida korvata. Käärinnän avulla voidaan kapseloida vanhat koodin osat uudemman kielen sisälle, jolloin modernisointi voidaan toteuttaa modernimmalla kiellellä, ilman legacy-osien menettämistä. (IONOS, 2020)

Ennen uuden teknologian sisäistämistä legacy-ohjelmistoon pitää selvittää ratkaiseeko se ongelman, jota yritetään ratkaista. Lisäksi tulisi tutkia onko käytäntö tai teknologia jo hyväksytty alalla. Tutkimus voi tuottaa haasteita, koska uuden teknologian sisäistymien alalle voi kestään 15–20 vuotta. Jatkuvasti muuttuvalle alalle 15 vuotta on liian pitkä aika odottaa, jonka seurauksena teknologia sisäistetään omaan ohjelmistoon kokeilumielessä. Innovatiivisten teknologioiden tutkiminen olisikin tämän takia tärkeää, jotta organisaatiot voivat tehdä tietoihin perustuvia päätöksiä uusien teknologioiden valinnassa. (Colosimo et.al., 2009)

5.2 Muutosvetoinen kehitys

Muutosvetoinen kehitys on Méndezin (2016) ehdotus uudeksi ketteräksi menetelmäksi, joka olisi uusi lähestymistapa ohjelmistojen ylläpitämisessä ja kehittämisessä. Tämä prosessi perustuu ohjelmiston ytimen periaatteisiin, integroituihin kehitystyökaluihin ja automatisoituihin lähdekoodimuunnoksiin.

Ensimmäiseen kohtaan, ohjelmiston ydinperiaatteet, ovat muutettavuus, monimutkaisuus, aineettomuus ja yhdenmukaisuus. Toiseen kohtaan, integroitu kehitystyökalu, sisältää IDE:n (integrated development environment) käytön (Méndez, 2016). IDE on kehittäjien käyttämä sovellus koodin kirjoittamista varten, joka nopeuttaa koodin kirjoittamisen prosessia. IDE sisältää ne työkalut, joita tarvitaan koodin kirjoittamista, rakentamista, testausta ja paketointia varten. (AWS, 2022). Kolmanteen kohtaan, automatisoituihin lähdekoodimuutoksiin, sisältyy koodimuutoksia, jotka suoritetaan refaktoroinnin avulla (Méndez, 2016).

Tämä ketterä lähestymistapa ottaa vastaan muutoksen työyksikkönä. Työyksikkö ohjaa kehitysprosessia. Kehitysprosessi puolestaan suoritetaan nelivaiheisessa syklissä. Méndezin (2016) näkemyksen mukaan yksi mielenkiintoisimmista tavoista soveltaa muutosvetoista kehitystä tieteellisiin ohjelmistoihin on modernisoida ja jopa rinnastaa peräkkäisiä ohjelmia (engl. ”sequential program”). Nämä ohjelmiston on voitu kirjoittaa 20 tai 30 vuotta sitten ja ne ovat edelleen käynnissä tuotantoympäristöissä. Tämä prosessi on vastakohta perinteiseen, suunnitelmaan pohjautuvaan kehitykseen. Méndezin (2016) uusi tapa perustuu muutosjohtoiseen kehitykseen. Tämän myötä muutokseen pitäisi suhtautua

osana prosessia, sen välttelyn sijaan. Lisäksi yllä mainitut ohjelmiston ydinperiaatteet sisältyvät luonnollisesti Méndezin uuteen kehitysprosessiin. Iso osa ohjelmistoihin tehtävästä työstä n vanhan ohjelmiston kehitystä uuden rakentamisen sijaan. Méndezin näkemyksen mukaan, tämä lähestymistapa on teeskentelemätön. (Méndez, 2016)

Yhdyn Méndezin näkemykseen, varsinkin kun katsomme ohjelmiston evoluutiota jatkuvan muuttumisen kannalta. Ohjelmisto ei ole staattinen asia, vaan uusien vaatimuksien noustessa pintaan vaaditaan modernisaatiota. Ohjelmiston pysyvä poistaminen, kuten aiemmin todettu, on huono vaihtoehto ja tätä tulisi välttää. Muutoksen hyväksymien on yksityiselämässäkin vaikea hyväksyä, ja on ymmärrettävää, että muutosta halutaan välttää. Se voi herättää epämukavia tunteita. Kirjallisuudesta voisi päätellä, että olisi organisaation edunmukaista puhua ja sisällyttää kehittäjät modernisaation prosessiin jo ennen modernisaation aloittamista, jotta muutosvastarinta olisi mahdollisimman vähän.

Refaktorointi on noussut kahdessa otteessa pintaan metodina, joka auttaisi modernisointiprosessia. Riederer (2020) yhtyy Méndezin (2016) näkemykseen siitä, että refaktoroinnin avulla voidaan saavuttaa mielekäs modernisoitu ohjelmisto. Riedereristä poiketen, Méndez (2016) ehdottaa kokonaisvaltaisempaa lähestymistapaa, jossa ei mietitä vain koodin modernisointiin liittyvää menettelytapaa, vaan myös kehittäjien työskentelytapaa.

5.3 Iteratiivinen prosessi

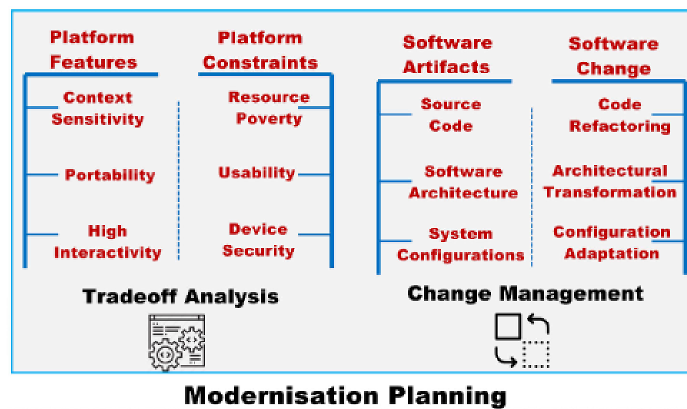
Iteratiiviset mallit jakavat ajatuksen siitä, että kokonaisen ohjelmiston vaatimuksia ei voida täysin ymmärtää, eikä kehittäjät osaa rakentaa koko ohjelmiston yhden vaatimuslistan perusteella. Siksi ohjelmistot rakennetaan versioina, joista jokainen on jalostus edellisen version vaatimuksista ja rakennetta hiotaan ottamalla huomioon käyttäjiltä saatu palaute. Iteratiivisestä prosessista voidaan huomata, etteivät ylläpito- ja kehitystoiminnot ole erillisiä vaiheita, vaan pikemminkin ne ovat tiiviisti kietoutuneet kehitysprosessiin. (Tripathy, 2014)

Esimerkki iteratiivisestä prosessista on työtapa nimeltään Scrum. Scrum metodissa työtä tehdään pienissä osissa paloissa, jotka tuottavat eniten hyötyä käyttäjälle. Scrumissa työskennellään lyhyissä pyrähdyksissä (engl. ”sprint”), jonka aikana toteutetaan sille pyrähdykselle sovitut tehtävät. Pyrähdyksen lopussa tuotos esitellään sidosryhmille ja kirjataan mahdollinen palaute ylös. Tämä prosessi toistetaan, kunnes ohjelmiston kehitys on valmis. (Scrum.org, 2022)

5.4 Prosessikeskeinen kehitys

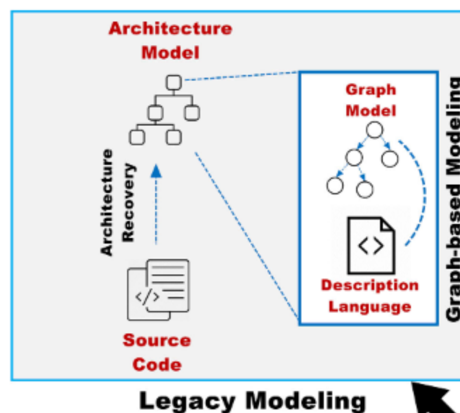
Ahmad ja muut (2021) puolestaan ehdottavat prosessikeskeistä menetelmää legacy-ohjelmiston modernisaatiota varten. Ahmad ja muitten (2021) prosessikeskeinen uudistus sisältää kolme vaihetta, joita tutkimme seuraavissa osissa lähemmin.

Ensimmäinen askel on prosessitoimintojen läpikäynti. Prosessitoimintoihin kuuluu kohdat Suunnittelu (Plan), Mallinnus (Model), Muunnos (Transform) ja Arviointi (Evaluate). **Suunnittelussa** suunnitellaan tuleva muunnostyö. Suunnittelussa huomioidaan asioita, kuten kompromissianalyysi ja muutoksenhallinta. Kompromissianalyysissä arvioidaan alustan asettamat mahdollisuudet ja rajoitukset. Mahdollisuuksia voivat olla esimerkiksi kontekstiherkkyys ja siirrettävyys. Rajoittavat tekijät voivat olla esimerkiksi käytettävyys ja resurssien puute. Tämä askel suoritetaan ihmisen valvonnan alla. Kuvassa 1 Ahmad ja muut (2021) havainnollistaa suunnitteluprosessia. (Ahmad et.al., 2021)



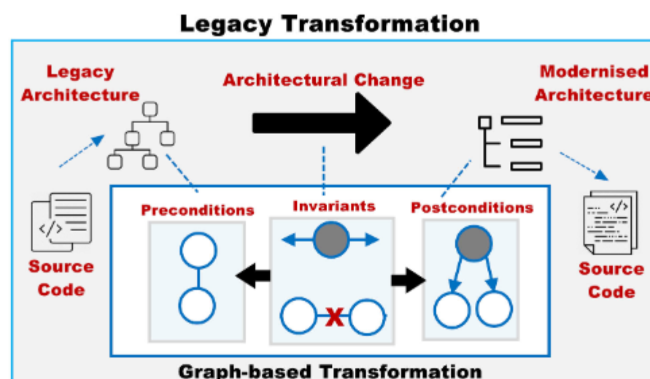
Kuva 1: Suunnitteluvaiheen visualisointi (Ahmad et.al. 2021)

Mallinnuksessa (model) arvioidaan lähdekoodia ja miten legacy-koodia voidaan käyttää uudelleen. Lähdekoodista rakennetaan arkkitehtuurinen graafimallinnus, joka kuvaa nykyistä lähdekoodin tilaa. Tässä vaiheessa voidaan jo käyttää valvottua automaatiota, toisin sanoen ihminen valvoo koneen prosessia. Kone puolestaan voi analysoida olemassa olevan koodin ja luoda arkkitehtuurisen graafin koodista. Graafia hyödynnetään koodin analysoimisessa. Kuvassa 2 Ahmad ja muut. (2021) havainnollistaa mallinnusvaiheen. (Ahmad et.al., 2021)



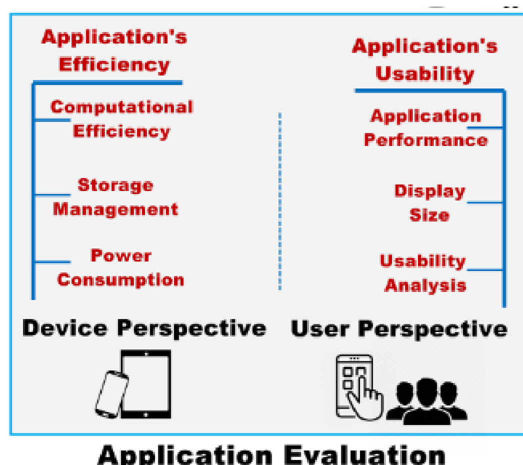
Kuva 2 Mallinnuksen visualisointi (Ahmad et.al. 2021)

Muunnoksessa (transformation) käytetään mallinnusvaiheessa syntynyttä arkkitehtuurista graafia uuden arkkitehtuurin suunnittelua varten. Legacy-arkkitehtuurisen kuvan perusteella voidaan tunnistaa ohjelmiston ennakkoehdot ja muuttumattomat osat. Näiden avulla saavutaan lopputilanteeseen. Lopputilanteessa prosessi on tuottanut modernisoidun arkkitehtuurin, joka sisältää legacy-ohjelmiston keskeiset osat. Tämäkin vaihe toteutuu valvotun automaation kautta. Kuvassa 3 Ahmad ja muut (2021) havainnollistaa muutosvaiheen. (Ahmad et.al., 2021)



Kuva 3 Muunnoksen visualisointi (Ahmad et.al. 2021)

Arvioinnissa (evaluate) arvioidaan nykyaikaistetun ohjelmiston kahta näkökulmaa: laitteen näkökulmasta ja käyttäjän näkökulmasta. Laitteen näkökulmasta arvioidaan ohjelmiston suorituskykyä seuraavin parametrein: laskennallisen tehokkuuden kautta, muistinhallinnan näkökulmasta ja energiankulutuksen kautta. Käyttäjän näkökulmasta arvioidaan ohjelmiston käytettävyyttä seuraavin parametrein: ohjelmiston suorituskyvyllä, näytön kokoon tarkastelua ja käytettävyysanalyysillä. Ahmad ja muut (2021) tutkimuksessa kyseessä on mobiiliapplikaatio. Tästä syystä näytön kokoa tarkastellaan osana tutkimusta. Kuvassa 4 Ahmad ja muut (Ahmad et.al., 2021) havainnollistaa arviointivaiheen. (Ahmad et.al., 2021)



Kuva 5 Arviointivaiheen visualisointi (Ahmad et.al. 2021)

Prosessin toinen askel on kokonaisuuden pohtiminen prosessinäkymien kautta. Prosessinäkymät ovat abstraktio ja ilmentymän luominen (eng. ”instantiation"). Abstraktioprosessi tarjoaa tavan huomata mitä vaiheita ohjelmiston modernisointiprosessia voi olla. Tästä laaditaan lista niistä askeleista ja tehtävistä, jotka pitää ottaa, jotta modernisaatiossa päästään eteenpäin. (Ahmad et.al.,2021)

Ilmentymän luominen auttaa ymmärtämään miten ohjelmiston modernisaatio tehdään konkretian tasolla. Tällä tarkoitetaan laajemman kokonaisuuden yksittäiset tehtävät ja miten tämä tehtävä suoritetaan. Esimerkiksi prosessitoiminnan Arviointi-osuuden voi ilmentymän luomisen avulla pilkkoa pieniin osatehtäviin, joiden kautta saavutetaan Arviointi-osuuden valmistuminen. (Ahmad et.al.,2021)

Prosessin viimeinen askel on oikeiden työkalujen löytäminen modernisaatioprosessin läpivientiä varten. Ahmad ja muut (2021) ehdottaa automaation sisällyttämistä ihmisen valvonnan alla. Automaatiolla Ahmad ja muut (2021) tarkoittaa automaattista apua automaatiolta raskaissa, virheherkissä ja toistuvissa tehtävissä. Valvontatehtävät ovat puolestaan tehtäviä, jossa ihmisen valvonta on tarpeellista, eikä koneellinen automaatio voi hoitaa näitä itsenäisesti. Vaikkakin täysautomaatio on toivottua, ohjelmiston modernisaatio on älyllinen prosessi, jossa ryhmän päätöksenteko, ihmisen valvonta ja perustelut ovat tarpeen. (Ahmad et.al., 2021)

6 Keskustelu ja yhteenveto

Tämän tutkielman tavoitteena oli antaa legacy-ohjelmiston kanssa työskentelevälle työkalut legacy-ohjelmiston tunnistamiseksi ja työkaluja legacy-ohjelmiston modernisoimiseen. Koska legacy-ohjelmistot ovat laajoja, voi oikean työkalun löytäminen tuntua hankalalta. Modernisaation näkökulmasta nopea ratkaisu ei ole paras. Legacy-ohjelmiston modernisointi on työläs prosessi, mutta kun sen tekee kerralla oikein, on organisaatiolla helpompi tie tulevaisuudessa. Kirjallisuuskatsauksen avulla selvitin, mitä legacy-ohjelmista jo tiedetään. Lähteitä löytyi runsaasti, vaikkakin huomasin tiedon olevan sidottu tiettyyn kooditapaukseen, esimerkiksi tietyn ohjelmiston modernisointi. Löysin vain yhden teoksen, jossa keskusteltiin modernisaatioprosessista yleisellä tasolla. Tämä hidasti työn yleistämistä, ja taustatieto oli pirstaloitunut monen lähteen yli.

Toinen rajoittava tekijä oli aiheen rajauksen yhteydessä tehty rajaus. Päätin kirjallisuuskatsauksen alussa keskittyä kahteen osa-alueeseen: taustatietoihin Lehmanin sääntöihin nojautuen ja työkalujen esittelyyn modernisaatiota varten. Näin ollen tämän työn ulkopuolelle jäi esimerkiksi arkkitehtuuriin ja pilvipalveluihin liittyvät koko järjestelmän muutokset. Ahmad ja muut (2021) keskittyvät muita työkaluja enemmän kokonaisuuden modernisointiin, mutta tässäkään ratkaisussa ei varsinaisesti mietitä järjestelmän alustan muutoksia.

Lehmanin (1996/2005) evoluution säännöt ovat nousseet useaan otteeseen pintaan, ja useat lähteet viittaavat Lehmanin eri teoksiin, kuten Tripathyn kirja Software Evolution and Maintenance: A practitioners approach (2014) ja Bennett ja Rajlichin (2000) artikkeli Software Maintenance: a Roadmap. Lehmanin säännöt ovat tänäkin päivänä käytettäviä, ja tämän kirjallisuuskatsauksen perusteella sääntöihin nojaudutaan usein. Uusi kirjallisuus tukee Lehmanin sääntöjä, ja lisää tietoa ja käytäntöjä näiden ympärille. Pelkillä säännöillä ohjelmistoa ei kuitenkaan voi modernisoida. Sääntöjen lisäksi pitää löytää organisaatiolle sopiva menetelmä. Kuten viidennessä luvussa mainittiin, on monta eri tapaa toteuttaa modernisaatioprosessi. Avainasemassa on löytää omalle organisaatiolle sopiva menetelmä. On myös organisaation edun mukaista varmistaa kokeneen henkilöstön säilyttäminen organisaatiossa. Jos kokenut henkilöstö siirtyy uuteen työpaikkaan häivää organisaatiosta merkittävä osa tietoa ja taitoa.

Koska legacy-ohjelmistot ovat valtavan laajoja kokonaisuuksia on kaikki jo olemassa oleva tieto tarpeen. Yksi vaihtoehto voisi olla yhdistää scrum-metodi prosessikeskeisellä toteutuksella. Koska legacy-ohjelmisto on niin laaja kokonaisuus olisi suotuisaa sovittaa metodeja, jotka pilkkovat kokonaisuutta pienempiin osiin. Lisäksi olisi mielekästä pitää käärintäteknologiat mielessä, jotta varmistetaan vanhan logiikan säilyminen. On turhaa lähteä keksimään pyörää uudestaan. Prosessikeskeinen kehitys voi kuitenkin olla raskas. Pelkona on, että prosessi tuntuu liian raskaalta jo alussa eikä prosessia jakseta viedä loppuun asti.

Näiden perusteella ehdotan, että ratkaisu on kaksiosainen: kehitystiimin työskentelytapojen arviointi ja koodin muutos. Mielestäni mielekäs työskentelytapa kehitystiimille on noudattaa scrum- ja agile toimintatapoja viikoittaisessa toiminnassa. Lisäksi ehdottaisin, että organisaation johto on kuuntelemassa kehitystiimin ehdotuksia käytettävistä teknologioista, sekä aikatauluista. Varsinkin tilanteessa, jossa kokeneet kehittäjät ovat jo siirtyneet toisiin tehtäviin, tulisi modernisaatioprosessille varata runsaasti aikaa. Odotuksien hallinta ja kommunikointi sidosryhmille on tässä avainasemassa. Lisäksi kehitystiimin lisääminen jo modernisaation alkuvaiheessa prosessiin ehkäisee mahdollista muutosvastarintaa.

Koodin muunnostyöhön ehdotan työkaluksi refaktoroinnin ja käärintäteknologian yhdistelmän. Refaktoroinnin avulla voidaan tehdä muutoksia koodiin, jolla on nopeasti vaikutusta koodin luettavuuteen ja nopeuteen. Käärintäteknologian avulla voidaan säästää mahdollisimman paljon legacy-koodia, jota ei voi helposti siirtää uuteen teknologiaan.
Ehdotettu ratkaisu on useamman metodin yhdistäminen organisaatiolle sopivaksi, ja ketterän työskentelytavan sisäistämisen kehitystiimissä. Näiden työkalujen avulla päivittäinen ja viikoittainen työmäärä ei kasva liian isoksi. Kokonaisuus pysyy käsiteltävän kokoisena ja modernisaatioprosessi edistyy. Organisaation kannalta on tärkeä löytää sopiva työkalujen yhdistelmä, jonka avulla ohjelmisto saadaan modernisoitua.

Tämän työn ehdotettu ratkaisu voi olla haastava panna täytäntöön, varsinkin vanhassa organisaatiossa. Muutos on aina haastava, oli muutos mikä tahansa, ja vastarintaa voi syntyä. Muutosta ei kuitenkaan voi välttää ja kysymys kuuluu, miten muutos tehdään, ei jos muutos tehdään. Tulevaisuuden tutkimuksissa toivoisin näkeväni taustatietoa legacy-ohjelmistoista, kuten eri tunnusmerkit, sekä koostetta eri modernisaatiotavoista ja niiden positiivisia kuin negatiivisia puolia.

Lähdeluettelo

Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., Boussaidi, G. E., Hecht, G., Privat, J., & Guéhéneuc, Y.-G. (2021). A taxonomy of service identification approaches for legacy software systems modernization. *The Journal of Systems and Software 173*, 110868. https://doi.org/10.1016/j.jss.2020.110868

Ahmad, A., Alkhalil, A., Altamimi, A. B., Sultan, K., and Khan, W. (2021). Modernizing legacy software as context—sensitive and portable mobile-enabled application. *IT Professional*, *23*(1), 42-50. https://doi.org/10.1109/MITP.2020.2975997

Aljannan, A., Ismail, M., & Salah, A. (2020). Enhancing software evolution requirements engineering based on user feedback. *Computer and Information Science (Toronto), 13*(3), 16–. https://doi.org/10.5539/cis.v13n3p16

AWS (2022) *What is an IDE?* https://aws.amazon.com/what-is/ide/ (Haettu 25.11.2022)

Bennett, K., & Rajlich, V. (2000). Software maintenance and evolution: a roadmap. *International Conference on Software Engineering: Proceedings of the Conference on The Future of Software Engineering; 73–87.* https://doi.org/10.1145/336512.336534

Bosch, J., & Bosch-Sijtsema, P. M. (2011). Introducing agile customer-centered development in a legacy software product line. *Software, Practice & Experience, 41*(8), 871–882. https://doi.org/10.1002/spe.1063

Colosimo, M., Lucia, A. D., Scanniello, G., & Tortora, G. (2009). Evaluating legacy system migration technologies through empirical studies. *Information and Software Technology, 51*(2), 433–447. https://doi.org/10.1016/j.infsof.2008.05.012

IONOS Inc. (2020) *What is a wrapper in programming?* https://www.ionos.com/digitalguide/websites/web-development/what-is-a-wrapper/ (Haettu 6.11.2022)

Lehman, M.M. (2005). Laws of software evolution revisited. *Montangero, C. (eds) Software Process Technology. EWSPT 1996. Lecture Notes in Computer Science*, vol 1149. (Alkuperäinen työ julkaistu 1996) Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0017737

Mendez, M. (2016). Scientific software: legacy software maintenance. J*ournal Computer Science & Technology, 16*(2), 101–03.

Merriam-Webster -sanakirja: Legacy (2022). https://www.merriam-webster.com/dictionary/legacy. (Haettu 23.10.2022)

Merriam-Webster-sanakirja: Entropy (2022). https://www.merriam-webster.com/dictionary/entropy

MOT-sanakirja: perinnejärjestelmä (2022). (Haettu 23.10.2022)

Pohjalainen, J. (2014). Vanhan ohjelmiston nopeutus ja siirto web-arkkitehtuuriin. Diplomityö. Tampereen yliopisto. *Trepo https://urn.fi/URN:NBN:fi:tty-201406051252*

Rajlich, V. (1997) Comprehension and evolution of legacy software. *Proceedings – International Conference on Software Engineering, IEEE,* 669–670, https://doi.org/10.1109/ICSE.1997.610465.

Riederer, N. (2020). Refactoring: die Frischzellenkur für Legacy-Software. *Wirtschaftsinformatik & Management 12*, 279–281. https://doi.org/10.1365/s35764-020-00270-2

Scrum.org (2022) *What is scrum? https://www.scrum.org/resources/what-is-scrum* (Haettu 22.10.2022)

Tinetti, F., & Méndez, M. (2012). Fortran Legacy software: source code update and possible parallelisation issues. Fortran Forum, 31(1), 5–22. https://doi.org/10.1145/2179280.2179281

Tripathy, P. (2014). *Software evolution and maintenance: a practitioner's approach*, John Wiley & Sons, Incorporated.