TURUN YLIOPISTO
UNIVERSITY OF TURKU

Ohjelmistotuotannon perusteet

TURUN YLIOPISTO
UNIVERSITY OF TURKU

Ohjelmistotuotannon perusteet © 2024 by Turun yliopisto Tietotekniikan laitos is licensed under CC BY-NC-SA 4.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0/

Ohjelmistotuotannon perusteet

.
Turun yliopisto
Tietotekniikan laitos
Ohjelmistotekniikka
2024
Panu Puhtila, Hanna-Kaisa Terho, Oshani Weerakoon, Olli Heimo, Tapani Joelsson, Jari Lehto, Tuomas Mäkilä

Sisällys

1 Johdanto **1**

1.1 Ongelmanmäärittely 2

1.2 Ohjelmisto ja ohjelmistotuote 4

1.3 Ohjelmiston elinkaari 6

1.3.1 Vesiputousmalli 6

1.3.2 RADIT 7

1.3.3 Ketterät menetelmät 8

1.4 Ohjelmistoprojekti 9

1.5 Ohjelmistojen tyypit 11

1.5.1 Hyllytuote 12

1.5.2 Räätälöity ohjelmisto 12

1.6 Ohjelmistokehityksen roolit 12

1.6.1 Tilaaja/asiakas 13

1.6.2 Loppukäyttäjä 13

1.6.3 Kehitystiimi 13

1.7 Ohjelmistoprojektin työkalut 16

1.7.1 Kehitystyökalut 16

1.7.2 Versiohallinta ja CI/CD 17

1.7.3 Kommunikointityökalut 18

1.7.4 Muita työkaluja 18

1.8 Yhteenveto 20

2 Kestävä ohjelmistotuotanto **21**

2.1 Ohjelmistotuotannon etiikka 21

2.1.1 Etiikan merkitys ohjelmistotuotannossa 21

2.2 Ohjelmistokehityksen eettiset ulottuvuudet 22

2.3 Eettinen ohjelmistotuotanto 23

2.3.1 Ohjelmistotuotannon eettiset haasteet 23

2.3.2 Algoritmien aiheuttamat vinoumat 24

2.3.3 Turvallisuusheikkoudet 25

2.3.4 Prioriteettien määrittelemisen ongelmat 25

2.3.5 25

2.4 Kestävä ohjelmistokehitys 26

2.4.1 Ympäristövastuu 26

2.4.2 Vihreä koodaus 26

2.4.3 Vihreä UI/UX-suunnittelu 27

2.4.4 Virrankulutuksen mittaaminen 27

2.5 Ohjelmistotuotteiden saavutettavuus 28

3 Ohjelmistoprojektin aloitus **31**

3.1 Asiakkaan taustojen selvittäminen 31

3.2 Myy osaaminen / idea 32

3.3 Tunnista sidosryhmät 33

3.4 Toteutusteknologian kartoitus 35

3.5 Ohjelmiston elinkaaren suunnittelu 37

3.6 Aikataulutus, resurssointi ja budjetointi 39

3.7 Valitse toteutustiimin jäsenet 41

3.8 Rakenna kehitysympäristö 44

3.8.1 Versionhallinta 44

3.8.2 CI/CD -putket 45

3.8.3 Testiympäristö 46

3.9 Tiimityön pohjustus 46

3.10 Korkean tason komponentit ja rajapinnat 47
3.11 Olemassa olevan koodin hyödyntäminen 48

4 Ohjelmistojärjestelmän kehitys **50**
4.1 Toiminnalliset ja ei-toiminnalliset vaatimukset 50
4.2 Ymmärrä asiakkaan tarve yleisellä tasolla 51
4.3 Järjestelmän yleisrakenteen ymmärtäminen 53
4.4 Järjestelmän mallintaminen (UML) 54
4.5 Suunnittele järjestelmän tietoturva, tietosuoja ja käyttöturvallisuus sekä eettiset aspektit 57
4.6 Suunnittele järjestelmän käytettävyys ja käyttötilanteet 59
4.7 Kehitystyön koordinointi ja johtaminen 62
4.7.1 Scrum 62
4.7.2 Lean 66
4.7.3 Kanban 67
4.8 Suunnitteluvirheet ja kuviot 68
4.9 Tee yksikkö- ja muut automaattiset testit 69
4.10 Implementoi 71
4.11 Hanki valmiit komponentit ja konfiguroi 72
4.12 Versionhallinta 73
4.13 Integroi ja asenna (CI/CD) 75
4.14 Kommunikointi ja koordinointi 76
4.14.1 Kokouskäytännöt 76
4.14.2 Viestintä- ja tiketointiohjelmistot 77
4.15 Asiakkaan tarpeiden oppiminen iteratiivisesti 78
4.16 Ohjelmistokehityksen mittaaminen 78
4.17 Retrospektiivit eli työskentelytavan mukauttaminen 79
4.18 Riskien seuranta ja reagointi 80
4.19 Osajärjestelmien integrointi 81
4.20 Testaus, skaalautuvuus ja kuormitus 82

4.21 Käytettävyystestaus . . . 83
4.22 Teknisen velan hallinta . . . 85
4.23 Pidä huolta dokumentaatiosta ja käyttöohjeista . . . 86
4.24 Tukijärjestelmät ja -toimet . . . 87
4.25 Käyttäjien koulutus . . . 88
4.26 Julkaisu . . . 88

5 Ohjelmistotuotteen ja -palvelun ylläpito 90
5.1 Järjestelmän julkaisu . . . 90
5.2 Datan keruu . . . 91
5.3 Monitorointi . . . 92
5.4 Ekosysteemin seuranta . . . 93
5.5 Ylläpidä tietoturvaa . . . 93
5.6 Korjaa ohjelmointivirheet hallitusti . . . 94
5.7 Elinkaaren aikainen päivittäminen . . . 94
5.8 Jatkuva testaaminen . . . 95
5.9 Päivitä käyttäjää häiritsemättä . . . 95
5.10 Ylläpito . . . 96
5.11 Elinkaaren päätös . . . 96

6 M.STUP -menetelmäopas 98
6.1 M.STUP -menetelmäopas . . . 98

Lähdeluettelo 99

Kuvat

1.1 Ohjelmistotuotantoprosessi projektina [4] 3
1.2 Vesiputousmalli 6
1.3 Testauksen V-malli 9
1.4 Asiakas- ja ohjelmistovaatimukset [4] 10
1.5 Projektikolmio 11

3.1 Esimerkki Gantt-kaaviosta (Lähde: Wikipedia) 41

4.1 UML-kaavioiden eri tyypit. 55
4.2 Scrum framework (Lähde: scrum.org) 62

5.1 Tuotteenhallinnan osa-alueet [4] 95

Taulukot

1 Johdanto

Kansainvälisen tekniikan alan kattojärjestön IEEE:n määritelmän mukaan ohjelmistotuotanto (engl. software engineering) tarkoittaa:

> *Systemaattista, kurinalaista ja mitattavissa olevaa lähestymistapaa ohjelmistojen kehittämiseen, käyttöön ja ylläpitoon eli insinöörimenetelmien soveltamista ohjelmistoihin.*[1]

On lukuisia syitä kirjoittaa tietokoneohjelmia. Toiset tekevät niitä harrastusprojekteinaan tai parantamaan elämänlaatuaan, jotkut ratkaistaakseen ongelmia esimerkiksi matematiikassa tai tähtitieteessä. Ohjelmointiprojektien tekeminen on ollut klassisesti hyvä tapa oppia ja opiskella tietokoneiden toimintaa.

Suurin osa suuremmista ohjelmistoprojekteista tehdään liiketaloudellisista syistä ja siksi ohjelmistotuotannon periaatteet noudattavatkin liiketaloudellisia lainalaisuuksia. Tästä seuraa se, että ammattimaisen ohjelmistokehittäjän on osattava huomioida laajasti erilaisia työnsä osa-alueita projektihallinnasta budjetointiin sekä sidosryhmien ymmärtämisestä elinkaariajatteluun. Monia ohjelmistoja sitovat myös lait ja asetukset, joiden perusperiaatteet tai ainakin olemassaolo olisi ohjelmistokehittäjän hyvä tiedostaa.

Ohjelmistot ovat nykyään pervasiivisia – ne ovat kaikkialla – ja digitalisaatio onkin edennyt analogisten järjestelmien digitoinnista ja automatisoinnista digitaalijärjestelmien päivittämiseksi, parantamiseksi ja automatisoinniksi. Automaation lisäksi ohjelmistoja käytetään laajasti muun muassa prosessien tehostamiseksi, uusien palveluiden ja toimintatapojen luomiseksi, asiakastarpeiden täyttämiseksi, kilpailukyvyn parantamiseksi sekä tietenkin säästöjen saamiseksi. Näille kaikille yhteistä on ohjelmistokehityksen valtava voima muuttaa ympäröivää maailmaa.

[1] IEEE 610.12-1990

Tämä oppikirja tarjoaa yhden läpileikkauksen ohjelmistotuotantoon. Tutustumme ohjelmistotuotantoon ketterien menetelmien kautta tarkastellen sen eri vaiheita ja menetelmiä siten kuin yksittäinen ohjelmistokehittäjä ne näkee ohjelmistoprojektin kuluessa. Ohjelmistotuotannon iteratiivisesta luonteesta johtuen monet osat materiaalista menevät keskenään päällekkäin, sillä samoja tekniikoita hyödynnetään ohjelmistokehityksen eri vaiheissa. Näin ollen kirjan eri luvuissa käsitellään samoja tekniikoita eri näkökulmista laajemman näkemyksen saavuttamiseksi. Materiaali on lähteytetty monipuolisesti muun muassa tieteellisillä artikkeleilla, aihepiirin videoilla, ja ohjelmistoyritysten sekä yksittäisten ohjelmistokehittäjien blogeilla, jotta lukija voi tutustua tarkemmin häntä kiinnostaviin ohjelmistotuotannon osa-alueisiin.

1.1 Ongelmanmäärittely

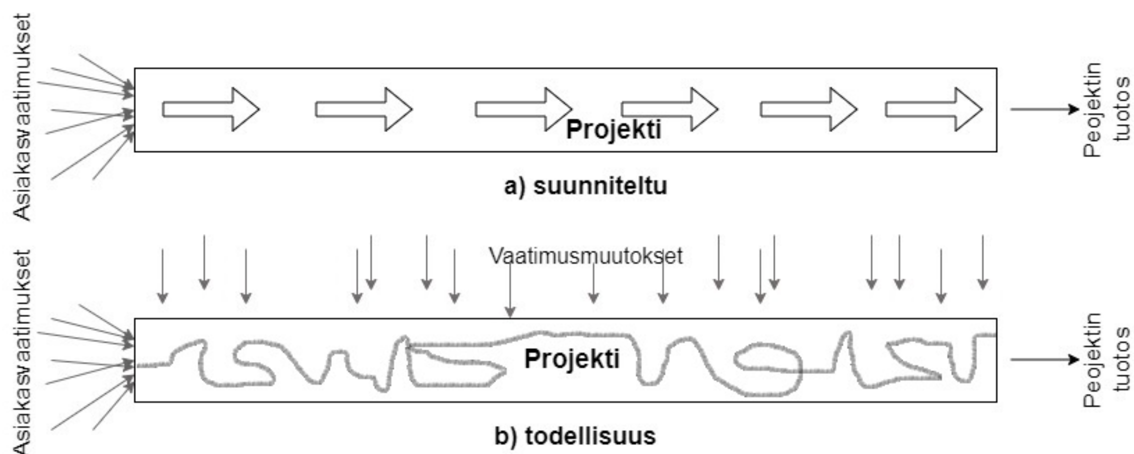
Ongelmanmäärittelyn näkökulmasta ohjelmistotuotanto on prosessi, joka sisältää useita toisiinsa kytkeytyviä, vaikeasti hallittavia ja ratkaistavia tekijöitä. Ohjelmistotuotantoa voidaankin kuvailla käsitteellä “viheliäinen ongelma eli wicked problem”. Ongelma on tällainen, kun sen ratkaisun vaatimukset tiedetään vasta sitten kun työ on saatu valmiiksi tai vaatimukset elävät työn aikana. [1] Vastaavasti “kesyt ongelmat eli tamed problems” ovat ongelmia, joille löytyy selkeä määritelmä. Tämä ei kuitenkaan tarkoita, että ongelma olisi pieni tai helppo ratkaista, se on vain hyvin määritelty [2]. Ongelmanmäärittelystä kiinnostuneiden kannattaa tutustua John Dooleyn blogikirjoitukseen, jossa on tiivis koonti aiheeseen [3].

Yksittäisen ohjelmiston kehittäminen on projekti, johon on olemassa rajaton määrä erilaisia lähestymistapoja. Erityyppiset ohjelmistot vaativat erilaisia toimintamalleja niin menetelmien, tekniikoiden kuin työtapojenkin suhteen: Globaalin tietojärjestelmän kehittäminen poikkeaa suuresti esimerkiksi lääkinnällisen laitteen kehittämisestä. Yksinkertaisin ohjelmistokehityksen menetelmä on ns. Koodaa ja korjaa[2] (engl. code-and-hack, code-and-fix), jossa ohjelmistoa muokataan niin kauan, kunnes asiakas (yleensä ohjelmoija itse) on siihen tyytyväinen. Ohjelmistotuotteen kehittäminen vaatii kuitenkin tyypillisesti laajan ja monimuotoisen ryhmän osallistujia. Tämä joukko voi koostua yksittäisestä freelancerista, pienestä startup-tiimistä tai suuresta monikansallisesta organisaatiosta, jonka palveluksessa voi olla tuhansia kehittäjiä. Hajau-

[2] The Code and Fix Model

tettu ympäristö tuo omat haasteensa kommunikaatioon, koordinointiin ja projektinhallintaan, toisaalta teknologian kehitys, kuten pilvipalvelut, yhteistyötyökalut ja hajautetun versionhallinnan työkalut, ovat helpottaneet näiden haasteiden voittamista. Hajautettu työskentely voi myös avata uusia mahdollisuuksia, kuten osaajien rekrytoinnin maarajojen yli, työnteon joustavuuden ja mahdollisuuden hyödyntää eri aikavyöhykkeiden tuomaa etua työn jatkuvuuden näkökulmasta.

Ohjelmistotuotanto on luovaa työtä (vrt. Knuthin *The Art of Computer Programming* -kirjasarjan[3] nimi, art, ei science). Useimpiin ohjelmistotuotannon ongelmiin ei ole olemassa yhtä oikeaa ratkaisua. Lineaariseen, ennustettavaan etenemiseen perustuva projektinhallinta soveltuu huonosti tähän ongelmakenttään. Nykyaikaisissa ohjelmistoprojekteissa sovelletaankin mittavissa määrin joustavia ja iteratiivisia eli toistoon perustuvia toimintatapoja, jotka mahdollistavat mukautumisen ja oppimisen ongelman tarkentumisen myötä. Seuraavan sivun kuva 1.1 ohjelmistotuotannon prosessista havainnollistaa hyvin suunnitellun ja todellisen toteutuksen eroa. Käytännön ohjelmistotuotanto on siis enneminkin yksityisjetti, jota aktiivisesti ohjataan kohti asiakkaalle parhaiten sopivaa kohdettaan, kuin juna, joka etenee raiteillaan pisteestä A pisteeseen B.



Kuva 1.1: Ohjelmistotuotantoprosessi projektina [4]

[3]The Art of Computer Programming: `https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming`

1.2 Ohjelmisto ja ohjelmistotuote

Ohjelmistotuote voi tarkoittaa monenlaista ohjelmistoa, aina yksinkertaisista apuskripteistä monimutkaisiin useista ohjelmistokomponenteista, laitteista ja palveluista muodostuviin järjestelmiin. Puhtaista tietokoneohjelmista sen erottaa liiketaloudelliset ulottuvuudet. Ohjelmistotuote on **tuote**, jonka tavoitteena kehittäjän näkökulmasta on tuottaa liikevaihtoa, jolla katetaan vähintäänkin ohjelmistotuotteen välittömät kehityskulut, ja yritysasiakkaan näkökulmasta on tehostaa jotakin liiketoiminnallista osa-aluetta tai prosessia. Kuluttaja-asiakkaille suunnatut ohjelmistotuotteet tuovat lisäarvoa arkeen tai harrastuksiin, mutta voivat olla myös puhtaasti viihteellisiä.

Ohjelmistotuotteet vaihtelevat kompleksisuudeltaan ja laajuudeltaan. Koodin määrä ohjelmistotuotteessa voi vaihdella yksittäisestä rivistä aina kymmeniin miljooniin riveihin, ja koodi voi olla jakautunut kymmeniin tuhansiin erillisiin tiedostoihin [5]. Tekijöitä voi olla yhdestä tuhansiin, hajautuneina lukuisiin tiimeihin, jotka voivat työskennellä eri toimistoissa, eri maissa, eri mantereilla ja eri aikavyöhykkeillä. Esimerkkinä tällaisesta hajautetusta monivuotisesta ja monipuolisesta projektista voidaan pitää GTA V -peliprojektia jota toteuttamassa oli noin tuhat kehittäjää usean vuoden ajan[4]. Tämä heijastaa ohjelmistotuotteiden kehityksen monimutkaisuutta ja laajuutta sekä sitä, miten paljon työtä kehittämiseen ja ylläpitoon tarvitaan.

Ohjelmistotuote ei koostu pelkästään koodista. Sen ytimessä on koodi, mutta se ei ole ainoa osa, joka määrittää tuotteen laadun ja toimivuuden. Koodin lisäksi ohjelmistotuotteeseen liittyy usein laaja kirjo muita materiaaleja ja komponentteja, jotka täydentävät tuotteen toimivuutta ja käytettävyyttä. Esimerkiksi käyttöohjeet ja koulutusmateriaalit ovat keskeisiä osa käyttäjän tuen kannalta ja mahdollistavat sen, että ohjelmiston käyttö on selkeää.

Monissa ohjelmistotuotteissa hyödynnetään myös grafiikkaa, ääniä ja videoita, jotka laadukkaasti toteutettuna voivat parantaa ohjelmiston käyttökokemusta ja tehdä ohjelmistosta houkuttelevamman. Audiovisuaalisen materiaalin avulla voidaan havainnollistaa monimutkaisiakin konsepteja, joka on tärkeää erityisesti järjestelmissä, joissa käyttäjän on vuorovaikutettava monien eri elementtien kanssa. Joissakin tapauksissa ohjelmistotuote voi sisältää myös lait-

[4]Wikipedia-artikkeli GTA Vn kehittämisestä: `https://en.wikipedia.org/wiki/Development_of_Grand_Theft_Auto_V`

teistokomponentteja. Esimerkiksi sulautetuissa järjestelmissä ja IoT-sovelluksissa ohjelmiston ja laitteiston yhdistelmä muodostaa tuotekokonaisuuden.

On kuitenkin hyvä muistaa, että ohjelmistotuote on tuote, eli hyödyke tai palvelu, jonka tavoitteena on luoda taloudellista arvoa sekä asiakkaalle että tuotteen kehittäjälle. Vaikka on selvää, että lähtökohtana ohjelmistoprojektissa ovat lisäarvon tuottaminen asiakkaalle, ongelmia tulee vastaan jo projektin alussa, jos asiakas ei pysty kuvaamaan tarpeitaan selkeästi eikä ohjelmiston kehittäjä onnistu hahmottamaan sitä, mihin projektilla pyritään. Ohjelmiston kehittäjän tulee myös ottaa huomioon oma liiketoimintansa, eli se, että projekti on myös tekijälle kannattava.

Vaikka ohjelmistotuotantoon on olemassa järjestelmällisiä ja tehokkaita lähestymistapoja, Tivin artikkelin[5] mukaan vain kolmasosan IT-projekteista voidaan katsoa onnistuneen, kun onnistumisen kriteereinä on aikataulussa pysyminen, budjetin pitävyys ja tavoitteen mukainen lopputulos. Lisäksi arvion mukaan jopa 10–15 prosenttia IT-hankkeista epäonnistuu täydellisesti. Tivian raportti [6] tietojärjestelmähankinnoista esittää, että tyypillisimmät syyt tietojärjestelmäprojektien epäonnistumiseen ovat kommunikoinnin ja viestinnän puute, resurssien riittämättömyys sekä tehoton projektinhallinta. Raportissa todetaan myös, että asiakaspuolella saattaa olla puutteellinen käsitys siitä, millaisia lainalaisuuksia ja toimintatapoja ohjelmistoalaan liittyy, joka voi johtaa kommunikaatio-ongelmiin ja väärinkäsityksiin. Ohjelmistotuotannon osaamisella ja erityisesti kommunikaatioon panostamalla voidaan välttää näitä sudenkuoppia.

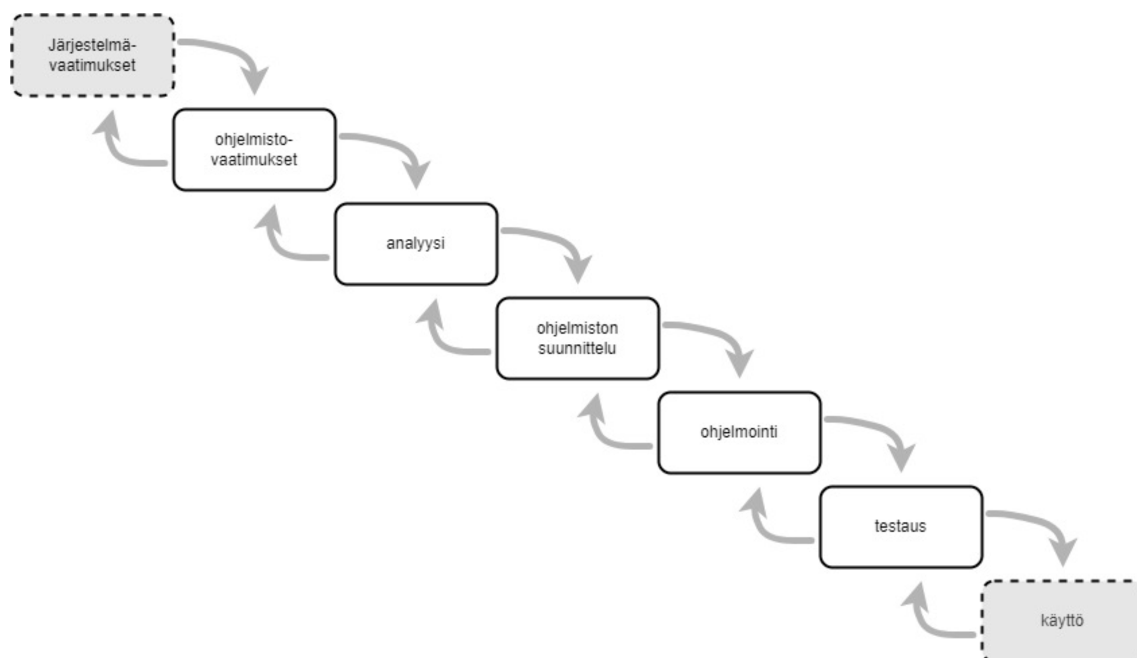
Kuten todettua, ohjelmistotuote on enemmän kuin pelkkä ohjelmakoodi – se on kokonaisvaltainen ratkaisu, joka vastaa asiakkaan tarpeita ja odotuksia sekä tarjoaa taloudellista arvoa sekä tuottajalleen että asiakkailleen. Ohjelmistotuotteen kehittäminen on ennen kaikkea yhteistyötä ja aktiivista molemminpuolista viestintää. Tilaaja–toimittaja-vastakkainasettelun sijaan ohjelmiston kehittäjä ja asiakas tulisi nähdä ohjelmistoprojektin yhteistyökumppaneina työskentelemässä yhteisen onnistumisen eteen.

[5] Miksi suurin osa IT-hankkeista epäonnistuu?

1.3 Ohjelmiston elinkaari

Ohjelmiston kehitystä voidaan tarkastella elinkaariajattelun kautta. Elinkaarimallin tarkoitus on kuvata kehitysvaiheet, joita noudatetaan ohjelmistokehitysprojektissa. Oleellista elinkaarimallissa on toistettavuus; se tarjoaa määritellyn kehyksen, joka soveltuu monenlaisten ohjelmistoprojektien pohjaksi. Elinkaarimalleja on useita, joista eräs klassinen esimerkki on seuraavassa alaluvussa kuvattu vesiputousmalli. Elinkaaren vaiheista toteutus- eli ohjelmointivaihe on yleensä tutuin ja konkreettisin harrastelijoille sekä uransa alkuvaiheessa oleville kehittäjille. Varsinainen ohjelmistoprojekti sisältää kuitenkin useita eri vaiheita, joista kaikissa ei välttämättä kirjoiteta riviäkään ohjelmistokoodia.

1.3.1 Vesiputousmalli



Kuva 1.2: Vesiputousmalli

Vesiputousmalli[6] on tunnettu käsitteenä ainakin 1950-luvulta saakka. Klassinen artikkeli aiheesta on W. Roycen *Managing the Development of Large Software Systems* [7] vuodelta 1970. Malli (kuva 1.2) esittää ohjelmiston suunnittelun vaiheistettuna prosessina, joka etenee järjestelmän ja ohjelmiston vaatimuksien määrittelystä analyysiin ja ohjelmiston suunnitteluun. Näiden jälkeen suoritetaan ohjelmointi ja testaus, sekä viimeisenä vaiheena käyttöönotto. Roycen

[6] Waterfall model

mallia on tulkittu niin, että vaiheet tapahtuvat peräkkäin järjestyksessä ilman että edeltävään vaiheeseen enää palataan. Tämä ajattelutapa on virheellinen, sillä Roycen näkemyksen mukaan samaa sykliä voidaan suorittaa useamman kerran, vaiheiden välisiä iteraatioita on mahdollista hyödyntää ja asiakasta voidaan myös osallistaa prosessin eri vaiheissa. [4] Malli ei ota kantaa ohjelmiston käyttöönoton jälkeisiin vaiheisiin eikä vaatimusmäärittelyä edeltäviin vaiheisiin. Niiden osalta täydentäen vaiheet näyttävät seuraavalta:

- *Esimäärittely*
- *Hankinta*
- Vaatimusmäärittely
- Analyysi
- Suunnittelu
- Kehitys
- Validointi
- Käyttöönotto
- *Ylläpito*
- *Käytöstä poistuminen/Korvaaminen uudella*

Ohjelmistotuotannon historian keskeiset kehityskulut lyhyesti: History of software engineering

1.3.2 RADIT

RADIT on ohjelmistokehityksen prosessimalli, jonka vaiheet ovat vaatimukset (*Requirements*), analyysi (*Analysis*), suunnittelu (*Design*), toteutus (*Ímplementation*) ja testaus (*Testing*). [8] RADIT-aktiviteetit löytyvät Roycen vesiputousmallista ja on korostettu vesiputousmallia kuvaavassa kuviossa 1.2 valkoisella. Kuviossa ohjelmistovaatimukset vastaa RADIT-mallin *Requirements*-vaihetta ja ohjelmointi *Implementation*-vaihetta. RADIT-mallin toiminnot löytyvät lähes jokaisesta ohjelmistokehityksen projektimallista jossain muodossa.

Vaatimusmäärittelyvaiheen tavoitteena on ymmärtää asiakkaan tarpeet ja odotukset ohjelmistolle. Tähän kuuluvat sekä toiminnalliset vaatimukset eli mitä ohjelmisto tekee (esim. käyttäjän tulee voida lähettää järjestelmän kautta viestejä muille käyttäjille) että ei-toiminnalliset vaatimukset eli miten ohjelmiston tulisi toimia (esim. järjestelmän tulee mahdollistaa 10000 käyttäjän yhtäaikainen toiminta) . Analyysivaihe ei keskity ainoastaan vaatimusten analysointiin. Analyysivaiheessa tehdään korkean tason teknistä analyysiä järjestelmän arkkitehtuurin muodostamiseksi. Alemman tason ohjelmistosuunnittelu tapahtuu suunnitteluvaiheessa. Tässä vaiheessa määritellään yleensä ohjelmiston komponentit, komponenttien tietomalli, komponenttien metodirajapinnat ja komponenttien väliset suhteet ylemmän tason arkkitehtuuria seuraten. Suunnitteluvaiheen tuloksena on ohjelmistosuunnitelma, joka ohjaa toteutusvaihetta. Toteutusvaiheessa ohjelmiston koodi kirjoitetaan ja toteutetaan käyttäen valittuja ohjelmointikieliä ja työkaluja. Vaihe sisältää varsinaisen ohjelmoinnin sekä monesti myös metoditason toiminnallisuuksien testauksen eli yksikkötestauksen. Vaihetta kutsutaan usein myös integraatiovaiheeksi, koska sen merkittävin haaste on erillisten kehittäjien tuottaman ohjelmakoodin ja ohjelmistomoduulien yhteentoimivuus. RADIT-syklin viimeinen vaihe on testaus. Testausvaiheessa varmistetaan, että ohjelmisto toimii kokonaisuutena suunnitellusti ja täyttää asiakkaan vaatimukset. [8]

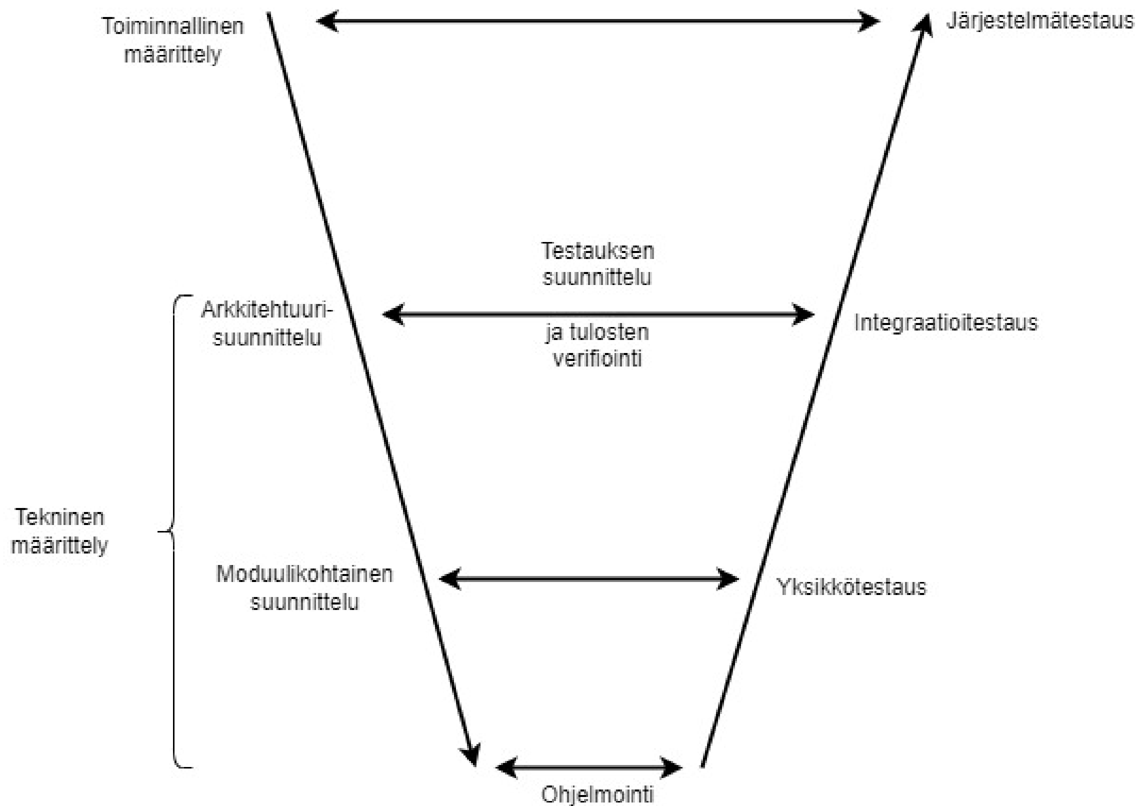
Testausspesifeistä prosessimalleista on syytä mainita V-malli (kuva 1.3), joka kuvaa suhdetta ohjelmiston elinkaaren suunnittelun ja testauksen välillä. V-mallia tarkastellaan yksityiskohtaisemmin testausta käsittelevän alaluvun 4.9 infolaatikossa.

1.3.3 Ketterät menetelmät

Ketterän ohjelmistokehityksen juuret ulottuvat 1990-luvulle, jolloin ohjelmistoteollisuudessa alettiin etsiä vaihtoehtoja perinteisille projektimalleille. Vuonna 2001 ketteryyden ajatus konkretisoitui, kun 17 ohjelmistokehityksen asiantuntijaa kokoontui Utahissa ja laati ketterän ohjelmistokehityksen manifestin[7].

Julistuksessa määriteltiin neljä keskeistä arvoa ja kaksitoista periaatetta, jotka ohjaavat ketterää ohjelmistokehitystä. Ne korostavat muun muassa yksilöiden ja vuorovaikutuksen merkitystä prosesseja ja työkaluja enemmän, toimivan ohjelmiston etusijaa kattavaan dokumentaatioon

[7] Manifesto for Agile Software Development



Kuva 1.3: Testauksen V-malli

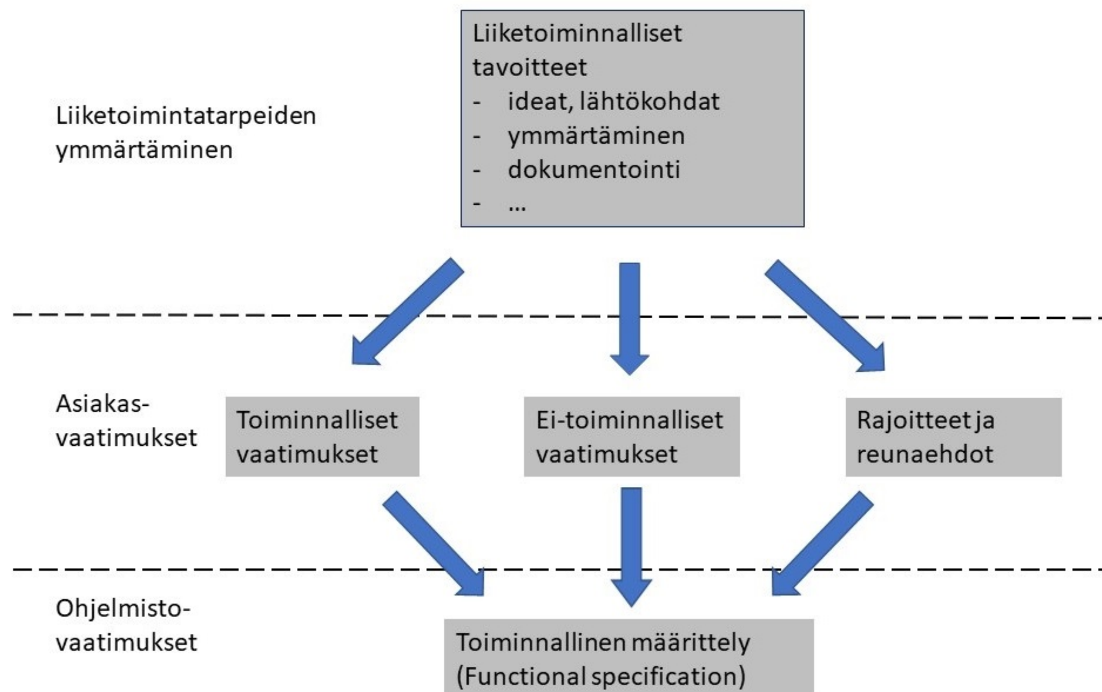
nähden, asiakasyhteistyötä sopimusneuvotteluiden sijaan, sekä valmiutta vastata muutoksiin suunnitelmien seuraamisen sijaan. Ketterä manifesti on toiminut pohjana monille nykyaikaisille agile-menetelmille kuten Scrum, Kanban ja Extreme Programming (XP).

Ketterillä käytänteillä tarkoitetaan työskentelymenetelmiä, jotka tukevat ja mahdollistavat kehittäjien itsenäistä vastuuta omasta työstään ja työn suunnittelusta. Ketterän toiminnan periaatteiden mukaan tiimit toimivat itseohjautuvasti sopien työtehtävistä tiimin kesken. Samalla kehitetään tiimin yhteisiä toimintatapoja ja käytänteitä. Teknisen tekemisen lisäksi tiimi esittää, millä aikataululla tehtäviä on mahdollista tehdä. Käytännön työ pohjautuu aktiiviseen kommunikaatioon yksilöiden ja ryhmien välillä, jota käytössä olevat työkalut ja menetelmät tukevat.

1.4 Ohjelmistoprojekti

Ohjelmistoprojekti on prosessi, jossa uutta ohjelmistoa tuotetaan tai olemassa olevaa ohjelmistoa päivitetään. Kuten aiemmissa luvuissa on esitetty, ohjelmistoprojektin perimmäisenä

tavoitteena on asiakkaan liiketoimintatarpeiden ymmärtäminen ja niiden muuttaminen toiminnallisiksi (engl. Functional requirements, FR) ja ei-toiminnallisiksi vaatimuksiksi (engl. Non-functional requirements, NFR), huomioiden projektin rajoitteet ja reunaehdot (kuvio 1.4). Vasta tämän jälkeen voidaan siirtyä toiminnalliseen määrittelyyn. Nykymuotoiset ohjelmistoprojektit harvoin lähtevät liikeelle täysin tyhjältä pöydältä, sillä usein uudet ohjelmistot rakennetaan olemassa olevien alustojen, sovelluskehysten tai jo olemassa olevien ohjelmaversioiden pohjalle. Ne voivat myös perustua samankaltaisiin järjestelmiin, jotka on suunniteltu eri ympäristöihin tai käyttötarkoituksiin. [4, s. 26] Toisinaan vanha koodipohja voi määrätä työkalut, jotka projektissa ovat käytettävissä, ja myös asiakkaalla voi olla asian suhteen omia linjauksia.

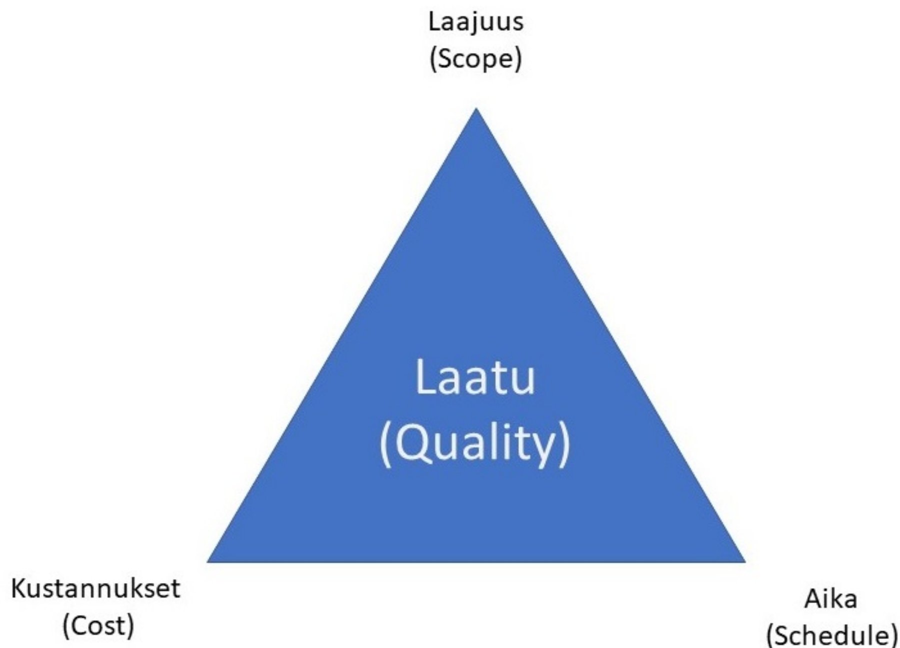


Kuva 1.4: Asiakas- ja ohjelmistovaatimukset [4]

Projektinhallinnan rautakolmio[8] (engl. *Iron Triangle, Triple Constraint*) on eräs tapa kuvata ohjelmistoprojektiin (tai mihin tahansa projektiin) vaikuttavia rajoitteita, eli aikaa, kustannuksia ja laajuutta (kuva 1.5 sivulla 11). Laatu on kuvattu kolmion keskelle ja se on projektikolmion ainoa muuttumaton tekijä, laadun tulee siis säilyä, vaikka projektin muissa osa-alueissa tapahtuu muutoksia. Projektikolmion perusidea on, että kolmion kärjet ovat riippuvaisia toisissaan tapahtuvista muutoksista. Jos esimerkiksi laajuutta halutaan muuttaa ohjelmiston toi-

[8] Iron Triangle, Triple Constraint

minnallisuutta lisäämällä, näkyy tämä vaikutus budjetin kasvattamisena ja/tai aikataulun venymisenä. Samalla logiikalla projektin nopeuttaminen vaatii joko budjetin kasvattamista tai projektin laajuudesta tinkimistä. Jos budjetissa ja aikataulussa ei ole joustovaraa, täytyy mahdollisesti tinkiä ohjelmiston toiminnallisuuksista.



Kuva 1.5: Projektikolmio

1.5 Ohjelmistojen tyypit

Ohjelmistosta ja tarpeista riippuen ohjelmistotuotteen asiakaskunta voi olla hyvinkin erilainen. Ohjelmistotyypit voidaan karkeasti jakaa hyllytuotteisiin (engl. Off the shelf software, commercial off the shelf software, COTS software) ja räätälöityihin ohjelmistoihin (engl. custom software, tailored software). Ohjelmistotyön haaste on tasapainoilu tuotekeskeisyyden ja asiakaskohtaisuuden välillä [4, s. 23]. Mitä tarkemmin asiakkaan vaatimukset voidaan ottaa huomioon, sitä tyytyväisempi asiakas tyypillisesti on lopputulokseen. Toisaalta mitä useamman asiakkaan tarpeet voidaan tyydyttää valmisohjelmistolla, sitä vähemmän resursseja tuottajalta kuluu monien erilaisten ohjelmistojen ylläpitoon, mikä monesti lisää kehittäjän liikevaihtoa mutta myös pienentää merkittävästi asiakkaalle koituvia kustannuksia.

1.5.1 Hyllytuote

Hyllytuotteella tarkoitetaan ohjelmistotuotetta, jolla ei ole ainoastaan yhtä kohdeasiakasta, vaan laajempi asiakasryhmä. Näillä asiakkailla voi olla osittain yhteneväiset ja osittain erilaiset tarpeet ohjelmiston suhteen, mutta asiakasvaatimukset ovat usein kuitenkin riittävän samanlaiset, että samaa tuotetta voidaan tarjota usealle asiakkaalle sopivasti konfiguroimalla. Hyllytuotteen keskeisin etu on sen vähäisempi muokattavuus, joka tarkoittaa toimittajan tapauksessa vähemmän määriteltävä, kehittämistä ja testausta. Ohjelmistohankintanäkökulmasta hyllytuote eroaa räätälödystä tuotteesta siinä, että asiakas lähestyy ohjelmistohankintaa tuotekeskeisesti [4]. Tällöin ohjelmistoprojektin painopisteenä on olemassa olevan tuotetarjonnan kartoitus, eikä niinkään sopivan toimittajan valinta. Hyllytuotteen kehittämisessä on keskeistä erinomainen markkinoiden ja asiakastarpeiden tuntemus.

1.5.2 Räätälöity ohjelmisto

Räätälöidyt ohjelmistot suunnitellaan ja kehitetään tietyn organisaation tai käyttäjäryhmän yksilöllisiin tarpeisiin. Niitä kehitetään usein alusta alkaen asiakkaan vaatimusten mukaisesti, ja niissä voidaan ottaa huomioon asiakkaan erityistarpeet ja -toiveet. Räätälöityä ohjelmistoa hankkivan asiakkaan kohdalla painottuu sopivimman ohjelmistotoimittajan valinta [4]. Toimittajan näkökulmasta on huomioitava, että mitä useammalle asiakkaalle tehdään räätälöityjä ohjelmistoja, sitä enemmän toimittaja joutuu huomioimaan ohjelmistojen ylläpitämiseen liittyvät erot ja yksityiskohtaisuudet. On kuitenkin huomattavaa, että räätälöityjä ohjelmistoja rakennetaan nykyään sekä avoimesti saatavien alustojen että toimittajan omien sisäisten alustojen päälle. Tällöin ero hyllytuotteen ja räätälöidyn ohjelmiston välillä hämärtyy.

1.6 Ohjelmistokehityksen roolit

Ohjelmistoprojektissa tarvitaan monipuolista osaamista sekä kehitystiimin sisällä että kehitystyöhön osallistuvissa sidosryhmissä. Nykyaikaisessa ketterässä ohjelmistokehityksessä erityisesti pienempien kehitystiimien kohdalla korostetaan jokaisen tiiminjäsenen osallistumista kaikkiin kehitystyön vaiheisiin. Tästä huolimatta projekteissa tarvitaan myös erikoistunutta osaamista sekä projektin sujuuvudesta vastaavia rooleja.

1.6.1 Tilaaja/asiakas

Asiakas eli tilaaja on henkilö tai organisaatio, joka rahoittaa ohjelmistoprojektin ja on yleensä vastuussa projektin hallinnasta ja sen tavoitteiden määrittelystä. Tilaaja tekee sopimuksen toimittajan kanssa, ja määrittelee projektin budjetin, aikataulun ja laajuuden. Tilaajan näkökulmasta ohjelmistoprojekti liittyy yleensä konkreettisiin liiketoimintapäämääriin; ohjelmistoprojektin ensisijainen tavoite on varmistaa, että investointi tuottaa halutun liiketoimintahyödyn tai ratkaisee tietyn ongelman. Tämän vuoksi asiakas kokee projektin usein laajempana kokonaisuutena kuin pelkkänä ohjelmistotoimituksena. Asiakas voi olla osa samaa organisaatiota kuin loppukäyttäjät, mutta ei välttämättä suoraan käytä lopputuotetta. Asiakkaan näkökulmasta projekti on onnistunut, kun se valmistuu aikataulunsa ja budjettinsa pitäen sisältäen ne ominaisuudet ja toiminnot, jotka vaaditaan asetetun liiketoimintatavoitteen saavuttamiseksi.

1.6.2 Loppukäyttäjä

Loppukäyttäjä on henkilö tai ryhmä, joka lopulta käyttää ohjelmistotuotetta. Loppukäyttäjän tyytyväisyys on eräs tärkeimmistä kriteereistä mitata ohjelmistoprojektin onnistumista. Toisinaan voi käydä niinkin, että taloudellisesti ja hallinnollisesti projekti voi olla hyvin onnistunut, mutta loppukäyttäjä ei ole tuotteeseen tyytyväinen. Tämän takia loppukäyttäjien tarpeita, toiveita ja kokemuksia on hyvä ottaa huomiooon jo kehitysprojektin alkuvaiheessa. On kuitenkin myös hyvä huomioida, ettei loppukäyttäjä tiedä kaikkea eikä aina ole oikeassa.

1.6.3 Kehitystiimi

Ohjelmistotiimi koostuu erilaisista osaajaryhmistä. Suurten ohjelmistoprojektien läpivienti vaatii paljon työvoimaa ja erilaisia taitoja. Erityisesti isommissa ohjelmistotaloissa tiimit toimivat maantieteellisesti hajautetusti. Tiimien koostumus riippuu projektista ja projektin koosta. Esimerkiksi sulautettua järjestelmää kehittävässä projektissa tarvitaan ohjelmistokehittäjiä, elektroniikkasuunnittelijoita sekä mekaniikkasuunnittelijoita. Tyypillisesti ohjelmistotiimi vastaa itse työnsä organisoinnista ja teknisten ratkaisujen määrittelystä asiakkaan vaatimusten täyttämiseksi, huomioiden projektin rajoitteet ja reunaehdot. Nykyään ohjelmistokehitystiimeissä noudatetaan tyypillisesti ketteryyden arvoihin ja periaatteisiin nojaavia toimintatapoja ja työskentelymalleja kuten Scrumia.

Tuoteomistaja

Scrum-tiimissä tuoteomistajan (engl. Product Owner) [4, s. 48] tehtävänä on vastata projektin taloudellisesta tuloksesta ja tuotteen vision pysymisestä selkeänä. Hänen tehtävänään on varmistaa, että tiimi ymmärtää sekä asiakkaan että loppukäyttäjien tarpeet. Tuoteomistajan tehtävänä on toimia rajapintana ohjelmistoprojektin moniin sidosryhmiin. Tuoteomistaja voi olla joko asiakkaan tai toimittajan palveluksessa.

Tuoteomistaja ylläpitää ja priorisoi järjestelmän vaatimusten mukaista työlistaa (engl. product backlog). Työlista koostuu alkioista (engl. item), joista jokaisella on alustava aika-arvio sekä arvio liiketoiminta-arvosta. Työlistan alkiot voivat olla esimerkiksi:

- tuotteen ominaisuuksia
- käyttötapauksia
- käyttäjätarinoita
- vaatimuksia
- virheraportteja
- dokumentaation kehittämistä
- arkkitehtuurin parantamista

Scrum Master

Scrum Master ohjaa tiimin työskentelyä ja pitää huolen ketteristä käytänteistä sekä niiden seuraamisesta. Scrum masteria voidaan pitää eräänlaisena tiimin valmentajana, joka fasilitoi tiimin sisäistä koheesiota, itse-organisoitumista ja optimaalista suoritusta. Hänen tehtävänään on toimia Scrum-asiantuntijana ja huolehtia Scrum-prosessin käytänteiden mukaisesta toiminnasta tiimissä. Scrum masterin vastuualueisiin kuuluu sprintin tavoitteiden toteutumisen seuranta. Scrum master varmistaa, että tehtävää ei merkitä valmiiksi, ennen kuin kaikki siihen määritellyt ehdot ovat toteutuneet (engl. *definition-of-done*, DoD), esimerkiksi koodi kirjoitettu, testitapaukset olemassa ja ajettu onnistuneesti sekä dokumentaatio on päivitetty. Lisäksi scrum

masterin vastuualueella on tiimin toimivuus ja tiimin työtä haittaavien esteiden (engl. impediment) poistaminen. [4, s. 49] Vaikka scrum masterin työtehtävät muistuttavat osin perinteisen projektipäällikön työtehtäviä, on tärkeää huomata, että hän ei kuitenkaan ole esimiesasemassa tiimin muihin jäseniin nähden.

Ohjelmistosuunnittelija

Suunnittelijan keskeinen tehtävä on selvittää, mitä ohjelmiston pitää tehdä ja kuinka se olisi hyvä toteuttaa, niin teknisesti kuin ei-teknisesti. Suunnitteluvaiheessa luodaan perusta ohjelmiston rakenteelle, toiminnoille ja sille, miten se integroituu muihin järjestelmiin. Suunnitteluroolit ovat erilaisia ja kohdistuvat ohjelmiston eri osa-alueisiin esim. arkkitehtuuriin, käyttäjäkokemukseen ja käyttöliittymään tai vaikkapa tietokantoihin. Isoimmissa järjestelmissä korkean tason suunnittelua tekeviä kutsutaan ohjelmistoarkkitehdeiksi. Suunnittelijat käyttävät apunaan usein kaavioita (esim. UML) ja muita työkaluja rakenteen suunnittelussa.

Ohjelmistokehittäjä

Ohjelmistokehittäjät muuntavat suunnitelmat ja vaatimukset toimivaksi ohjelmistoksi kirjoittamalla, testaamalla ja ylläpitämällä koodia. Tosiasiassa useimmat koodaajat lukevat koodia paljon enemmän kuin kirjoittavat itse uutta. Koodin ensimmäisen version kirjoittaminen vie vain murto-osan siitä ajasta, mitä koodin ylläpitämiseen vaaditaan. Ohjelmistokehittäjän työ voi siis olla hyvin vaihteleva riippuen projektista, työn vaiheesta ja tiimin rooleista. Joissakin projekteissa koodari saattaa keskittyä tiukasti tiettyyn kehitysvaiheeseen, kun taas toisissa hän voi olla mukana projektin eri osa-alueilla, kuten suunnittelussa, testauksessa ja käyttöönotossa.

Testaaja

Ohjelmistotestaajan tehtävänä on varmistaa, että kehitetty ohjelmisto toimii suunnitellusti, täyttää määritellyt vaatimukset ja on vapaa virheistä ennen julkaisua. Testaajan rooliin kuuluu monenlaisia tehtäviä, jotka voivat vaihdella projektin luonteen ja kehitysmenetelmien mukaan. Ohjelmistotestauksessa käytetään useita testaustasoja varmistamaan ohjelmiston laatu eri kehitysvaiheissa. Kunkin tason tavoitteena on testata ohjelmistoa eri näkökulmista, alkaen yksittäisistä komponenteista aina koko järjestelmän toimintaan asti. Aloitteleva ohjelmistotes-

taaja tekee tyypillisesti testausta yksikkö-, integraatio- ja systeemitasoilla. Yleensä jokainen ohjelmistokehittäjä tekee itse yksikkötestit kirjoittamalleen koodille.

Tukihenkilö

Ohjelmiston käyttöön tarvitaan monenlaista tukea, koulutusta ja ohjeistusta. Erilaisten tukitoimintojen ja materiaalien avulla varmistetaan, että ohjelmiston käyttäjillä on riittävä taitotaso järjestelmän käyttöön. Koulutusta voidaan järjestää loppukäyttäjien lisäksi esimerkiksi järjestelmän ylläpitäjille ja pääkäyttäjille. Joissain tapauksissa voi olla järkevää pilotoida uutta järjestelmää pienemmällä ryhmällä koekäyttäjiä. Koekäyttäjät voivat toimia asiakkaan organisaatiossa tukihenkilöinä, kun järjestelmä avataan muille työntekijöille. Tukimateriaalit voivat olla muun muassa käyttöohjeita, wikisivuja tai koulutusvideoita. Teknisen tuen materiaaleja laadittaessa on tärkeää huomioida myös saavutettavuus.

1.7 Ohjelmistoprojektin työkalut

Ohjelmistoprojekteissa käytetään erilaisia työkaluja muun muassa suunnittelussa, kehityksessä ja kommunikoinnissa. Työkalujen ja toimintatapojen tarkoituksena on tukea ja sujuvoittaa työskentelyä ohjelmistotiimeissä. Tässä alaluvussa tutustutaan niistä muutamiin.

1.7.1 Kehitystyökalut

Integroidulla kehitysympäristöllä (engl. Integrated Development Environment, IDE) tarkoitetaan kehitystyössä käytettävää ohjelmistoa. Kehitysympäristö tarjoaa useita käytännöllisiä ohjelmoinnin työkaluja, kuten tekstieditorin, kääntäjän/tulkin, debuggausvälineet sekä usein myös versiohallintajärjestelmän. IDE:n käyttö tehostaa ohjelmistokehitysprosessia integroimalla tarvittavat kehitystyökalut samaan ympäristöön. IDE:t helpottavat ohjelmistokehittäjän työtä monin tavoin; syntaksin korostus selkeyttää koodia ja helpottaa sen lukemista, automaattinen täydennys puolestaan nopeuttaa kirjoitustyötä ja vähentää virheitä. Lisäksi monet IDE:t tukevat ohjelmistokehitysprojektin hallintaa, kuten tehtävien seurantaa ja aikataulutusta. Modernit IDE:t ovat usein laajennettavissa ja mukautettavissa lisäosien tai pluginien avulla, jolloin kehittäjät voivat räätälöidä ympäristön vastaamaan omia tarpeitaan tai tietylle ohjelmointikielelle tai teknologiapinolle ominaisia vaatimuksia. Esimerkkejä suosituista IDE:istä ovat Visual

Studio Code, IntelliJ IDEA, Eclipse ja PyCharm, jotka tukevat eri ohjelmointikieliä ja kehitysframeworkeja.

Laajoihin kielimalleihin perustuva tekoäly on viime vuosina yleistynyt ja sitä voidaan hyödyntää ohjelmoinnin tukena monilla tavoilla. Eräs tekoälyä hyödyntävä työkalu on GitHub Copilot, joka käyttää OpenAI:n GPT-3 -mallia ehdottaakseen kokonaisia koodirivejä tai jopa funktioita perustuen käyttäjän kirjoittamaan koodiin ja kommentteihin. Tekoälyä voidaan hyödyntää muun muassa koodin refaktoroinnissa ja optimoinnissa, virheenkorjauksessa sekä kommentoinnissa. Ohjelmointia avustavat työkalut voivat tehostaa ohjelmoinnin vaiheita, mutta eivät poista kokonaisymmärryksen ja ongelmanratkaisun merkitystä keskeisenä ohjelmointiosaamisena.

1.7.2 Versiohallinta ja CI/CD

Ketterien menetelmien ytimessä on iteratiivinen ja inkrementaalinen työskentelytapa, mikä tarkoittaa, että ohjelmistoa kehitetään lyhyissä sykleissä, jotka tuottavat jatkuvasti toimivaa ohjelmistoa. Versionhallinta on tämän toimintatavan mahdollistava järjestelmä, jolla seurataan ja hallinnoidaan ohjelmistoprojektin koodimuutoksia. Linus Torvaldsin vuonna 2005 kehittämä Git on nykyisin yksi suosituimmista versionhallintajärjestelmistä. Git mahdollistaa tavan hallita ohjelmistoprojekteja tarjoamalla ominaisuuksia, kuten haarojen (engl. branch) luomisen, muutosten yhdistämisen (engl. merge) sekä muutoshistorian tarkastelun. Versionhallinta mahdollistaa myös useamman kehittäjän työskentelyn saman lähdekoodin kanssa yhtäaikaisesti. Ilman versionhallintaa muutamaa henkeä isomman kehitystiimin tuottaman lähdekoodin hallinnointi olisi käytännössä mahdotonta.

CI/CD on lyhenne sanoista Continuous Integration (jatkuva integraatio) ja Continuous Delivery/Deployment (jatkuva toimitus/käyttöönotto), jotka ovat keskeisiä käytäntöjä modernissa ohjelmistokehityksessä. CI/CD:n tavoitteena on automatisoida ohjelmistokehityksen ja julkaisun prosessit, mikä nopeuttaa ja tehostaa ohjelmiston toimitusta asiakkaille.

CI viittaa käytäntöön, jossa kehittäjät integroivat koodimuutoksensa yhteiseen säilöön useita kertoja päivässä. Jokaisen koodimuutoksen yhteydessä suoritetaan automatisoituja testejä (esim. yksikkötestejä ja integraatiotestejä), jotka varmistavat, että uusi koodi ei riko olemassa olevaa ohjelmistoa. CI:n tavoitteena on havaita ja korjata virheet mahdollisimman varhaisessa

vaiheessa, mikä vähentää ongelmien monimutkaisuutta ja korjauskustannuksia. Versionhallintatyökalu on CI-järjestelmän ydin.

Nykyaikaisissa ohjelmistoprojekteissa pyritään julkaisemaan uudet muutokset sitä mukaa kun niitä tulee. Uusia ominaisuuksia ja muutoksia voidaan integroida testausympäristössä olevaan tuotteeseen tai jo tuotannossa olevaan. Jatkuva käyttöönotto vie jatkuvan toimituksen askeleen pidemmälle automatisoimalla ohjelmiston julkaisuprosessin niin, että hyväksytyt muutokset siirretään suoraan tuotantoympäristöön ilman manuaalisia vaiheita. Tämä edellyttää laajaa testikattavuutta ja automatisointia varmistamaan, että julkaisut tuotantoon eivät aiheuta ongelmia käyttäjille.

Nykyaikaisen ohjelmistokehityksen web-pohjaisista projektityökaluista suosituimpia ovat GitLab ja GitHub, jotka sisältävät myös työkalut ohjelmiston elinkaaren hallintaan. Molemmat pohjautuvat Git-versionhallintajärjestelmään ja niissä yhdistyy versionhallinta, koodikatselmointi ja CI/CD. Jenkins on myös suosittu web-pohjainen CI/CD:ssä käytettävä sovellus.

1.7.3 Kommunikointityökalut

Korona-ajan jälkeen yleistynyt etä- ja hybridityö on muuttanut ohjelmistoalan työtapoja perustavanlaatuisesti. Etätyöskentely on erityisen yleistä IT-alalla, jossa monien tehtävien luonne mahdollistaa työskentelyn paikasta riippumatta. Etätyön yleistymisen myötä myös kommunikaatiotavat ovat muuttuneet. Perinteisen sähköpostin rinnalle ja korvaajaksi ovat tulleet pikaviestipalvelut, jotka mahdollistavat reaaliaikaisen ja asynkronisen viestinnän. Lisäksi alalla hyödynnetään etätyökäytänteitä, kuten videokokoussovelluksia, jotka mahdollistavat paikkariippumattoman osallistumisen. Esimerkkejä yleisesti käytetyistä etäviestintätyökaluista ovat Teams, Zoom, Slack ja Discord.

1.7.4 Muita työkaluja

Projektin aikana käytetään todennäköisesti myös muita työkaluja, jotka riippuvat mm. projektista ja sen laajuudesta, toimintaympäristöstä ja asiakkaasta. Seuraavissa alaluvuissa on muutamia esimerkkejä työkaluista ja niiden sovellusalueista, joita myös tulevissa luvuissa käsitellään lisää.

Suunnittelu

Ohjelmistojen suunnittelutyössä ja prosessien mallintamisessa hyödynnetään erilaisia mallinnusstandardeja ja työkaluja. Mallinnuksen avulla voidaan esittää järjestelmän rakenne, käyttäytyminen ja eri osien väliset suhteet. Tällä kurssilla perehdytään UML-mallinnukseen, jonka avulla voidaan esittää luokkarakenteita, sekvenssejä ja vuorovaikutuksia ohjelmiston komponenttien välillä. Eräs ohjelmistotuotannossa laajalti käytetty mallinnustyökalu on Visual Paradigm, joka tukee useita mallinnusstandardeja. Pienemmissä projekteissa voidaan hyödyntää mitä tahansa ohjelmaa, josta saa ulos laatikoita ja viivoja. Eräs helppokäyttöinen ilman lisenssimaksuja käytettävissä oleva suunnittelutyökalu on Draw.io, jolla kaavioita voi tehdä näppärästi suoraan selaimessa.

Testaus

Ohjelmistotestauksen työkaluja on olemassa monenlaisiin käyttötarkoituksiin ja testaustarpeisiin. Työkalujen valinta ja käyttö riippuu pitkälti siitä, miten ne sopivat yhteen projektin vaatimusten, käytettävien teknologioiden ja tiimin preferenssien kanssa. Eräs esimerkki suositusta testaustyökalusta on Selenium (WebDriver ja IDE), joka on avoimen lähdekoodin automatisointityökalu web-sovellusten testaukseen. Seleniumiin voidaan liittää kirjastoja ja työkaluja testaustarpeen mukaan. Cucumber on esimerkki työkalusta, joka integroituu saumattomasti Seleniumiin. Sen avulla voidaan kirjoittaa testitapaukset käyttämällä luonnollista kieltä muistuttavaa Gherkin-syntaksia. Tämä tekee testauskäsikirjoituksista ymmärrettäviä myös ei-teknisille sidosryhmille.

Dokumentointi

Dokumentaatiolle on monta tyyliä ja tapaa aina yksinkertaisesta tekstitiedostosta visuaalisiin esityksiin. Tyypillisesti dokumentaation apuna toimivat wiki- ja muut tiedonhallinta-alustat. Nostetaan tähän esimerkiksi Confluence, Atlassianin tuottama wiki-tyyppinen alusta, joka mahdollistaa projektien, ideoiden ja dokumentaation hallinnan ja jakamisen. Myös GitHub-projekteihin on integroitu wiki, joka mahdollistaa ohjelmistoprojektien dokumentaation versionhallinnan ja jakamisen. Projekteissa voidaan käyttää myös perinteisiä tekstinkäsittelyohjelmia (Word, Latex), tekniseen kirjoittamiseen suunnattuja työkaluja, API-dokumentaation

työkaluja ja lähdekoodista automaattisesti dokumentaatiota generoivia työkaluja. Kannattaa muistaa, että dokumentaatio ei ole itseisarvo, vaan työväline, joka auttaa tiimiä, projektia ja tuotetta eteenpäin. Se auttaa tiedon jäsentelyssä ja siihen saa ulkoistettua useiden eri tekijöiden ajatuksia.

1.8 Yhteenveto

Tämä oppikirja ei käsittele ohjelmointia, eikä se ole ohjelmointikirja. Kirjan tavoitteena on ymmärtää kokonaisuus, jonka tuloksena syntyy halutunlainen **ohjelmistotuote** ja hahmottaa yksittäisen ohjelmistokehittäjän roolia tässä kokonaisuudessa. Seuraavissa luvuissa esitellään tarkemmin ohjelmistotuotannon eri vaiheita, näihin liittyviä tehtäviä sekä muita vaikuttavia tekijöitä. Huomaa, että iso osa vaatimuksista on pehmeitä, ne eivät ilmene suoraan koodiriveinä. Nämä ei-toiminnalliset vaatimukset ovat koko ajan työn taustalla ohjaten suunnittelua ja toteutusta, muodostaen ohjelmistotuotannon ammatillisen kokonaisuuden ja yhteiset toimintatavat.

2 Kestävä ohjelmistotuotanto

Kestävässä ohjelmistotuotannossa tavoitteena on huomioida kestävyyden ulottuvuudet eli ympäristölliset, taloudelliset ja sosiaaliset näkökohdat ohjelmistokehityksessä. Tässä luvussa tarkastellaan kestävän ohjelmistokehityksen eettisiä perusteita, vihreän koodauksen käytänteitä ja oikeudenmukaisen saatavuuden varmistamista saavutettavuuden avulla. Kestävän kehityksen huomioiminen ohjelmistoprojektin alusta alkaen auttaa vähentämään ohjelmistoalan ympäristökuormitusta ja tuottamaan ohjelmistoja, jotka tukevat sekä yhteiskunnan että planeetan pitkän aikavälin hyvinvointia.

2.1 Ohjelmistotuotannon etiikka

Etiikka ohjaa kestävän ohjelmistosuunnittelun käytäntöjä. Eettiset näkökohdat tarjoavat kehyksen vastuullisten päätösten tekemiselle, joissa otetaan huomioon teknologian laajempi vaikutus yhteiskuntaan ja ympäristöön.

2.1.1 Etiikan merkitys ohjelmistotuotannossa

Ohjelmistoilla on valtava potentiaali vaikuttaa ihmisten elämänlaatuun. Ohjelmistot ovat keskeisiä monissa yhteiskunnan toiminnoissa ja niillä on näin ollen vaikuttava rooli ihmisten arjessa ja elämänkuluissa. Tämä on eräs syy, miksi on tärkeää, että eettiset näkökohdat otetaan huomioon ohjelmistojen suunnittelussa ja kehittämisessä. Eettiset näkökohdat luovat perustan myös myöhemmin tässä luvussa esiteltävälle kestävälle ohjelmistokehitykselle.

Jokaisen ohjelmistokehittäjän on hyvä pysähtyä pohtimaan sitä, mitä etiikka tarkoittaa käytännössä ohjelmistoprojektia toteutettaessa. Eettisesti latautuneiden tilanteiden tunnistaminen edellyttää ensinnäkin aiheen tiedostamista ja lisäksi herkkyyttä tunnistaa eettiset kysymykset ja haasteet eri asiayhteyksissä. Sidosryhmien ja niiden roolin ymmärtäminen ohjelmistokehi-

tyksen kontekstissa on ensiarvoisen tärkeää. Tarkastelua vaativien aihepiirien tunnistamista helpottaa myös tiettyjen käsitteiden, kuten yksityisyyden suoja, tietoturva, tekijänoikeudet, automaatio, turvallisuus, kestävä kehitys, energiankulutus, tekoäly jne. eettisten ulottuvuuksien hahmottaminen alan diskurssissa.

2.2 Ohjelmistokehityksen eettiset ulottuvuudet

Kuten todettua, etiikka on olennainen osa ohjelmistosuunnittelun vastuukenttää ja ohjelmistoalalla työskentelevän ammatillista osaamista. Eettisiä periaatteita noudattava ohjelmistokehittäjä osaa ottaa työssään huomioon avoimuuden, vastuullisuuden ja muut työn kannalta relevantit eettiset näkökohdat. Eettisten lähtökohtien huomioiminen ja aktiivinen esiintuominen ohjelmistoprojektissa luo luottamusta työyhteisön, asiakkaiden, käyttäjien ja yhteistyökumppaneiden keskuudessa ja muodostaa vankan perustan pitkän aikavälin menestykselle. Etiikan tiedostaminen näkyy myös ohjelmistojen ominaisuuksissa ja toiminnallisuuksissa; niihin on sisäänrakennettu haitallisten vaikutusten ja riskien hallinta, jolla pyritään esimerkiksi suojaamaan käyttäjien yksityisyyttä.

Ohjelmistoalan eräs haaste ovat henkilöt tai ryhmät, jotka tiedostaen tai tietämättään laiminlyövät eettisiä periaatteita ja ammatillisia velvollisuuksiaan aiheuttaen riskejä ja ongelmia. Tällaiset laiminlyönnit eivät ainoastaan riko eettisiä normeja, vaan niillä voi olla vakavia seurauksia; esimerkiksi kriittisten ohjelmistojärjestelmien vikaantuminen voi aiheuttaa henkilö- tai omaisuusvahinkoja [9], asettaen vastuullisille merkittävän moraalisen taakan [10]. Ei ole yhdentekevää, millaisia ohjelmistoja luodaan ja miten ihmiset niitä käyttävät, sillä ohjelmistoihin liittyy usein laajempi yhteiskunnallinen ulottuvuus. Moraalisen vastuunsa tiedostavan kehittäjän tulisi pyrkiä työssään välttämään sellaisia ratkaisuja, jotka luovat teknisiä ja yhteiskunnallisia vinoumia ja epätasa-arvoa [11]. Ohjelmistoilla on laaja-alaisia vaikutuksia erilaisiin yhteiskunnallisiin ilmiöihin, kuten työllisyyteen, talouteen, koulutukseen, lasten kehitykseen ja demokratiaan. Ohjelmistoalan sitoutuminen eettisesti vastuulliseen toimintaan auttaa muun muassa ennaltaehkäisemään syrjintää, valheellisen tiedon leviämistä, haitallista seurantaa ja väärinkäyttöä. Eettisten periaatteiden mukaan toimiminen edistää sosiaalista hyvinvointia ja oikeudenmukaisuutta myös laajemmin yhteiskunnassa.

Ohjelmistokehityksessä ja erityisesti sidosryhmäviestinnässä on hyvä noudattaa läpinäkyvyyden periaatetta. Erityisen tärkeää on viestiä ohjelmiston tarkoitus ja toimintaperiaatteet ilman salattuja tarkoitusperiä. Esimerkkinä tällaisista epämääräisistä tarkoitusperistä ovat pimeät suunnittelumallit (engl. dark patterns), jotka ovat verkkosivustoissa ja sovelluksissa käytettyjä suunnittelutapoja, joilla käyttäjää pyritään harhauttamaan tai houkuttelemaan tekemään itselleen epäedullisia valintoja. Eräs pimeiden suunnittelumallien ilmenemismuodoista ovat verkkosivustojen evästeiden suostumusbannerit, jotka on mahdollista suunnitella siten, että käyttäjä huijataan antamaan suostumus tiedonkeruuseen.

Eettisestä näkökulmasta erityistä huomiota tulee kiinnittää tilanteisiin, jotka liittyvät lapsiin, terveyteen, valtasuhteisiin, yhteiskunnan ydintoimintoihin, seksuaalisuuteen, ympäristökysymyksiin sekä perus- ja ihmisoikeuksiin. Eettisesti latautuneiden tilanteiden ja ongelmien arvioinnissa voidaan hyödyntää eettisten teorioiden lisäksi toiminta-alueella sovellettavia lakeja. On tärkeää huomata, että eettinen toiminta ohjelmistosuunnittelussa ei ole vain hyvä käytäntö, vaan se on usein myös lakisääteinen vaatimus. Valtiot ja niiden lainsäädännön puitteissa toimivat organisaatiot on velvoitettu noudattamaan tietoturvaan, yksityisyyteen ja tietosuojaan liittyvää regulaatiota, joka edellyttää eettistä toimintaa ohjelmistojen suunnittelussa ja käytössä. Huonot eettiset valinnat voivat altistaa yrityksen monimutkaisten vastuukysymysten lisäksi mainehaitalle.

2.3 Eettinen ohjelmistotuotanto

Eettinen ohjelmistotuotanto on ennen kaikkea tietoon perustuvaa päätöksentekoa, joka huomioi teknologian laajemmat vaikutukset yhteiskunnassa. Ymmärrys kestävien valintojen merkityksestä ja näiden periaatteiden soveltaminen ohjelmistokehitykseen antavat ohjelmistokehittäjille mahdollisuuden tehdä ympäristön, talouden ja sosiaalisen kestävyyden kannalta toimivia ohjelmistoja, joissa ohjelmiston suorituskyky, toiminnallisuus ja energiatehokkuus ovat tasapainossa.

2.3.1 Ohjelmistotuotannon eettiset haasteet

Eettiset kysymykset näyttäytyvät eri tavoin riippuen ohjelmistotuotantoprosessin vaiheesta. Hankkeen alussa ei useinkaan olla vielä oikeissa ongelmissa, ja tällöin puhutaankin eettisesti

latautuneista tilanteista, joissa esiintyy jonkinasteista eettistä harkintaa ja arviointia vaativaa problematiikkaa. Tällaisia haasteita voi syntyä monenlaisissa konteksteissa, ja ne on otettava huomioon päätöksenteossa. Eettinen ongelma taas viittaa konkreettiseen eettiseen pulmaan tai ristiriitaan – tilanteeseen, joka on tiedostettu ja joka olisi suotavaa ratkaista. Ohjelmistokehittäjät ovat vastuussa eettisten normien noudattamisesta ja käyttäjien oikeuksien kunnioittamisesta riippumatta teknologian muuttuvista suuntauksista, mutta silti sopivan toimintatavan valinta voi olla haastavaa. Seuraavaksi tarkastellaan joitakin ohjelmistokehityksen eettisiä pulmia [12].

Epäeettinen tietojen kerääminen

Digitaalisen markkinoinnin yleistyminen on lisännyt eri järjestelmien käyttäjistä kerätyn tiedon arvoa. Käyttäjädata on myös tärkeää tiettyjen ohjelmistotuotteiden kehittämisessä. Tämän takia on tärkeää, että käyttäjät ovat yksiselitteisen tietoisia siitä, mitä tietoja he jakavat ja miten niitä käytetään. Ohjelmistokehityksen parissa työskentelevät voivat kohdata käyttäjätietojen käsittelyyn, tietosuojaan ja yksityisyyden suojaan liittyviä ristiriitoja, varsinkin jos käsiteltävä tieto on arkaluonteista. Tällöin henkilö voi joutua työssään valinnan eteen: hyödyntääkö käyttäjätietoja järjestelmän kehittämisessä vai tuodako esiin tietojen väärinkäyttöön liittyvät huolenaiheet. Eettisiä ongelmia voi ilmetä myös teknologioiden ominaisuuksiin ja käyttöön liittyen, tällaisia ovat esimerkiksi tasapainoilu sovelluksen energiankäytön ja ympäristövaikutusten välillä sekä tietyn teknologian tai sovelluksen laajemmassa tarkastelussa ilmenevät epäsuorat ja rakenteelliset vaikutukset.

2.3.2 Algoritmien aiheuttamat vinoumat

Tietokoneilla ei ole moraalikäsitystä. Algoritminen vinouma eli biasoituminen (engl. algorithmic bias) voi aiheutua tahattomasti, jos asiaa ei oteta huolellisesti huomioon kehitystyön aikana. Esimerkiksi Google keräsi runsaasti negatiivista huomiota Gemini-tekoälyn tuottamien historia-aiheisten henkilökuvien poikkeavasta inklusiivisuudesta. Tämä seurauksena yhtiö joutui estämään Geminiä luomasta enempää vastaavia kuvia [13]. Erityisesti tekoälyn parissa työskentelyssä tulee ottaa huomioon biasoitumisen mahdollisuus. Tämä vaatii sekä ymmärrystä sosiaalisista normeista että datan käytön kriittistä arviointia.

2.3.3 Turvallisuusheikkoudet

Ohjelmistoihin kohdistuu monia tietoturvahaasteita. Tämä edellyttää kehittäjiltä oman osaamisalansa tietoturvakäytänteiden hallintaa ja osaamisen jatkuvaa päivittämistä. Tietoturvauhista ja niiden torjuntakeinoista on tärkeää pysyä ajan tasalla ja ohjelmistojen turvatoimista huolehdittava koko ohjelmiston elinkaaren ajan.

2.3.4 Prioriteettien määrittelemisen ongelmat

Ohjelmistokehitys keskittyy liian usein uusien ominaisuuksien kehittämiseen olemassa olevien parantamisen kustannuksella. Tämä on kestävyyden näkökulmasta problemaattista ja voi johtaa monenlaisiin eettisiin ongelmiin. Useat etiikan niin sanotuista dilemmoista koskevat kuitenkin hyvin kapeaa ohjelmistokehityksen aluetta. Ne ovat pikemminkin kysymyksiä siitä, missä menevät ohjelmistokehittäjän henkilökohtaisen moraalin rajat. Esimerkkeinä voidaan mainita Googlen työntekijöiden vastalauseet yhtiön yhteistyöstä Yhdysvaltain armeijan kanssa, kasvojentunnistusjärjestelmät tai osallistuminen itseohjautuvien asejärjestelmien kehittämiseen. Eettisten ongelmien ratkaiseminen edellyttää pohdintaa ja arviointia, jossa tasapainoillaan useiden eri arvojen ja periaatteiden välillä. Tämä on erityisen haastavaa, sillä usein ei ole olemassa yhtä ainoaa oikeaa ratkaisua.

2.3.5

Esimerkki: **Eettisesti latautunut tilanne.** Olet uusi työntekijä koeajalla ohjelmistoalan yrityksessä. Asiakas pyytää sähköpostiviestillä (cc esimiehellesi) sinua tekemään heille asiakasrekisterin, jonka huomaat olevan mahdollisesti tietosuojalainsäädännön vastainen ja saattavan heidän asiakkaidensa yksityisyyden vaaraan. Tarvitset tätä työpaikkaa. Kysymys 1: Miten toimit?

Esimerkkilaatikko: Kysymys 2: Olet vastaamassa edelliseen sähköpostiviestiin, kun esimiehesi vastaa sinulle ja asiakkaalle ja lupaa, että toteutat tämän rekisterin pyynnön mukaisesti. Mitä teet?

Kysymysboksi: Miten nämä asiat suhteutuvat kestävään ohjelmistokehitykseen?

2.4 Kestävä ohjelmistokehitys

Ohjelmistokehitys käyttää nykyään huomattavan määrän ekologisia, taloudellisia ja sosiaalisia resursseja, ja siksi on tärkeää, että näiden resurssien hyödyntämisessä käytetään eettistä harkintaa. Kestävällä ohjelmistosuunnittelulla pyritään vähentämään ohjelmistokehityksen haitallisia vaikutuksia koko yhteiskuntaan [14].

2.4.1 Ympäristövastuu

Ohjelmistokehittäjien vastuulla on minimoida työnsä ympäristövaikutuksia optimoimalla mahdollisuuksien mukaan ohjelmistotuotteiden kehittämisestä ja niiden käytöstä aiheutuvaa energiankulutusta. IT-sektorin kokonaisenergiankulutus muodostuu erilaisten kuluttajalaitteiden (mm. tietokoneet, mobiililaitteet ja pelikonsolit) ja yhteiskunnan ja yrityssektorin järjestelmistä (mm. voimalaitokset, datakeskukset, supertietokoneet/suurteholaskenta). Nämä järjestelmät aiheuttavat kasvihuonepäästöjä ja monissa tapauksissa laskentatehokkuuden ja sitä kautta energiatehokkuuden parantaminen voi tuoda mittavia säästöjä niin taloudellisesti kuin ympäristökuormituksenkin kannalta. Eettisyyden näkökulmasta on tärkeää, että jokainen ohjelmistosuunnittelija koodaa ympäristön kannalta kestävästi. Samalla tietoisuus ja osaaminen kestävän koodauksen praktiikoista tulee hiljalleen osaksi toimialan käytänteitä.

2.4.2 Vihreä koodaus

Käytännön vihreä ohjelmistosuunnittelu tarkoittaa muun muassa tehokkaampien ja vähemmän resursseja käyttävien algoritmien luomista. Tehokkaat algoritmit kuluttavat vähemmän resursseja muun muassa vähentämällä suoraan muistin käyttöä ja siten sovelluksen energiankulutusta. Mikäli asiakasvaatimusten puolesta on mahdollista, kehittäjän kannattaa valita mahdollisimman **energiatehokas ohjelmointikieli**. Palvelujen välinen tiedonsiirto keskeinen on osa nykyisenlaisia tietojärjestelmiä. Resurssitehokkaissa toimintatavoissa pyritäänkin **optimoimaan tiedonsiirron käyttämät resurssit** esimerkiksi pakkaamalla tiedot ennen siirtoa ja minimoimalla tarpeettomat tiedonsiirrot. Tämä vähentää laskennallista kuormitusta, mikä vähentää energiankulutusta. Myös **koodin tehokkuuden** parantamisella on tärkeä rooli. Hyvin suunniteltu koodi vaatii vähemmän laskentatehoa ja kuluttaa näin ollen vähemmän energiaa.

Tämän saavuttamiseksi voidaan käyttää seuraavia lähestymistapoja. *Epäedullisten ohjelmointiratkaisujen välttäminen* on käytäntö, jossa vältetään tehotonta koodausta alusta alkaen. Tällöin pyritään lähtökohtaisesti kirjoittamaan koodia välttämällä tarpeetonta monimutkaisuutta ja resurssi-intensiivisyyttä. *Algoritmien optimointi* tarkoittaa algoritmien toimintalogiikan ja tehokkuuden parantamista laskennallisen monimutkaisuuden vähentämiseksi, mikä puolestaan vähentää käytön aikana käytettävää energiaa. *Silmukoiden optimointiin* kuuluu toistuvasta silmukan suorittamisesta aiheutuvien kustannusten minimointi eri menetelmien avulla, esimerkiksi välttämällä silmukoiden välisiä riippuvuuksia. *Rinnakkaisprosessointi* voi parantaa huomattavasti suoritusnopeutta ja vähentää energiankulutusta, eli tehtävät kannattaa mahdollisuuksien mukaan jakaa useille ytimille.

2.4.3 Vihreä UI/UX-suunnittelu

Vihreä ohjelmistokehitys on laaja aihealue, jossa tekninen optimointi muodostaa vain yhden osan. Toinen osa-alue on käyttäjäkokemuksen optimointi, jonka keinoin voidaan tukea järjestelmän kestävämpää käyttöä ja säästää käyttäjien aikaa mielekkäämpiin toimintoihin. Käyttäjäystävällisten, mutta myös energiatehokkaiden käyttöliittymien luominen on tärkeässä asemassa kestävässä ohjelmistokehityksessä. Liialliset animaatiot ja monimutkaiset käyttöliittymät lisäävät väistämättä sivuston tai ohjelmiston virrankulutusta. Toisinaan UI/UX-elementtien organisoimaton ja liiallinen käyttö voi jopa heikentää käyttäjäkokemusta. Pimeiden suunnittelumallien välttäminen on oleellinen osa kestävää UI/UX-suunnittelua. Harhaanjohtavat käytännöt eivät ole ainoastaan epäeettisiä ja joskus jopa laittomia, vaan ne myös vaikuttavat energiankulutukseen. Ensinnäkin, pimeiden suunnittelumallien avulla voidaan erehdyttää käyttäjä antamaan suostumuksensa tiedonkeruuseen, mikä lisää verkkosivujen analytiikan käyttöä. Toiseksi ne lisäävät tarpeettomia elementtejä ja monimutkaistavat käyttöliittymää, jolloin käyttäjät käyttävät sivustolla enemmän aikaa ja klikkauksia, mikä lisää energiankulutusta entisestään.

2.4.4 Virrankulutuksen mittaaminen

Ilman keinoja analysoida objektiivisesti ohjelmistojärjestelmän virrankulutusta voi olla vaikea parantaa järjestelmän ympäristöystävällisyyttä, puhumattakaan lisäoptimoinnin tarpeiden tunnistamisesta. Ohjelmiston käyttämän energian mittaamisessa voidaankin käyttää sekä lait-

teistoja (esim. AC-mittarit, tasavirtalähteeseen liitetyt mittarit, integroidut tehonmittauspiirit) että ohjelmistotyökaluja (esim.: Intel PCM[1], Syspower[2], Windows E3[3], Website Carbon Calculator[4]). Green Algorithms [5] on esimerkki Lannelongue et al. vuonna 2021 kehittämästä vapaasti saatavilla olevasta online-työkalusta ohjelmistojen hiilijalanjäljen arvioimiseksi. [15]

- Hiilijalanjälki ja Co2-päästöjen mittaaminen

Hiilijalanjälki kuvaa ihmisen toiminnan aikaansaamaa ilmastovaikutusta, joka määritellään yleisesti hiilidioksidin (CO2) ja metaanin (CH4) kokonaispäästöiksi. Hiililaskenta on menetelmä, jolla voidaan arvioida ryhmän, järjestelmän tai toiminnan aiheuttamia kasvihuonekaasupäästöjä.[16] Elinkaariarviointi (Life-Cycle Assessment, LCA) [17] ja GHG-protokolla (Greenhouse Gas Protocol, GHG) [18] ovat kaksi yleistä menetelmää tuotteiden ja prosessien hiilijalanjäljen laskemiseen. LCA:ta käytetään usein yksittäisten tuotteiden ja prosessien arviointiin, GHG puolestaan on maailmanlaajuinen standardi, joka on suunniteltu valtioita ja suuria yrityksiä varten. Vaikka laskelmissa keskitytään joskus pelkästään hiilidioksidipäästöihin, on tavallisempaa ottaa huomioon myös muiden kasvihuonekaasujen osuus osana kokonaishiilijalanjälkeä. On huomattava, että monissa tapauksissa suurin osa ohjelmistojärjestelmän aiheuttamista hiilidioksidipäästöistä liittyy järjestelmän toiminnan aikaiseen energiankulutukseen.Pilvisovellusten energiamittaukseen löytyy omia työkaluja, kuten Googlen Carbon Footprint [6] ja AWS:n Customer Carbon Footprint Tool [7].

2.5 Ohjelmistotuotteiden saavutettavuus

Saavutettavuuden sisällyttäminen ohjelmistojen kehitysprosessiin on tärkeä osa ohjelmistojen sosiaalista kestävyyttä. Saavutettavuuden näkökulmien huomiointi tuo erilaisten käyttäjien tarpeet ja lähtökohdat näkyväksi kaikissa kehitysvaiheissa suunnittelusta toteutukseen ja ylläpi-

[1] Intel PCM

[2] Syspower

[3] Windows E3

[4] Website Carbon Calculator

[5] Green Algorithms

[6] Google: Carbon Footprint

[7] AWS: Customer Carbon Footprint Tool

toon. On eettisesti ja sosiaalisesti perusteltua, että myös ohjelmistokehityksen parissa edistetään yhdenvertaisuutta ja oikeudenmukaisuutta huomioimalla eri ikäiset ja taustaiset toimijat. Osallistavan suunnittelun tavoitteena on tehdä tuotteista helppokäyttöisiä ja käyttökelpoisia erilaisista taustoista tuleville ja erilaisin kyvykkyyksin toimiville ihmisille [19]. Lähestymistavassa otetaan huomioon monenlaiset käyttäjien tarpeet ja mieltymykset koko suunnittelu- ja kehitysprosessin ajan. Suuret yritykset, kuten Microsoft, Apple ja Google hyödyntävät laajasti osallistavan suunnittelun menetelmiä, mikä osoittaa käyttäjälähtöisyyden merkityksellisyyttä menestyksekkäiden ja käyttäjäystävällisten tuotteiden luomisessa. Inklusiivisessa suunnittelussa ei ole kysymys ainoastaan psyykkisten ja fyysisten rajoitusten huomioimisesta, vaan se ja kattaa laajan skaalan yhteiskunnallisen monimuotoisuuden ilmenemistapoja. World Wide Web Consortiumin (W3C) Web Accessibility Initiative (WAI) on laatinut Web Con- tent Accessibility Guidelines (WCAG) -ohjeet [20], joiden tarkoituksena on parantaa verkkosisällön saavutettavuutta vammaisten näkökulmasta. WCAG perustuu neljään perusperiaatteeseen, joihin viitataan usein lyhenteellä POUR (Perceivable, Operable, Understandable ja Robust). Tämä määrittely pitää sisällään havaittavuuden, hallittavuuden, ymmärrettävyyden ja toimintavarmuuden saavutettavuusvaatimukset. Näillä periaatteilla varmistetaan sisällön saavutettavuus ja mahdollisimman monen ihmisen mahdollisimman sujuva käyttö. esimerkiksi *turku.fi* [8] -sivusto tarjoaa kontrastisuhteen muuttamisen, eli tekstin ja taustavärien välillä on riittävä kontrasti sekä vaihtoehtoisia fonttikokoja sivustolla navigoimiseen. Jotta saavutettavuus voidaan sisällyttää tehokkaasti osaksi ohjelmiston elinkaaren hallintaa, ohjelmistosuunnittelutyössä on kiinnitettävä huomioita seuraaviin seikkoihin:

1. Tutustu tarkasti kohderyhmän ominaispiirteisiin ja huomio tarpeet ja rajoitteet. On tärkeää, että erilaiset käyttäjät pääsevät sisältöön käsiksi ja pystyvät käyttämään toimintoja rajoitteistaan huolimatta.

2. Tutustu WCAG-ohjeistukseen ja hyödynnä mahdollisuuksien mukaan osallistavaa suunnittelua. Huomioi saavutettavuus aktiivisesti kehitysprosessin kaikissa vaiheissa.

3. Osallistu ja kannusta saavutettavuusosaamisen parantamiseen oppimisen ja opiskelun kautta. Kun saavutettavuusnäkökohdat sisällytetään perus- ja ammatillisen täydennys-

[8] turku.fi

koulutukseen, tulevilla kehittäjillä on lähtökohtaisesti tarvittavat taidot inklusiiviseen sovelluskehitykseen.

3 Ohjelmistoprojektin aloitus

Tämän kurssin pyrkimyksenä on perehdyttää lukija ohjelmistotuotannon keskeisiin teemoihin kronologisessa järjestyksessä, mistä syystä tässä osiossa käsittelemmekin aiheita jotka liittyvät ensisijaisesti ohjelmistoprojektin alkumetreihin. Tästä syystä suuri osa aihealueista käsittelee ensisijaisesti organisatorisia ja kommunikaatioon liittyviä kysymyksiä, teknisen totetutuksen ja sen suunnittelun jäädessä pitkälti luvun 4 aiheiksi.

3.1 Asiakkaan taustojen selvittäminen

Kotitehtävien tekeminen on keskeistä myös ohjelmistoprojektin osalta. Projektin alkuvaiheessa on hyvä selvittää potentiaalisen asiakkaan taustatiedot. Tämä esitutkimus tarjoaa pohjan, jolta rakentaa syvällisempi ymmärrys asiakkaasta ohjelmistoprojektin aloituksen aikana. Yksityiskohtaisempien tietojen kerääminen tässä vaiheessa auttaa projektitiimiä ymmärtämään asiakkaan tarpeita paremmin, säästämään aikaa ja rajoittamaan väärinkäsitysten mahdollisuutta.

On olennaista tuntea yrityksen perustiedot, mukaan lukien sen koko, toimiala, sijainti, perustamisvuosi ja työntekijämäärä. Lisäksi yrityksen liiketoimintamallin ymmärtäminen – kenelle tuotteet tai palvelut on suunnattu, arvolupaus ja ansaintalogiikka – on tärkeää, jotta voidaan ymmärtää, kuinka ohjelmiston tulisi tukea yrityksen tavoitteita. On siis ymmärrettävä, kenelle tuotetta ollaan tekemässä.

Myös yrityksen toimialasta olisi hyvä kerätä ymmärrystä. Toimialan nykyiset trendit, lainsäädännölliset vaatimukset sekä alan erityiset haasteet ja mahdollisuudet ovat kaikki tekijöitä, jotka voivat vaikuttaa ohjelmistoprojektin suunnitteluun ja toteutukseen. Tämä tieto auttaa tunnistamaan asiakkaan kanssa työskenneltäessä mahdolliset esteet ja mahdollisuudet ajoissa. Myös asiakasyrityksen kilpailijoiden tarjoamien tuotteiden tai palveluiden tunteminen, sekä

heidän käyttämänsä teknologiat, tarjoavat arvokkaita viitteitä siitä, millaisia innovaatioita tai parannuksia asiakas saattaa kaivata. Taustoja voi kysellä myös omilta verkostoilta sekä kerätä avoimista tietolähteistä. Yrityksen verkkosivujen, sosiaalisen median ja uutisartikkelien tutkiminen voi antaa tietoa yrityksen nykytilasta, ohjelmistotarpeista sekä antaa arvokkaita oivalluksia yrityksen kulttuurista, arvoista sekä siitä, miten se kommunikoi asiakkaidensa ja sidosryhmiensä kanssa.

3.2 Myy osaaminen / idea

Kun on idea siitä miten jokin asia pitäisi toteuttaa, se pitää myydä potentiaalisille asiakkaille. Asiakkaalla tässä merkityksessä ei tarkoiteta pelkästään maksavia asiakkaita, vaan yleisesti ottaen niitä tahoja joiden hyväksynnästä on kiinni se tartutaanko tähän ideaan ja lähdetään työstämään sitä. Asiakas voi siis olla myös oman organisaation jäsen, esim. esimies tai Product Owner tai oikeastaan kuka tahansa, joka on asemassa jossa tehdään päätöksiä siitä miten projekti toteutetaan.

Asia on nimittäin niin, että loistavia ideoita maailma täynnä. Vain pieni osa niistä ideoista koskaan pääsee toteutusvaiheeseen, ja tämä on hyvin vahvasti kiinni siitä miten hyvin asia osataan esittää muulle organisaatiolle eli potentiaalisille asiakkaille, niin että myös nämä tahot kokevat että juuri tämä idea pitäisi toteuttaa. Ikävä fakta on myös se, että idean hyvyys itsessään ei useinkaan riitä siihen että se saisi vihreää valoa päätöksiä tekevien tahojen mielessä, vaan se pitää osata esittää tavalla joka on vakuuttava ja “myyvä”, niin että myös muut näkevät arvon tässä ideassa. Käymme seuraavaksi läpi joitakin keskeisiä askelia siinä miten tämä onnistuu.

- **Idean sitominen organisaation laajempaan kontekstiin ja päämääriin:** Tämä kiteytyy hyvin pitkälti siihen, että idean pitää olla jotain mikä edistää organisaation yleistä päämäärää ja tavoitteita. Ei ole välttämättä turhaa yrittää laajentaa näitä päämäriä, mutta se on huomattavasti haastavampaa kuin saada läpi sellainen idea joka ruokkii niitä.

- **Osoita selvää ymmärrystä siitä mihin ongelmaan tämä idea vastaa:** Ratkaisua ongelmaan jota organisaatio ei tunnista selvästi, ja jota et itsekään osaa selittää järkevästi, on äärimmäisen vaikea saada hyväksyttyä.

- **Tiivistä idea helposti ymmärrettävään muotoon:** Idea pitää osata tiivistää niin selkeästi että sen keskeiset puolet voi esittää "viidessä minuutissa", ennen kuin kohteen mielenkiinto katoaa.
- **Esitä järkevä arvio tarvittavasta ajasta ja resursseista:** Järkevän työsuunnitelman laatiminen osoittaa muille että olet paneutunut tähän ongelmaan ja miettinyt sitä ajan kanssa.
- **Selitä millä metriikalla projektin onnistuminen arvioidaan:** Raakojen faktojen esittäminen siitä miten idean toteutumista voidaan arvioida edistää valtavasti sitä että päätöksiä tekevät tahot haluavat tarttua siihen.
- **Ole rehellinen epäselvistä puolista ja idean heikkouksista:** Älä valehtele, jos jokin puoli ideassa ei ole täysin selvillä (koska se ei välttämättä voikaan olla ideointivaiheessa), vaan myönnä suoraan mahdolliset tuntemattomat tekijät jos niitä on.
- **Osaa selittää idea ilman powerpoint-slideja ja muita apuvälineitä:** Sinun täytyy tuntea ideasi niin hyvin että osaat tarvittaessa selittää sen ilman mitään apuvälineitä siitä kiinnostuneille.
- **Wow-faktori:** Ei haittaa että ideaan saa lisättyä jonkin sellaisen elementin joka vetoaa kuulijoiden tunteisiin ja saa heidät ajattelemaan että tämä on paras asia sitten valmiiksi leikatun leivän.

3.3 Tunnista sidosryhmät

Ohjelmistotekniikan kontekstissa sidosryhmät ovat ihmisiä, ihmisryhmiä tai organisaatioita, jotka liittyvät jollain tavalla ohjelmistoprojektiin. Sidosryhmät voivat olla sekä sisäisiä että ulkoisia. Sisäiset sidosryhmät liittyvät suoraan organisaatioon tai projektiin, kun taas ulkoiset sidosryhmät ovat organisaation tai projektin ulkopuolella olevia tahoja. Sidosryhmien tunnistaminen on prosessi, jossa pyritään tunnistamaan ja ymmärtämään ne henkilöt, ryhmät ja organisaatiot, jotka liittyvät projektiin tai joihin projektin toiminta voi vaikuttaa. Sidosryhmien tunnistaminen on keskeinen osa projektinhallintaa ja suunnittelua, sillä se edesauttaa varmistamaan kaikkien asianosaisten huomioon ottamisen sekä tarpeiden ja odotusten täyttämisen.

Ohjelmistoprojektin sidosryhmiä voivat olla esimerkiksi seuraavat:

- Asiakkaat: Ohjelmiston käyttäjät tai organisaatiot, jotka tilaavat tai hyödyntävät ohjelmistoa. Asiakkaat voivat olla loppukäyttäjiä tai muita organisaatioita.
- Asiakkaan edustaja: Asiakkaan edustajalla voi olla edustamansa tahon kanssa eriäviä tavoitteita.
- Käyttäjät: Henkilöt tai organisaatiot, jotka lopulta käyttävät ohjelmistoa. Käyttäjien tarpeiden ja odotusten ymmärtäminen on tärkeää ohjelmiston suunnittelussa ja kehityksessä.
- Käytön kohteet: Ohjelmiston tarkoituksena voi olla hallinnoida jonkin ihmisryhmän tietoja tai heidän jokapäiväistä elämäänsä, esimerkiksi vanhainkodissa, HR-osastolla tai vankilassa.
- Kansalaiset: Jos kyseessä on julkishallinnon projekti, sen maksaja on usein eri taho kuin projektin asiakas.
- Osakkeenomistajat ja yrityksen johto: Osakeyhtiön toiminnan tarkoituksena on tuottaa voittoa osakkeenomistajille, jollei yhtiöjärjestyksessä määrätä toisin.
- Projektiryhmä: Ohjelmiston kehitykseen osallistuvat tiimin jäsenet, kuten ohjelmistosuunnittelijat, ohjelmoijat, testaajat ja projektipäällikkö. Projektiryhmä on myös tärkeä sidosryhmä, joka vaikuttaa ohjelmiston kehitysprosessiin.
- Projektin omistaja: Projektissa esimiesasemassa olevilla henkilöillä on toisinaan eriäviä tavoitteita projektin henkilöstön kanssa.
- Sidosryhmäorganisaatiot: Muut organisaatiot tai sidosryhmät, jotka ovat jollain tavalla liitoksissa ohjelmistoprojektiin. Esimerkiksi integraatiokumppanit, alihankkijat, sertifiointiviranomaiset tai sääntelyelimet voivat olla tärkeitä sidosryhmiä.
- Ylläpitäjät ja muut asiantuntijat: Henkilöt tai organisaatiot, jotka vastaavat ohjelmiston ylläpidosta, tietoturvasta, päivityksistä ja jatkuvasta tuesta sen käytön jälkeen.

Sidosryhmien tunnistamiseksi on ymmärrettävä niin projektin kulku kuin ohjelmistotuotteen toimintaympäristö. Projektiin välittömästi osallistuvien tahojen luokittelu ryhmiksi onkin hyvä tapa aloittaa sidosryhmäanalyysi. Tämän jälkeen on suotavaa aloittaa laajempi analyysi ohjelmiston vaikutuksista sen toteutettavaan ympäristöön. Tässä voidaan hyödyntää kyselytutkimuksesta tuttuja menetelmiä kartoittaakseen organisaation rakennetta sekä ohjelmistotuotteen oletettua vaikutusta todellisuuteen. Tärkeää on myös selvittää, mitä kukin sidosryhmä odottaa projektista tai ohjelmistosta, jotta heidän tarpeensa voidaan ottaa huomioon. Ammattitaitoa on osata erottaa toivomukset tarpeista.

Merkittävimmissä tai suurempia muutoksia tekevissä hankkeissa on hyvä harkita sidosryhmien sitouttamista ja osallistamista hankkeeseen jo alkuvaiheessa, jolloin voidaan välttää sekä suunnitteluvirheitä että käyttäjävastarintaa. Sidosryhmien tunnistaminen on jatkuva prosessi, ja sidosryhmäluetteloa tulee päivittää tarpeen mukaan projektin edetessä. Näin varmistetaan, että sidosryhmät otetaan huomioon koko projektin ajan ja että heidän tarpeensa ja odotuksensa täytetään.

3.4 Toteutusteknologian kartoitus

Ohjelmistoprojektia suunnitellessa toteutusteknologiat määräytyvät pitkälti, mutta eivät täysin, sen mukaan mitä ollaan tekemässä. Projektin yleinen luonne, esim. tehdäänkö inventaarionhallintajärjestelmää logistiikka-alalle vai julkaisujärjestelmää medialle, toimiiko systeemi verkon yli vai lokaalisti, kuinka montaa samanaikaista käyttäjää järjestelmän pitää kestää, kuinka suuria datamassoja järjestelmä käsittelee yms. määrittää ja rajoittaa mahdollista käytettyä teknologiaa tietyllä tavalla. Jotkin viitekehykset soveltuvat ominaisuuksiensa puolesta paremmin isojen järjestelmien rakentamiseen, toiset taas paremmin pienten ja keskisuurien. Jos systeemissä pyörivät datamäärät ovat valtavia pitää backend -toteutuksen perustua johonkin sellaiseen kieleen ja tietokantatyyppiin jotka on hyvin optimoitu tällaista kuormitusta varten jne.

- **Skaalautuvuus:** Hyvin keskeinen asia jota tulee miettiä toteutusteknologiaa valitessa, erityisesti silloin kun ollaan rakentamassa järjestelmää jonka käyttäjämäärät ja/tai käsitellyn datan määrä kasvavat jatkuvasti on skaalautuvuus, eli se miten käytetty teknologia soveltuu näiden muuttuvien olosuhteiden aiheuttamiin vaatimuksiin. Jotkin teknologiat

on suunniteltu skaalautumaan vaivattomasti suunnilleen kaikille eri käytön tasoille, mutta toiset taas eivät skaalaudu järkevällä tavalla siirryttäessä suurempaan volyymiin, mikä voi aiheuttaa merkittäviä ongelmia ja jopa tilanteen jossa koko järjestelmä on migroitava uuteen rakenteeseen.

- **Teknologian levinneisyys:** Tietyt ohjelmistotekniikat saavuttavat teknologisen standardinomaisuuden, tai vähintään äärimmäisen laajan käyttötason, ohjelmistokehitystä tekevien tahojen parissa. Toiset taas eivät. Mitä laajemmin käytetty jokin ohjelmointikieli, moottori, viitekehys yms. on, sitä paremmin sen toiminta on yleensä dokumentoitu, testattu käytännössä eri skaaloissa, sitä enemmän sille löytyy valmiita kirjastoja ja sitä helpompaa on löytää ohjeita yleisten (ja harvinaisempienkin) ongelmien ratkomiseen joita kehitystyössä voi tulla vastaan. Tämä luo efektin jossa "suosio luo suosiota" ja ruokkii ennestään sitä että kyseistä teknologiaa päättävät käyttää muutkin alan toimijat.

- **Erikoistuneet käyttötarkoitukset:** Periaatteessa ohjelmointi on ohjelmointia ja tarpeeksi vääntämällä millä tahansa kielellä tai viitekehyksellä voi, ainakin teoriassa, tehdä mitä tahansa. Suurin osa näistä teknologioista on kuitenkin suunniteltu ja tarkoitettu tietynlaisten ongelmien ratkaisuun, jossa ne ovat juurikin oikeassa elementissään, mutta tämän kääntöpuolena on se että ne eivät ole kovinkaan käytännöllisiä johonkin toiseen tilanteeseen. Esim. vaikka onkin teoriassa mahdollista luoda verkkosivuja pelkällä C -kielellä, ei tätä kukaan kuitenkaan käytännössä edes halua yrittää, koska web-ohjelmointiin on olemassa paljon parempia työkaluja. Samoin jos halutaan tehdä signaalinprosessointia, sulautettuja järjestelmiä tai 3D -moottoreita niin niitä ei tehdä Javalla. Toteutusteknologiaa valitessa tuleekin perehtyä eri teknologioiden yleisiin käyttötarkoituksiin ja vahvuusalueisiin sekä rajoitteisiin, jotta valitun teknologian ominaisuudet tehokkaimmin mahdollistavat halutun toteutuksen rakentamisen.

Käytännössä on kuitenkin niin, että kaikki ohjelmistoyritykset erikoistuvat tietyntyyppisiin ohjelmistoihin ja sen seurauksena ylläpitävät tiettyjen tekniikoiden osaamista. Ohjelmistotalo joka tekee järjestelmien välisiä integraatioita tilauksesta ei koodaa pelejä, eikä verkkosivuja rakentava yritys toteuta sulautettujen järjestelmien laiteohjelmointeja, koska käyttötarkoitukset ja toteutukseen käytetyt teknologiat eroavat toisistaan valtavasti. Tästä syystä oikeassa

ohjelmistokehityksessä edellämainitut kysymykset toteutusteknologian valinnasta on pitkälti ratkaistu jo ennalta, joskaan ei täysin. Verkkosivujakin voi tehdä monella eri alustalla, viitekehyksellä ja kielellä, pelejä voidaan rakentaa erilaisilla moottoreilla ja kielillä ja niin edelleen. Hyvin usein on kuitenkin niin että jos asiakas haluaa toteutuksen tekniikalla X, eikä yritys jo valmiiksi tarjoa ratkaisuja tällä kyseisellä tekniikalla, asiakas etsii toisen toteuttajan, koska ohjelmistokehityksen luonteesta johtuen uusien teknologioiden ottaminen käyttöön on yritykselle melko suuri kuluerä, niin rahallisesti kuin ihmisresursseissa mitattuna.

3.5 Ohjelmiston elinkaaren suunnittelu

Ohjelmiston elinkaaren suunnittelu ennaltakäsin, eli se miten pitkään järjestelmää on tarkoitus käyttää ja miten sen käyttö aikanaan ajetaan alas ja toiminta siirretään hallitusti ilman käyttökatkoksia uuteen järjestelmään toteutetaan, on oleellinen osa nykyajan ohjelmistokehitystä. Paino sanalla nykyajan, koska menneisyydessä tätä piirrettä, kuten monia muitakaan tällä kurssilla käsiteltyjä asioita ei juurikaan huomioitu ohjelmistoja suunnitellessa, mikä onkin yksi syy siihen miksi käytössä on yhä lukuisia erittäin vanhoilla teknologioilla toteutteja kriittisiä järjestelmiä, joita ylläpidetään suurella rahalla ja vaivalla koska niiden käytöstä poistamista ei voida toteuttaa turvallisesti niin että palveluiden saatavuus migraatiovaiheessa uuteen systeemiin voitaisiin varmistaa. Erityisesti sellaisilla aloilla joissa tietotekniikka otettiin jo varhaisessa vaiheessa käyttöön, esimerkiksi pankkisektorilla ja rahoitusalalla, on yhä tänäkin päivänä päivittäisessä käytössä tietojärjestelmiä jotka on toteutettu esim. COBOLilla, joka on 65 vuotta vanha ohjelmointikieli.

Vaikka COBOL onkin alunperin suunniteltu juuri tällaisten järjestelmien tarpeisiin, on selvää että aika on ajanut sen ohi teknisesti, mutta koska sen varaan on rakennettu järjestelmiä jotka vastaavat maailmanlaajuisesti jopa 3 biljoonan dollarin rahansiirroista päivittäin, ei ole mitenkään yksinkertainen operaatio alkaa uudistamaan tällaista rakennelmaa. Tästä syystä COBOL -kehittäjät ovatkin nykyaikana varsin erikoislaatuisessa asemassa, jossa näiden osaajien vähäisestä määrästä johtuen heidän osaamisestaan suorastaan kilpaillaan työnantajien puolelta, vaikka käytännössä työ on pelkkää vanhojen järjestelmien ylläpitoa – ja migraatioiden suunnittelua uudemmilla teknologioilla toteutettuihin järjestelmiin.

Yleisesti ottaen ohjelmiston elinkaari voidaan nähdä joko lineaarisena tai syklisenä. Lineaarisessa mallissa ohjelmisto on tarkoitettu käytettäväksi yleensä johonkin tiettyyn tarkoitukseen tietyn aikaa, jonka jälkeen se hallitusti ajetaan alas. Käytännössä yleisempi nykyaikana on syklinen malli, jossa ohjelmistokehityksen sykli toistuu kunnes ohjelmistoa kehittävä taho tai asiakas jolle ohjelmisto on rakennettu kokee sen vanhentuneeksi tai tarpeettomaksi, missä vaiheessa käyttäjille ilmoitetaan elinkaaren päättymisestä[1] ja mahdollisesta siirtymisestä uuteen järjestelmään.

Ohjelmiston elinkaaren päättymiseen liittyy paljon kysymyksiä, joiden ratkaisut on parasta suunnitella etukäteen. Mitä tapahtuu järjestelmässä käsitellyille tiedoille? Otetaanko niistä varmuuskopioita? Miten järjestelmän alasajo konkreettisesti toteutetaan? Siirtyvätkö vanhat tiedot mahdolliseen uuteen järjestelmään? Koska nykyaikana ohjelmistoja ei juurikaan jaella fyysisiä kanavia pitkin, vaan kaikki lataamiset ja asennukset hoidetaan verkon kautta, ovat aiemmin järjestelmän alasajoon liittyneet kysymykset materiaalien hävittämisestä muuttuneet merkityksettömiksi. Osittain nämä samat kysymykset kuitenkin koskevat nykyaikanakin sitä mitä tapahtuu järjestelmän fyysisille laitteille, joskin nekin ovat hyvin usein tänä päivänä leasing-sopimuksilla hankittuja tai sijaitsevat jossain datakeskuksessa josta vain vuokrataan tilaa.

Teknisellä velalla tarkoitetaan ilmiötä, jossa ohjelmiston vanheneminen, päivitykset, huono suunnittelu ja puutteelliset vaatimusmäärittelyt luovat tilanteita, jotka pakottavat tekemään muita päivityksiä. Osittain tässä on siis kyse juurikin siitä mistä tässä luvussa on jo puhuttu – järjestelmien vanheneva teknologia pakottaa tekemään muutoksia. Osittain on kuitenkin kyse hivenen monimutkaisemmasta ilmiöstä, joka osittain on riippumaton teknologian luonnollisesta vanhenemisesta. Kuvitellaan tilanne jossa aletaan tehdä muutoksia ohjelmistoon – kun ohjelmiston yhtä osaa muutetaan, se usein "kevyesti vaatii"[2] jonkin toisenkin, useimmiten useidenkin muiden osien muutosta. Tämä ei ole "kova vaatimus"[3] siinä mielessä että järjestelmä kyllä toimii jollain tavalla ilmankin että näitä muita muutoksia tehdään, mutta ei optimaalisesti. On siis kertynyt *teknistä velkaa*, joka pitää maksaa pois jossain vaiheessa. Mitä enemmän tällaista tek-

[1] engl. End-of-life

[2] engl. soft requirement

[3] engl. hard requirement

nistä velkaa kertyy, sitä työläämpää on lähteä sitä “maksamaan”, eli toteuttamaan muutoksia jotka pitää tehdä jotta velasta päästään eroon. Toinen esimerkki teknisen velan kertymisestä on tilanne jossa vaatimusmäärittelyjä ei ole tehty aivan loppuun asti, mutta lähdetään silti jo työstämään ensimmäistä versiota ohjelmistosta jonkin syyn takia, ja vaatimusmäärittelyjen tarkentuminen sitten kerryttää teknistä velkaa kun toteutusta on ehditty jo tehdä johonkin suuntaan.

3.6 Aikataulutus, resurssointi ja budjetointi

Kun järjestelmää lähdetään kehittämään sovitaan jonkinlainen alustava aikataulu sille koska järjestelmä on valmis ja otettavissa käyttöön. Aikataulutuksessa tulee myös huomioida eri työtehtävien arvioitu aika, sekä se miten eri työtehtävät linkittyvät toisiinsa muodostaen ns. *kriittisen polun*. Kriittisellä polulla tarkoitetaan niiden työtehtävien linkittymistä toisiinsa, joiden on pakko tapahtua tietyssä järjestyksessä, koska seuraavan tehtävän aloittaminen on riippuvaista aiemman tehtävän suorittamisesta.

Joissain tapauksissa tämä aikataulu voi olla joustavampi ja joissain tiukempi, mutta yhtäkaikki oleellista on ymmärtää se että aikaa on aina rajallisesti ja että sillä ajalla joka kehitystyöhön käytetään on tietty hinta, joka muodostuu työntekijöiden palkoista ja muista työnantajalle koituvista kuluista, työtilojen vuokrista, työskentelyyn tarvittavien laitteiden hinnoista, ohjelmistolisensseistä ja muista kulueristä joita yrityksen pyörittämisestä koituu. Tämän päälle pitää tietysti huomioida myös se miten paljon yrityksen on tarkoitus tehdä voittoa. Ottaen huomioon kehittäjien ja muiden teknisten osaajien palkkatason nousevat kulut, ja siten myös numero projektin lopullisessa hintalapussa, hyvin vauhdikkaasti ylöspäin mitä enemmän työtunteja se käsittää. Tätä asiaa tullaan käsittelemään tämän kurssin harjoitustehtävissä, mutta on niiden lisäksi hyödyllistä, oman hahmotuskyvyn kehittämisen vuoksi, pyöritellä hieman näitä lukuja jotta ymmärtäisi sen yleisen skaalan missä liikutaan kun puhutaan ohjelmistoprojektien hinnoista.

Otetaan esimerkiksi tilanne, jossa IT-alan yrittäjä pyörittää pienimuotoista osakeyhtiötä. Esimerkin vuoksi voimme olettaa että yhtiöllä on neljä kehittäjää täyspäiväisesti töissä ja se on erikoistunut verkkosivujen rakentamiseen ja ylläpitämiseen. Jos oletamme että yhden kehit-

täjän tuntipalkka on 20 €, pelkkään kehittäjien palkkaan tällaisessa yrityksessä per kuukausi menee 146×20×4 = 11680 €. Kun tähän lisätään työnantajalle lankeavat pakolliset maksut[4], nousee summa n. 30 % reiluun 15 000 €. Jos esimerkiksi olettaisimme että tällaisen yrityksen toimiston vuokra olisi kuukaudessa n. 1000 euroa, ja vaikka jättäisimme tässä arviossa pois muut kulut kuten sähkö- vesi ja internet-liittymät ja laitteiston leasing -kulut, niin jo pelkästään näistä luvuista näemme jonkinlaista viitteellistä arviota siitä miten paljon rahaa pitää tulla sisälle taloon joka kuukausi että toiminta pysyy edes mahdollisena, saati sitten kannattavana. Tulee myös huomioida että tässä oletettu 20 € /tunti on aloittelevan kehittäjän palkka; senior developereille ja erikoisosaajille pitää maksaa merkittävästi suurempia korvauksia. Ja vasta kaikkien näiden kulujen jälkeen voi yrittäjä alkaa miettimään sitä minkälaista summaa itselleen maksaisi palkkana.

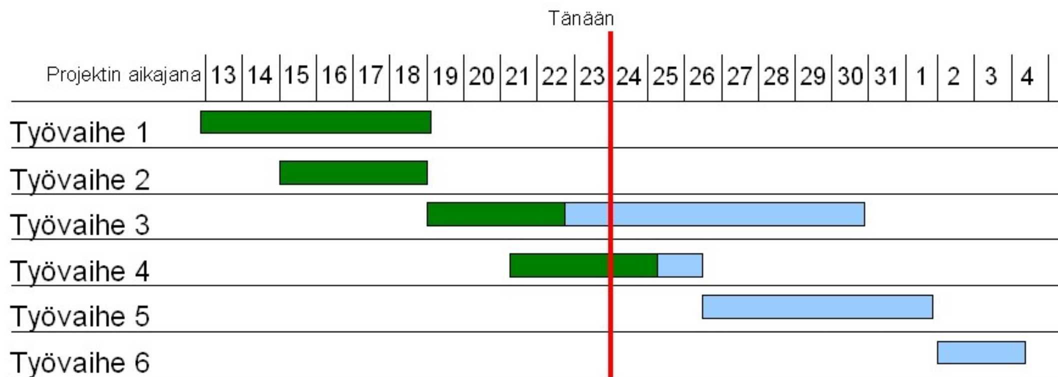
Jos tällainen yritys tekee sopimuksen projektista, jonka aika-arvio on 4 kuukautta ja jossa kaikki työntekijät tekevät töitä täysipäiväisesti (ts. heidän työaikaansa ei käytetä muihin projekteihin), niin tästä voidaan suoraan laskea että se vähimmäishinta joka tästä pitää pyytää on 4 × 16000 € = 64000 €, ja että tällä hinnalla yrittäjä ei vielä tee minkäänlaista voittoa koko hommasta. Jos halutaan päästä yrittäjän itsensä kannalta mitenkään järkeviin katteisiin nousee tämänkaltaisen projektin hintalappu vähintään 100 000 € tai yli.

Käytännössä näin ei tietenkään koskaan ole, vaan tällaisissa pienissä yrityksissä jokainen näistä neljästä kehittäjästä työstää jatkuvasti useita eri asiakasprojekteja, joskus yhteistyössä ja joskus yksinään, koska muuten tällaista toimintaa ei mitenkään päin saa kannattavaksi ilman että asiakkaan maksettavaksi tuleva summa kasvaa sietämättömäksi. Isommissa yrityksissä taas jokaiseen projektiin voidaan allokoida useita työntekijöitä, tai jopa tiimejä, koska lähtökohtaisesti myös projektien skaalat ovat huomattavasti suurempia.

Kuvassa 3.1 esitelty *Gantt-kaavio* on työtehtävien vaatiman ajan esittämiseen ja mittaamiseen tarkoitettu graafinen esitystapa, jossa skaala muodostuu käytettävästä aikayksiköstä, ja työtehtäviä kuvataan palkeilla jotka ovat jonkin tietyn aikajanan pituisia. Gantt-kaavio on melko vanha keksintö, mutta sitä käytetään yhä laajalti aikataulutuksen suunnittelussa. Kaavion heikkouksina pidetään kuitenkin sitä, että sen kuvaajat eivät ota huomioon työtehtävien aikaista ajan variaatiota, eivätkä muita työhön liittyviä ominaisuuksia kuin ajankäyttöä. Kaa-

[4] Työnantajamaksut 2024

vion yksinkertaisuus onkin tässä mielessä kaksiteräinen miekka, koska vaikka sen avulla on helppo hahmottaa sekä tekijöille että sidosryhmille projektin etenemistä, sillä ei voida kuvata kaikkia projektin kestoon vaikuttavia tekijöitä.



Kuva 3.1: Esimerkki Gantt-kaaviosta (Lähde: Wikipedia)

3.7 Valitse toteutustiimin jäsenet

Tiimin jäsenten valintaa mietittäessä tulee ottaa huomioon oman organisaation jo olemassa olevat voimavarat ts. mitä osaamista ohjelmistoa tekevältä taholta jo löytyy, sekä se miten toimitaan jos osaamista ei jo ole talossa. Jotta tätä prosessia olisi helpompi ymmärtää käydään ensin läpi yleisimmät ohjelmistokehitysorganisaation roolit.

- Ohjelmistokehittäjien osaamistasot:
  - **Traineet/Junior developerit:** Tällä nimikkeellä työskentelevät kehittäjät ovat uransa alussa, ja omaavat yleensä hyvin vähän, jos ollenkaan varsinaista työkokemusta. Lähes kaikissa organisaatioissa heidän työtehtävänsä mitoitetaan vastaamaan tätä asiantilaa, eikä junior developerilta yleensä vaadita läheskään samanlaista suoritustahtia kuin kokeneemmilta kehittäjiltä. Useimmiten myöskään junior developereille ei ainakaan ihan uran alkuvaiheessa sälytetä myöskään samanlaista itsenäistä vastuuta työstä kuin kokeneemmille kehittäjille, vaan yleensä työtä valvoo ja sen laatua tarkkailee läheisesti joku kokeneempi kehittäjä, esim. senior developer, joka voi tarvittaessa tarjota opastusta teknisten ongelmien ratkaisussa ja muutenkin toimia perehdyttäjän roolissa esim. organisaation työskentelykäytäntöjen suhteen.

- **Mid-level developerit:** Tätä termiä ei kovin usein käytetä, mutta sitä tarkoittavia "peruskoodareita" on paljon. Nämä ovat sellaisia kehittäjiä jotka omaavat yleensä muutamien vuosien kokemuksen työelämästä, ja ovat näinollen selvästi junior developereiden yläpuolella, mutta eivät toisaalta niin kokeneita että olisivat vielä siirtyneet senior developereiksi tai muihin vaativampiin rooleihin.
- **Senior developerit:** Tämän tason saavuttaneet kehittäjät ovat lähes aina vuosikausia tai vuosikymmeniä koodaamista tehneitä ammattilaisia, jotka ovat lunastaneet työnsä laadulla, nopeudella ja oma-aloitteisuudella paikkansa ohjelmistokehitysmaailmassa. Tästä syystä senior developereilla myös käytännössä aina on huomattavasti enemmän henkilökohtaista vastuuta työn tuloksesta kuin nuoremmilla kehittäjillä ja he toimivat jossain määrin enemmän esimiestehtävissä.
- **Superkoodarit:** Superkoodari ei varsinaisesti ole mikään työnimike, mutta sillä tarkoitetaan erittäin lahjakkaita ja tuotteliaita sovelluskehittäjiä, joiden työpanos vastaa useiden "tavallisten" kehittäjien tuottavuutta. Superkoodarit ovat käytännössä aina työnimikkeiltään vähintään senior developereita, ja omaavat vuosikymmenien hyvin laaja-alaisen kokemuksen ohjelmistokehityksen eri osa-alueista.

• Ohjelmistokehitystyön erikoistuneet roolit:

  - **Arkkitehdit:** Ohjelmistoarkkitehti on hahmo, joka määrittää ohjelmiston suuret linjat ohjelmistokehityksen alkuvaiheessa, ja visioi ja ohjaa sen laajennuksia myöhemmissä kehitysvaiheissa. Arkkitehdin ei välttämättä tarvitse olla kovatasoinen tekninen osaaja, mutta hyvin usein arkkitehdit ovat aiemmin senior developereina toimineita kehittäjiä joilla on pitkä kokemus ohjelmistokehityksestä. Yleensä varsinaista arkkitehdin nimikettä ei kohtaa pienemmissä ohjelmistotaloissa, vaan samoja tehtäviä tekevät esim. juurikin senior developerit.
  - **Front-end developer:** Ohjelmistokehittäjä joka on painottunut vahvasti ohjelmiston käyttöliittymäpuolen kehittämiseen ja siihen liittyviin teknologioihin.
  - **Back-end developer:** Kehittäjä joka on painottunut ensisijaisesti ohjelmiston taustalla pyörivän osuuden kehittämiseen.

- **Full-stack developer:** Kehittäjä joka on erikoistunut jonkinlaisen "teknologiapinon" (eli toisiinsa liittyvien front/back/tietokanta -teknologioiden) kokonaisuuden hallintaan, ja voi näinollen tehdä sekä front-end että back-end developerien työtä saumattomasti. Yleensä tällaiseen rooliin kehitytään keskittymällä ensin toiseen ja sitten toiseen aiemmin mainituista rooleista.
- **Erikoisosaajat:** Tämän kategorian alle menevät kaikki lukuisat ohjelmistokehityksen ja laajemmin IT-alan parissa työskentelevät henkilöt, jotka eivät varsinaisesti tee itse ohjelmistokehitystä, mutta joiden työpanos on välttämätön projektien ja organisaatioiden toiminnalle. Tähän kategoriaan kuuluvat serverien ylläpitäjät, yksikkötestaukseen erikoistuneet tekijät, käyttäjienhallinta ja niin edelleen. Pienemmissä organisaatioissa näitä tehtäviä hoitavat todennäköisesti kehitystyössä työskentelevät henkilöt muiden tehtäviensä ohella, mutta isommissa organisaatioissa ne ovat käytännössä aina omia työnimikkeittään.

Kun lähdetään miettimään sitä ketkä tekijät ohjelmistoprojektiin allokoidaan, tulee huomioida valittavissa olevien tekijöiden kokemustaso ja mahdollinen aiempi erikoistuminen johonkin ohjelmistokehityksen puoleen. Esimerkiksi mikäli projektiin laitetaan junior developer tai useampi, täytyy näille olla mieluusti joku kokeneempi kehittäjä mukana katselmoimassa työn sujumista. Mikäli projektissa on välttämätöntä käyttää jotain sellaista teknologiaa jonka osaajia ei valmiiksi löydy omasta tekijäpoolista, täytyy punnita sitä onko järkevämpää yrittää palkata uusia tekijöitä vai kouluttaa vanhoja uusiin työkaluihin.

Kokeneiden ohjelmistokehittäjien voidaan olettaa ottavan uusia kieliä ja viitekehyksiä haltuun hyvinkin pienellä vaivalla, kun taas junior developerin työajasta voi mennä kohtuuttoman suuri osa täysin uusien asioiden omaksumiseen. Tässä tulee kuitenkin huomioida sekin, että eri teknologiat ovat käsitteellisesti ja välillä myös ihan syntaktisella tasolla lähempänä ja kauempana toisiaan, jolloin niiden opettelua voi helpottaa se että tekijä osaa jo käyttää jotain samankaltaista systeemiä. Rautalangasta vääntäen, jos uusien teknologioiden opetteluun tarvitsee lähteä, niin jos tiimistä löytyy jo React- ja Python-osaajia, niin on järkevämpää laittaa Reactin osaaja opettelemaan Angularia ja Pythonin osaaja Djangoa jos jompikumpi näistä pitää ottaa haltuun.

Vaihtoehtona on tietysti myös sekin että lähdetään palkkaamaan täysin uutta henkilöstöä jolla jo on halutut taidot. Tätä asiaa voidaan lähestyä kahdesta näkökulmasta; joko palkataan ihan suoraan uusi työntekijä, tai ostetaan konsulttipalveluna väliaikaista työvoimaa jostain. Ensimmäisen prosessin ongelmana voi olla että sopivan tekijän löytämiseen menee rekrytointiprosesseineen aina tietty aika, jota ei välttämättä ole paljon käytettäväksi ja halutun osaamisen luonne voi rajoittaa saatavilla olevia mahdollisuuksia. Toisen vaihtoehdon ongelmana on että tällaiset tekijät ovat yleensä varsin kalliita ja vaikka konsulttipalveluita tarjoavat kehittäjät ovatkin yleensä varsin ammattitaitoisia, ei tämä välttämättä takaa sitä että täysin ulkopuolelta nopeasti tuotu tekijä soveltuisi työympäristöön tarpeeksi hyvin. Varsinaisen kehitystyön lisäksi myös joissakin muissa ohjelmistokehityksen osa-alueissa, esimerkiksi graafisessa suunnittelussa, voi olla järkevintä ostaa työpanos kokonaan joltain toiselta yritykseltä.

3.8 Rakenna kehitysympäristö

Kehitysympäristön rakentamisella tarkoitetaan hivenen eri asioita erityyppisten ohjelmistoprojektien ollessa kyseessä, mutta tietyt asiat kuten versionhallinnan käyttö ovat muuttuneet käytännössä standardeiksi, joita noudatetaan kaikilla ohjelmistoteollisuuden eri osa-alueilla. Käymme tässä alaluvussa läpi kehitysympäristön keskeisimpiä osa-alueita, eli versionhallintaa, CI/CD -putkia ja itse testiympäristöä, jossa ohjelmistoa kokeillaan ennen kuin se voidaan siirtää tuotantoon.

3.8.1 Versionhallinta

Versionhallintajärjestelmät ovat yksi nykyaikaisen ohjelmistokehityksen kulmakivistä, joka mahdollistaa sekä koodin muutosten seuraamisen koherentisti, niiden perumisen tarpeen vaatiessa, koodin saatavuuden varmistamisen repositorioiden kautta että usean kehittäjän työn koordinoimisen järkevästi ja tarpeen vaatiessa eristämisen erillisiksi kehityslinjoiksi, joita voidaan sitten sulauttaa toisiinsa. Käytännössä versionhallinnasta puhuttaessa tarkoitetaan lähes aina spesifisti Git -ohjelmistoa (ja siihen kytkeytyviä GitHub ja GitLab -ympäristöjä), joka

on ylivoimaisesti suosituin ja käytetyin versionhallintatyökalu, mutta todellisuudessa muitakin tämänkaltaisia järjestelmiä on paljon[5].

Versionhallintatyökalujen todella tehokas käyttäminen niin että niistä saa kaiken mahdollisen hyödyn irti vaatii jonkin verran perehtymistä, mutta oleellista ohjelmistoprojektin koordinoimisen ja toteuttamisen kannalta on se että kaikki tekijät ymmärtävät ainakin käytetyn järjestelmän perusteet. Monimutkaisempia versionhallinnan operaatioita varten tiimissä kannattaa olla kuitenkin ainakin yksi henkilö joka on opetellut käytössä olevan versionhallintajärjestelmän kehittyneempien ominaisuuksien käyttöä ja voi tarpeen tullen ratkoa mahdollisesti nousevia ongelmia, esim. merge -konflikteja.

Yleisesti ottaen versionhallinnan ottaminen käyttöön on triviaalia. Joissakin erikoisemmissa käyttötapauksissa, esimerkiksi työskenneltäessä isojen pelimoottorien kanssa, repositorioiden konfigurointi on kuitenkin hivenen monimutkaisempaa ja vaatii oman opettelunsa.

3.8.2 CI/CD -putket

Continuous Integration/Continuous Deployment -putkilla tarkoitetaan ohjelmistokehityksen suunnittelua siten, että valmiin koodin testaaminen, integroiminen ja siirtäminen tuotantoon on mahdollisimman suoraviivainen prosessi, minkä vuoksi tällaista järjestelmää kutsutaankin putkeksi[6]. Tällaiseen putkeen kuuluu oleellisesti myös kaikkien mahdollisten osa-alueiden automaatio siten, että ihmisen tekemää työtä missään julkaisuprosessin osassa tarvitaan mahdollisimman vähän. Isot versionhallintapalveluntarjoajat kuten GitLab ja GitHub tarjoavat omat CI/CD -työkalunsa, joilla nämä prosessit voidaan helposti integroida varsinaisen versionhallinnan kanssa yhdeksi kokonaisuudeksi, jossa kaikkein pisimmälle automatisoituna järjestelmä testaa uudet ominaisuudet koodissa ja sitten automaattisesti vie ne tuotantovaiheeseen, ilman että ihmisten täytyy edes varsinaisesti tarkkailla prosessia.[7]

CI/CD -käsiteparin ensimmäinen osa, Continuous Integration, pitää sisällään sen osan prosessista jossa muutokset lähdekoodiin testataan ja integroidaan osaksi repositoriota automaattisesti. Toisen osan, eli Continuous Deliveryn, nimi vaihtuu joissakin yhteyksissä Continuous

[5] Version control systems

[6] engl. pipeline

[7] What is CI/CD?

Deploymentiksi, mikä on yksityiskohta johon tulee kiinnittää huomiota. Kysymys on nimittäin siitä, että nämä kaksi lähes samankuuloista käsitettä eivät tarkoita aivan samaa asiaa, vaan lähtökohtaisesti kun puhutaan Continuous Deliverystä tarkoitetaan sellaista putkistoa jossa ennen tuotantoon viemistä jonkun vastuuhenkilön pitää vielä kuitata muutokset, kun taas Continuous Deploymentissa putkiston toiminta on täysin automatisoitu, tavalla jota kuvattiin edellisessä kappaleessa.[8]

3.8.3 Testiympäristö

Jotta voitaisiin varmistua siitä että kaikki toimii käytännössä kuten sen pitää on yleensä käytäntönä rakentaa testiympäristö, joka vastaa ominaisuuksiltaan tuotantoympäristöä täydellisesti. Tarkoituksena on siis luoda ympäristö, joka on konfiguroitu niin että kun kehitettävää ohjelmistoa ajetaan siinä, se antaa vasteen joka on täysin samanlainen kuin se mitä tuotannossa halutaan.

Käytännössä tämä tarkoittaa sitä että tietokannat, palvelimet, pilviympäristö ja muut muuttuvat tekijät on määritetty niin että ne ovat täydellisiä replikoita siitä mitä tullaan käyttämään tuotannossa. Yleistä on myös se että varsinaista kehitystä tehdessä itse työ tapahtuu testiympäristössä jonkinlaisen etäyhteyden kautta, siten että koodiin tehtyjen muutosten toiminta on mahdollisimman nopeasti nähtävissä. Vasta sitten kun jokin uusi ominaisuus toimii testiympäristössä juuri kuten sen pitää toimia tuotannossakin se viedään eteenpäin tuotantoon.

3.9 Tiimityön pohjustus

Kun uutta projektia aloitetaan ja tiimin jäsenet on lyöty lukkoon, on tarpeen pitää kokous jossa käydään projektin yleisilme läpi niin, että kaikki tekijät ovat samalla kartalla siitä mihin ollaan pyrkimässä ja mitkä vastuut kenellekin kuuluvat. Eksakteja työtehtäviä ei tietenkään vielä allokoida, koska ne tulevat Scrumin tai jonkin muun työnohjausmetodin myötä omassa prosessissaan sitä mukaa kun työ etenee, mutta tiimin roolitus lyödään lukkoon.

Tässä vaiheessa myös päätetään yhteisistä työskentelymetodeista, jos niitä ei ole jo organisaatiotasolla määritetty, ja käydään läpi mitä teknologioita projektissa tarvitaan. Optimaa-

[8] CI/CD: The what, why, and how

lisessa tilanteessa hyvin tehdyn pohjustustyön seurauksena tiimi operoi kuin hyvin viritetty koneisto, jossa jokainen tietää oman toimialueensa ja asiat soljuvat eteenpäin kitkattomasti ja omalla painollaan.

3.10 Korkean tason komponentit ja rajapinnat

Nykyaikana on hyvin vähän sellaisia ohjelmistoja, jotka toimisivat täysin yksinään omalla tontillaan. Melkein kaikki järjestelmät ovat jonkinlaisessa yhteydessä johonkin toiseen järjestelmään, pääasiassa siten että tietoa siirretään joko jatkuvasti tai periodisesti kahden tai useamman eri järjestelmän välillä. Yleensä tämän kautta joko haetaan jotain tietoja toisesta järjestelmästä, tai lähetetään tietoja toiseen järjestelmään joka tarjoaa palveluja joita oma järjestelmä ei voi toteuttaa, joiden parametreiksi tiedot siis lähetetään. Hyvin yleinen esimerkki tällaisesta ovat tekstiviestejä lähettävät järjestelmät; tekstiviestien lähettäminen verkon kautta vaatii tiettyjä teknisiä ominaisuuksia, joiden kehittäminen itse on mahdollista, mutta verrattain työlästä. Tästä syystä johtuen on yleistä käyttää palveluntarjoajia, jotka tarjoavat maksullisen APIn jonka kautta voi omasta järjestelmästä lähettää viestin sisällön toiseen järjestelmään, joka hoitaa itse viestin lähettämisen vastaanottajalle. Periaatteessa kun puhutaan jonkin ison yrityksen, vaikkapa vakuutusyhtiöiden tai pankkien tietojärjestelmästä niin tarkoitetaan oikeasti järjestelmien joukkoa, jossa on useita toisistaan erillisiä tietojärjestelmiä, jotka ovat jatkuvassa vuorovaikutuksessa keskenään ja yhdessä muodostavat yhden ison järjestelmän.

Jotta tämä olisi mahdollista ja organisoitua, täytyy näiden järjestelmien välille määritellä rajapintoja[9], jotka kuvaavat sitä miten järjestelmä A voi pyytää tietoja järjestelmältä B, ilman että sen tarvitsee tietää rajapintaa lukuunottamatta mitään B:n toiminnasta. Järjestelmän B täytyy siis tarjota API[10], joka määrittää miten A voi muotoilla pyyntöjä sellaisella tavalla että B osaa toimittaa halutun datan ja päinvastoin. Tämä mahdollistaa sen että täysin eri organisaatiot tai tiimit voivat olla vastuussa näiden kahden eri tietojärjestelmän toiminnasta tietämättä mitään siitä miten toinen järjestelmä on toteutettu, kunhan vain rajapinnat on määritelty siten että tiedonsiirto on mahdollista.

Tästä johtuen käytännössä kaikki järjestelmät nykyaikana tarjoavat tällaisen APIn, jonka

[9] engl. interface

[10] engl. Application Programming Interface

käyttö on yleensä hyvin suoraviivaista ja yksinkertaista. Rajapintoja määrittäessä täytyy kuitenkin ymmärtää mihin tietoihin rajapinnan käyttäjillä saa olla pääsy ja miten. Esim. hyvin äskettäin eräs iso suomalainen ohjelmistotalo, joka ylläpitää erästä hyvin laajassa käytössä olevaa opetuksen tukena olevaa järjestelmää, antoi pääsyn järjestelmän rajapintaan kolmannelle osapuolelle, joka kehitti omaa sovellustaan alkuperäisen järjestelmän rinnalle. Tästä seurasi se, että järjestelmässä käsiteltäviä henkilötietoja päätyi vahingossa kolmannen osapuolen käyttäjien tietoon.[11]

Ohjelmiston APIn suunnittelu tulee aloittaa jo varhaisessa vaiheessa kehitystyötä, ja siinä tulee huomioida käytettävyyden ohella tiedon saatavuus ja saatavuuden rajoittaminen, sekä tietysti muut tietoturvaan ja yksityisyyteen mahdollisesti liittyvät tekijät. Oman APIn suunnittelun lisäksi on elintärkeää kartoittaa mitä dataa tai palveluja ohjelmisto tarvitsee muilta järjestelmiltä, miten niiden APIt toimivat ja suunnitella miten näiden avulla kehitetään sellaiset komponentit jotka hakevat tai lähettävät tietoa halutulla tavalla.

3.11 Olemassa olevan koodin hyödyntäminen

Nykyaikana ohjelmistokehitys nojautuu hyvin vahvasti, ehkä jopa liikaa, jonkun toisen ohjelmistokehitystä tekevän tahon koodaamien kirjastojen ja moduulien käyttämiseen ja oman tiimin aiemmin rakentamien palikoiden uudelleenkäyttöön. Työtahdin kannalta tämä on tietysti posiitiivinen kehityssuunta, koska se mahdollistaa sen että sellaiset asiat joita varten aiemmin tarvitsi käyttää päiviä tai viikkoja koodaustyötä ovat saatavilla valmiina paketteina, useimmiten jopa ilmaiseksi. Kun puhutaan kolmansien osapuolien luomista ratkaisusta tulee kuitenkin huomioida se, että näiden osien kehitys ja päivittäminen on usein oman tiimin ulottumattomissa. Jos kysymys on jonkun kolmannen osapuolen ilmaiseksi kehittämistä tai ylläpitämistä järjestelmistä ei myöskään ole käytännössä mitään takeita sille että näitä komponentteja ei voitaisi muuttaa joksikin täysin toiseksi tai niiden ylläpitoa yksinkertaisesti lopettaa ilman mitään ilmoitusta. Kaupallisten komponenttien kohdalla tämä ei ole lähellekään niin yleistä. Suurimmaksi osaksi tämä ei ole ongelma, mutta se voi muodostua sellaiseksi joissakin tapauksissa.

Ohjelmistoprojektia aloittaessa tuleekin kartoittaa se, missä määrin omasta takaa löytyy

[11] Wilmasta vuoti oppilaiden tietoja

sellaista koodia joka olisi uudelleenkäytettävissä uudessa projektissa, ja pidempään ohjelmistokehitystä tehneillä tiimeillä onkin yleensä varsin laajat kirjastot joista voidaan ammentaa usein toistuvien toimintojen valmiita toteutuksia, tai ainakin aihioita jotka ovat nopeasti mukautettavissa uuteen käyttötarkoitukseen. Osittain tämä johtuu siitä että ohjelmistotalot useimmiten keskittyvät jonkin tietyntyyppisten ohjelmien tuottamiseen, esim. verkkosivuihin tai peleihin tai “integraatiopalikoihin” tai tuotannonvalvontajärjestelmiin, jolloin toisiaan seuraavien projektien perusrakenne pysyy yleensä jossain määrin samanlaisena. Osittain taas siitä, että jotkut ohjelmointikielet lähtökohtaisesti kannustavat rakentamaan mahdollisimman uudelleenkäytettäviä ja yleisiä rakenteita, joita voidaan soveltaa moniin erilaisiin ongelmanratkaisuihin.

Mikäli valmiita toteutuksia ei löydy omien aiempien projektien tiimoilta, on todennäköisyys sille että joku toinen on jo luonut tällaisen toteutuksen hyvin korkea. Tämä ei tarkoita sitä että kaikkiin ongelmiin on olemassa jo valmiiksi koodattu ratkaisu jonka voi nappia painamalla ottaa käyttöön omassa ohjelmistossa, mutta varsinkin yleisesti käytetyille kielille valmiita kirjastoja löytyy todella laaja-alaisesti.

4 Ohjelmistojärjestelmän kehitys

Tässä osiossa keskitymme otsikon mukaisesti itse kehitysprosessiin, pyrkien jälleen ylläpitämään edes viitteellisen kronologisen järjestyksen siinä miten asiat on esitetty.

4.1 Toiminnalliset ja ei-toiminnalliset vaatimukset

Ohjelmiston vaatimukset voidaan jakaa kahteen selkeästi toisistaan eroavaan osioon, jotka ovat toiminnalliset ja ei-toiminnalliset vaatimukset. Hyvin yksinkertaistetusti asian voisi ilmaista niin, että toiminnalliset vaatimukset määrittävät sen **mitä** ohjelmiston pitää tehdä ja ei-toiminnalliset vaatimukset määrittävät sen **miten** ohjelmisto tekee sen.

Näin ollen voisikin sanoa että toiminnalliset vaatimukset ovat paljon tärkeämpiä, koska niiden määrittämät asiat ovat välttämättömiä sille että ohjelmisto ylipäänsä toimii. Jos näitä vaatimuksia ei täytetä, ohjelmisto ei lähtökohtaisesti tee sitä mitä se on suunniteltu tekemään. Jos taas ei-toiminnallisia vaatimuksia ei täytetä, ohjelmisto kyllä tekee periaatteessa silti sen mihin se on tarkoitettu, mutta se ei tee sitä tavalla joka olisi esteettisesti, käytettävyydeltään tai muuten kokemuksellisesti haluttu. Koska ohjelmistokehitys suurimmaksi osaksi nykyaikana ei kuitenkaan tapahdu akateemisissa norsunluutorneissa joissa käytettävyydellä tai ulkoasulla ei ole mitään väliä, vaan hyvin pitkälti maksavien asiakkaiden tarpeisiin vastaten, on ei-toiminnallistenkin vaatimusten merkitys kasvanut huomattavasti. Nykyaikainen kuluttaja haluaa ohjelman joka ei pelkästään toimi hyvin, vaan on myös esteettisesti miellyttävä ja intuitiivinen käyttää. Tulee myös huomioida, että ei-toiminnallisiin vaatimuksiin katsotaan yleisesti ottaen kuuluvan erilaiset tietoturvaan ja käyttäjien yksityisyyteen liittyvät näkökohdat ohjelman toiminnassa.

Toiminnalliset vaatimukset on mahdollista ja usein järkevintä kuvata erilaisina *jos* $\implies$ *niin*

-skenaarioina, joita sanotaan käyttötapauskaavioiksi[1]. Näillä kaavioilla kuvataan ohjelmiston toiminnot mahdollisimman selkeästi ja korkealla tasolla. Niiden ei ole tarkoitus olla lopullisia pohjapiirrustuksia siitä miten ohjelmisto varsinaisesti rakennetaan, vaan ohjenuoria sille mitä eri toimintojen suorittamisesta seuraa.

Ei-toiminnallisten vaatimusten määrittelyyn ei ole olemassa aivan yhtä selkeitä ohjenuoria eikä työkaluja. Työkaluja tähänkin työhön on, mutta ne ovat, kuten ei-toiminnalliset vaatimuksetkin, huomattavasti avoimempia tulkinnalle ja toisaalta huomattavasti yleisempiä ja yleistettävämpiä erilaisiin ohjelmistoihin. Esimerkiksi ohjelmiston käytettävyyden suunnittelussa on jossain määrin vakiintunut Nielsenin heuristiikkojen käyttö[2], jotka määritteli jo 1994 käytettävyystutkimuksen pioneeri Jakob Nielsen[3]. Nämä heuristiikat koostuvat "kymmenestä käskystä" jotka Nielsen muotoili tutkimuksensa pohjalta, joiden on tarkoitus toimia peukalosääntöinä käytettävyysmuotoilulle. Täytyy kuitenkin ymmärtää että ei-toiminnalliset vaatimukset ovat osittain myös hyvin subjektiivisia, ja tehtäessä juuri tietylle käyttäjäkunnalle tai asiakkaalle räätälöityä ohjelmistoa asiakkaan henkilökohtaiset erikoistarpeet voivat olla tärkeämpiä kuin esim. edellämainittujen suunnitteluperiaatteiden noudattaminen.

Vaikka voikin tuntua hivenen epäintuitiiviselta, niin yleisesti ottaen ei-toiminnallisiin vaatimuksiin luokitellaan myös sellaisia asioita kuten tietoturva ja käyttäjien yksityisyyden varmistaminen. Nykyajan maailmassa nämä ovat tietysti ensiarvoisen tärkeitä näkökohtia, joiden noudattamatta jättäminen voi pahimmassa tapauksessa tulla kalliiksi niin järjestelmän tilanneille tahoille kuin sen toimittajillekin.

4.2 Ymmärrä asiakkaan tarve yleisellä tasolla

Ennen kuin mitään varsinaista suunnittelua tai kehitystyötä voidaan tehdä, tulee ymmärtää asiakkaan tarve ohjelmiston toiminnalle perinpohjaisesti, eli toisinsanoen pitää tehdä vaatimusmäärittely jossa toiminnalliset ja ei-toiminnaliset vaatimukset ohjelmiston toiminnalle määritetään selkeästi. Tämä vaatii käytännössä pitkällistä kommunikaatioprosessia asiakkaan kanssa, jossa käydään läpi sekä yleisellä tasolla se mitä asiakas haluaisi ohjelmiston tekevän, että

[1] Use-case Diagram

[2] Nielsenin heuristiikat

[3] Jacob Nielsen

yksityiskohtaisemmalla tasolla se miten ohjelmiston tulisi tämä asia tehdä. Hyvin usein ohjelmistoprojekteissa, silloinkin kun on kyse isoista asiakkaista ja isoista toimittajista, näitä asioita ei tarvittavalla tarkkuudella käsitellä, mikä johtaa lopputuloksiin jotka eivät ole asiakkaan näkökulmasta haluttuja.

Erittäin korkean profiilin esimerkki tällaisesta on joitakin vuosia sitten toteutettu Apotti -hanke, jossa Uudenmaan sairaanhoitopiiri hankki erittäin kalliin tietojärjestelmän, joka ei kuitenkaan lopulta täyttänyt niitä tarpeita joita asiakkaalla oli. Syynä tähän oli ensisijaisesti se (jos ei huomioida Apotin toteutukseen liittyviä suoranaisia teknisiä ongelmia), että järjestelmän toimittanut Epic Software on erikoistunut rakentamaan tietojärjestelmiä joita operoidaan Yhdysvaltojen pääosin yksityisomisteisessa sairaalaekosysteemissä, joka on kulu- ja laskutuslogiikaltaan hyvin erilainen ympäristö kuin Suomen julkinen sairaalajärjestelmä. Tätä eroavaisuutta ei osattu järjestelmän suunnitteluvaiheessa kommunikoida tarpeeksi, minkä vuoksi Apotissa on raportoitu olevan paljon sellaisia ominaisuuksia jotka ovat suoranaisen hyödyttömiä, tai jopa työntekoa hidastavia, koska niille ei yksinkertaisesti ole käyttöä.

Ohjelmiston tuottajan tulee muistaa myös se, että asiakas ei aina täysin tiedä tai ymmärrä mitä oikeastaan haluaa. Asiakkaalla on tietysti jonkinlainen yleistason idea siitä mitä halutaan, esim. verkkosivu, mobiiliapplikaatio asiaan X, tietojärjestelmä tuotteiden seurantaan tjsp, mutta tämä idea ei välttämättä ole kovinkaan refinoitunut siinä vaiheesa kun asiakas ja järjestelmän toimittaja aloittavat neuvottelut. Riippuen asiakasorganisaation aiemmasta kokemuksesta järjestelmien hankkimisessa ja omasta teknisestä osaamistasosta visio siitä mitä halutaan voi olla hyvinkin tarkka – mutta se voi myös olla näiden asioiden puutteesta johtuen hyvin sumea. Toisaalta voi olla niin että asiakas ei täysin ymmärrä mikä on mahdollista, toisaalta taas ei hahmoteta sitä miten paljon mikäkin haluttu ominaisuus tulee maksamaan, ja toisaalta ei välttämättä edes osata ajatella sitä mitä sen halutun järjestelmän tarkalleen ottaen pitäisi tehdä.

Näistä syistä johtuen onkin ensiarvoisen tärkeää, että ohjelmiston toimittaja osaa neuvottelutilanteessa kysyä oikeita kysymyksiä, ja pyytää selvennyksiä sellaisiin yksityiskohtiin jotka vaativat selventämistä. Jos toimittaja tyytyy asiakkaan ensimmäiseen vastaukseen siitä mitä halutaan, pyrkimättä selvittämään eksaktia tarvetta ja haluttua lopputulosta, päädytään helposti juurikin sellaiseen tilanteeseen jossa asiakas saa jotain vähän sinnepäin olevaa kuin mitä

alunperin ajatteli. Tämä johtaa epätyytyväiseen asiakkaaseen ja joko toimittajan vaihtamiseen tai mahdollisesti suureen määrään täysin turhaa työtä kun järjestelmää muokataan vastaamaan paremmin sitä todellista tarvetta – jotka ovat molemmat varsin epätoivottuja lopputuloksia.

Joskus kaikesta asiakkaan tarpeiden kartoituksesta huolimatta päädytään kuitenkin tilanteeseen, jossa asiakas ei eksaktisti osaa sanoa mitä haluaa. Tällaisessa tilanteessa ohjelmiston toimittajan pitääkin osata ehdottaa erilaisia toteutustapoja joiden pohjalta asiakas muodostaa lopullisen mielipiteensä, koska mikäli tällaisia avoimia kysymyksiä ei ratkota ja ohjelmistokehittäjät itse päättävät miten asiat ratkaistaan päädytään hyvin herkästi tilanteeseen jossa asiakas ei kuitenkaan ole tyytyväinen lopputulokseen.

Tietysti, kun puhutaan nykyaikaisesta ketterästä kehityksestä ei ole tarkoituskaan lyödä täysin lukkoon kaikkia asioita heti alussa ja sitten lähteä toteuttamaan niitä, koska kysymys on iteratiivisesta prosessista. Mitä tässä haetaan takaa on ns. co-design -ajattelu[21], eli että tämä kommunikaatio siitä mitä asiakas oikeastaan haluaa ja mitä ohjelmiston tulee tehdä ei saa myöskään loppua siihen että ollaan sovittu alustavasti siitä mitä ollaan toimittamassa, vaan mahdollisuuksien mukaan asiakkaan edustajia tulee osallistaa kehitysprosessiin esittelemällä heille uusia järjestelmän ominaisuuksia ja pyytämällä mielipiteitä siitä vastaavatko nämä asiakkaan tarpeita.

4.3 Järjestelmän yleisrakenteen ymmärtäminen

Kun lähdetään kuvaamaan järjestelmän arkkitehtuuria, tulee ymmärtää projektin skaalan vaikutus kuvausprosessiin. Jos kysymys on yhdestä nettisivustosta, ei ole välttämättä mielekästä alkaa piirtelemään yksityiskohtaisia UML -kaavioita kaikista sen toiminnoista. Toisaalta, jos puhutaan useiden kymmenien tai satojen tuhansien samanaikaisten käyttäjien systeemistä on hyvin yksityiskohtainen mallintaminen äärimmäisen tärkeää.

Arkkitehtuurityylit ovat tapoja kategorisoida erilaisia ohjelmistokehityksen arkkitehtuureja. Tällaisia malleja on useita, eikä voida varsinaisesti sanoa, että yksikään niistä olisi varsinaisesti standardinomainen, vaan että oikea kategoria valikoituu ohjelmistoprojektille pitkälti sen mukaan mitä ollaan tekemässä. Perinteisin näistä on *Monoliittinen Arkkitehtuuri*[4], jossa

[4] engl. Monolithic Architecture

ohjelmistokokonaisuus nähdään yhtenä suurena yksikkönä. Hivenen käänteisesti se nimestään huolimatta on parhaimmillaan pienien ja keskisuurten ohjelmistojen kuvaamisessa, koska suurempiin projekteihin se skaalautuu jossain määrin huonosti. Isompia ohjelmistoja suunnitellessa nykypäivänä käytetään esimerkiksi *Palvelukeskeistä Arkkitehtuuria*[5], jossa ohjelmisto rakentuu toisistaan irrallisten, rajapintojen kautta toistensa kanssa kommunikoivien yksiköiden varaan. Tästä vieläkin pidemmälle kehittynyt malli on *Mikropalveluarkkitehtuuri*, jossa ohjelmiston hajautus erillisiin yksiköihin on viety vielä paljon yksityiskohtaisemmalle tasolle, mikä mahdollistaa mallin soveltamisen erittäin hyvin nopeasti kehittyviin ohjelmistoihin. Näitä kahta mallia ei kuitenkaan tule sekoittaa *Komponenttikeskeiseen Arkkitehtuuriin*, joka on samankaltainen, mutta kuitenkin jossain määrin erilainen lähestymistapa ohjelmiston hajauttamiseen. Erona komponenttien ja mikropalvelujen välillä voitaisiin kenties nähdä se, että komponentit perinteisesti ymmärretään toimivan ohjelmiston osina, mutta ei irrallaan siitä, kun taas mikropalvelu joka on osa ohjelmistoa voi olla esimerkiksi kokonaan varsinaisen ohjelmiston ulkopuolinen ohjelma. Esimerkkinä tällaisesta voitaisiin käyttää esim. Twilio -palvelua, jonka kautta voidaan helposti toteuttaa tekstiviestien lähettämistä verkon kautta ja joka voidaan helposti ottaa käyttöön sen tarjoaman APIn kautta. Kaikenkaikkiaan erilaisia arkkitehtuurimalleja on tässä esiteltyjen lisäksi vielä paljon muitakin, mutta emme käsittele niitä tämän kurssin puitteissa.

4.4 Järjestelmän mallintaminen (UML)

Järjestelmän mallintaminen, eli toisinsanoen sen toimintojen ja osien kuvaaminen tarpeeksi suurella tarkkuudella on erittäin oleellinen osa ohjelmistokehitystyötä. Vaikka ketterässä kehityksessä onkin pyritty pois tästä toimintamallista ja suosimaan enemmän sitä että mallintaminen tapahtuisi jo toteutettujen osien esittämisestä UMLllä (tai vastaavilla systeemeillä), on totuus kuitenkin se että varsinkin yhtään isompia systeemejä rakentaessa mallintamisella on hyvin selkeä tarkoitus ohjelmiston eri osa-alueiden suhteiden hahmottamisessa.

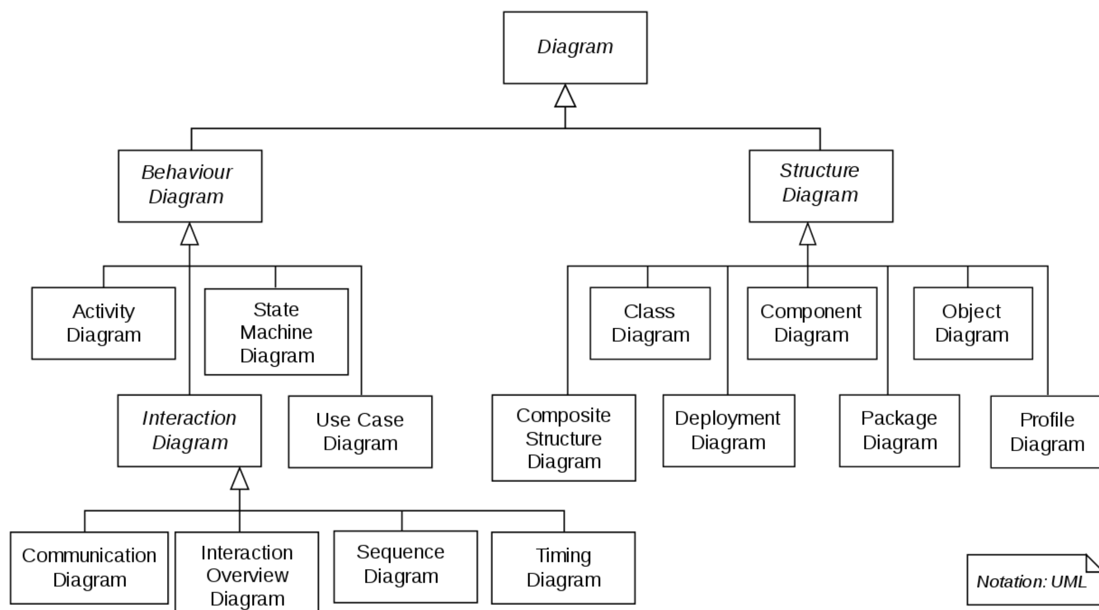
Unified Modelling Language (UML) on ohjelmiston toimintojen suunnitteluun tarkoitettu “mallinnuskieli”. On ehkä hivenen epäintuitiivista puhua kielestä, koska UML on oikeastaan vain joukko sääntöjä, jotka määrittävät sen miten ohjelmiston toimintaa kuvaavan graafisen

[5] engl. Service-Oriented Architecture

kaavion eri osat pitää kuvata ja miten ne yhdistyvät toisiinsa. Kyse ei siis ole "kielestä" siinä mielessä kuin puhuttaessa konekielistä, hakukielistä tai ylipäänsä mistään sanallisesti ilmaistusta kielestä. UMLn kielimäinen luonne on kuitenkin ehkä selvimmin nähtävissä siinä, että oikeaoppisesti tehdystä UML -kaaviosta voi nykyaikaisilla kehitystyökaluilla generoida suoraan esim. olio-ohjelmoinnin mukaisen luokkahierarkian, ja taas päinvastoin koodista voidaan oikeilla työkaluilla generoida suoraan UML -kaavioita, mitä nykyaikana käytetäänkin usein ketterien kehityskäytäntöjen yhteydessä.

Vaikka UML onkin alunperin kehitetty juurikin olio-ohjelmoinnin apuvälineeksi, sillä on yleisesti ottaen ollut voimakas vaikutus muihinkin ohjelmistokehityksen kehitysparadigmoihin ja prosesseihin. UMLää sinällään, sellaisessa puhtaassa muodossa jossa se on standardoitu vuodesta 1997 lähtien, ei kuitenkaan juurikaan nykyaikana käytetä, vaan käytännön ohjelmistokehityksessä käytetyt kaaviot ovat vapaamuotoisempia. Näissä yleisemmin käytetyissä vapaamuotoisemmissa kaavioissa on kuitenkin merkittävästi vaikutteita UMLstä. Puhtaan UMLn käyttöä rajoittaa myös, sen lisäksi että sille harvoin on välttämätöntä tarvetta, sekin että se on jossain määrin sidoksissa olio-ohjelmointiin paradigmana ja olio-ohjelmointi taas on jossain määrin menettänyt suosiotaan ohjelmistokehityksen parissa.

Varsinaisista UML -kaavioista puhuttaessa tarkoitetaan useaa erilaista kaaviotyyppiä, jotka on kuvattu seuraavassa kaaviossa:



Kuva 4.1: UML-kaavioiden eri tyypit.

Kuten ylläolevasta kaavioiden kaaviosta[6] voimme nähdä, UMLssä on kaksi pääasiallista kaaviotyyppiä, toimintakaaviot (engl. Behaviour Diagram) ja rakennekaaviot (engl. Structure Diagram). Näiden kahden päätyypin alle järjestäytyy useita erilaisia alatyyppejä ja niiden alatyyppejä, jotka on tarkoitettu tarjoamaan erilaisia näkökulmia ohjelmiston tarkasteluun. Rakennekaaviot ja erityisesti luokkakaaviot (engl. Class Diagram) ovat se puoli UMLstä jota suurin osa ihmisistä ajattelee, jos he koskaan ajattelevat UMLää. Nimenomaisesti luokkakaavio kuvaa ohjelmiston luokkahierarkiaa ja luokkien keskeisiä ominaisuuksia, ja on muunnettavissa suoraan koodiksi tietyillä työkaluilla. UML kokonaisuutena pitää sisällään kuitenkin huomattavasti enemmänkin erilaisia kaaviotyyppejä ja periaatteessa sitä voidaan käyttää kuvaamaan mitä tahansa ohjelmiston toimintaan tai rakenteeseen liittyvää asiaa.

UML:n lyhyt historia

Unified Modelling Languagea eli UMLää ei kehitetty tyhjästä, vaan se muodostui aiempien vastaavankaltaisten tekniikoiden ideoita ja metodeja yhdistelemällä. 1990 -luvun alkupuolella, olio-ohjelmoinnin kultakautena, kehittyi useita erilaisia järjestelmiä jotka oli tarkoitettu ohjelmiston ja erityisesti oliopohjaisten ratkaisujen rakenteiden suunnitteluun. Näistä järjestelmistä Grady Boochin kehittämä Booch Method, James Rumbaughin Object-modelling Technique (OMT) ja Ivar Jacobsonin Object-oriented Software Engineering (OOSE) olivat merkittävimmät vaikutteet UMLn kehittämisessä. Booch, Rumbaugh ja Jacobson työskentelivät 1990-luvun puolivälissä Rational Software -yhtiössä, jossa heidän yhteistyönsä seurauksena näiden jokaisen tahoillaan kehittämien järjestelmien perustuksille rakennettiin UML vuosien 1994-96 aikana, josta on sittemmin tullut samankaltainen de facto -standardi ohjelmistokehityksen suunnittelussa kuin Git -ohjelmistosta on tullut versionhallinnassa. Vuonna 1997 UMLn kehitys siirtyi Object Modelling Groupin (OMG) -alaisuuteen ja sen ensimmäinen standardoitu versio julkaistiin.

Teoriaboksi: Muut mallinnuskielet UML ei ole ainoa ohjelmistotuotantoa koskettava mallinnuskieli. Business Process Modelling Notation (BPMN) ja tietovuo(engl. flowchart).

Teoriaboksi: Model-driven engineering (MDE) low-code, no-code. Tasot: konfigurointi, graafiset sovelluskehittimet, koodipohjainen ohjelmointi

[6] UML diagram types

4.5 Suunnittele järjestelmän tietoturva, tietosuoja ja käyttöturvallisuus sekä eettiset aspektit

Käyttöturvallisuus, tietoturva ja eettisyys eivät ole asioita, joita voi jälkikäteen tarran lailla liimata ohjelmistotuotteen päälle, vaan ne on rakennettava järjestelmään sisälle. Ohjelmistoalan ammattilaisen haasteena onkin pitää nämä asiat jatkuvasti mielessä, vaikka ne eivät välttämättä vaikuta hänen päivittäiseen työntekoonsa. Lisäksi kyseessä on monipuoliset ja monimutkaiset asiakokonaisuudet, joissa useimmiten tukeudutaan erityisasiantuntijoiden apuun. Nämä erityisasiantuntijat saattavat olla organisaatiossa useammalla projektilla työskenteleviä spesialisteja tai ulkopuolisia konsultteja. Monesti näissä asioissa myös hankitaan ulkopuolinen auditointi, ns. 'second opinion', johon myös regulaatio toisinaan pakottaa (ks. edellinen luku).

On erityisen tärkeää huomata, ettei kyseessä ole pelkästään tekninen tai matemaattinen ongelma, vaan kehittäjän ja kehittäjätiimin tulee ymmärtää, miten käyttäjät toimivat, missä ympäristössä (ohjelmisto)tuotetta tullaan käyttämään ja mitkä sidosryhmät ovat tuotteen vaikutuspiirissä. On myös hyvä huomata, että pääosin tietoturvaan liittyvät kysymykset, kuten yksityisyys ja datan integriteetti, ovat eettisesti latautuneita kysymyksiä, mutta eettisesti latautuneita kysymyksiä voi projektissa esiintyä tietoturvakysymysten ulkopuolellakin. Tavoitteena on teknisin ja sosio-teknisin toimenpitein saada järjestelmä tukemaan haluttua ja 'oikeaa' toimintatapaa.

Käyttöturvallisuuden huomioiminen on keskeistä ohjelmistoprosessin kaikissa vaiheissa. Tämä sisältää käyttöliittymäsuunnittelun, virheidenhallinnan, dokumentaation ja ohjeiden luomisen, sekä käyttäjätestauksen. Käyttöturvallisuuden tavoitteena on varmistaa, että ohjelmisto on helppokäyttöinen, että käyttäjät ymmärtävät sen toiminnot ja että se toimii odotetulla tavalla.

Käyttöturvallisuus liittyy myös käyttäjien tietoturvaan ja yksityisyyteen. Tämä tarkoittaa esimerkiksi käyttäjätietojen suojaamista, salasanojen turvallisuutta ja käyttäjän suostumuksen varmistamista sekä luo käyttäjille turvallisen ja luotettavan kokemuksen ohjelmiston käytössä. Lisäksi käyttäjille on tärkeää tarjota asianmukaista koulutusta ja tukea ohjelmiston käyttöön liittyvissä turvallisuuskysymyksissä, jotta he ymmärtäisivät ohjelmiston turvallisuusominaisuudet ja mahdolliset riskit.

Tietoturvan huolellinen suunnittelu on useimmille ohjelmistojärjestelmälle elintärkeää ja se

kattaa toimenpiteet ja käytännöt järjestelmän ja sen sisältämän tietojen suojaamiseksi. Tavoitteena on varmistaa tuotettujen palveluiden ja niiden sisältämien tietojen eheys, luottamuksellisuus ja saatavuus. Tietoturvaan liittyy useita näkökohtia, kuten tietojen salaaminen, käyttöoikeuksien hallinta, haavoittuvuuksien tunnistaminen ja korjaaminen, sekä jatkuvan seurannan ja valvonnan toteuttaminen.

Ohjelmistotuotantoprosessin aikana tietoturvan huomioiminen tapahtuu useissa vaiheissa. Vaatimusmäärittelyssä tulee ottaa huomioon tietoturvavaatimukset ja riskianalyysi. Suunnittelussa tulee suunnitella tietoturvaratkaisut, kuten vahva autentikointi ja pääsynhallinta. Toteutuksessa on huolehdittava turvallisesta ohjelmointikäytännöstä ja haavoittuvuuksien minimoimisesta. Testauksessa on varmistettava, että ohjelmisto kestää tietoturvatarkastelun, ja ylläpidossa on seurattava ja päivitettävä tietoturvaan liittyviä asioita.

Sopiva määrä tietoturvaa riippuu useista tekijöistä, kuten ohjelmiston luonteesta, sen käyttötarkoituksesta, käyttäjien tarpeista ja sidosryhmien odotuksista. Tietoturva ei ole yksi absoluuttinen taso, vaan se on jatkuvaa tasapainottelua riskien ja resurssien välillä. Yleisesti ottaen pyrkimys on saavuttaa riittävä tietoturvan taso, joka suojaa ohjelmistoa merkittäviltä uhilta ja riskitilanteilta. Sopiva määrä tietoturvaa pyrkii minimoimaan riskit hyväksyttävälle tasolle ottaen huomioon käytettävissä olevat resurssit ja aikataulut. Tietoturvan taso voi myös vaihdella eri ohjelmistoissa ja konteksteissa. Esimerkiksi kriittisten infrastruktuurijärjestelmien tai henkilötietoja käsittelevien sovellusten tietoturvavaatimukset voivat olla erittäin korkeat, kun taas vähemmän arkaluonteisiin sovelluksiin voidaan asettaa vähemmän tiukat vaatimukset.

Tietoturvan tärkeys korostuu entisestään, kun otetaan huomioon käyttäjien toiminta, ympäristö, jossa ohjelmistoa käytetään, ja eri sidosryhmät, jotka ovat tuotteen vaikutuspiirissä. On myös tärkeää tiedostaa, että tietoturvaan liittyvät kysymykset, kuten yksityisyys ja datan integriteetti, ovat eettisesti merkittäviä. Tavoitteena on kehittää järjestelmä, joka teknisten ja sosiaalisten toimenpiteiden avulla tukee haluttua ja oikeaksi koettua toimintatapaa tietoturvallisesti.

Ohjelmistotuote on eettinen jos ja vain jos se on rakennettu eettisesti kestävälle pohjalle. Tällöin ohjelmistotuotteen suunnitteluratkaisut, kuten tuotteen datanhallinta, tietoturva, käyttäjäkokemussuunnittelu sekä monetisaatiomallit ovat suunniteltu jo hankkeen alussa siten, että ne kestävät eettistä tarkastelua. (Heimo, Kimppa & Nurminen, 2014; Heimo, 2018) Esi-

merkiksi mobiilipeli, jonka suunnittelun pohjana on käytetty monetisaatiomallia, jossa tuote on suunnattu lapsille ja jossa käytetään psykologisia koukutusmekanismeja ja mikromaksuja, ei voi olla eettisesti järin kestävä ratkaisu. (Heimo et al. 2018) Lisäksi eettisen suunnittelun ja analysoinnin tulee jatkua koko ohjelmistoprojektin ajan.

Ohjelmistotuotteen tulee heijastaa yhteiskunnallisten sekä kehittäjiensä eettisten arvojen lisäksi asiakkaansa sekä toimialansa eettisiä arvoja. Terveydenhuollon tarpeisiin kehitettävien ohjelmistojen kehittäjien olisi esimerkiksi hyvä tutustua lääketieteelliseen etiikkaan[7].

Lakiboksi: GDPR EU:n yleinen tietosuoja-asetus (GDPR) on asetus, joka säätelee henkilötietojen käsittelyä Euroopan unionissa. Ohjelmistotekniikan näkökulmasta se edellyttää, että ohjelmistoissa, jotka käsittelevät henkilötietoja, on toteutettava asianmukaiset suojatoimenpiteet, kuten pseudonymisointi, salaus, ja oltava kykeneviä osoittamaan näiden toimenpiteiden noudattaminen. Tämä koskee niin ohjelmiston suunnittelua, kehitystä kuin ylläpitoakin.

GDPR:n soveltaminen tarkoittaa ohjelmistojen suunnittelua ja kehittämistä “privacy by design” -periaatteen mukaisesti. Tämä tarkoittaa, että henkilötietojen suoja on sisällytettävä ohjelmistoon jo suunnitteluvaiheessa, ei jälkikäteen. Lisäksi tulee varmistaa, että asiakkaat voivat hallinnoida omia tietojaan, mukaan lukien oikeus tietojen poistamiseen ja siirtämiseen.

Linkki: GDPR

Kts. myös: TÄMÄ LINKKI EI JOHDA OIKEAAN PAIKKAAN

Ota nämä huomioon jo prosessin alussa ja pidä mielessä koko prosessin ajan!

4.6 Suunnittele järjestelmän käytettävyys ja käyttötilanteet

Käytettävyys[8] tarkoittaa ohjelmistosuunnittelussa ohjelmiston kykyä täyttää käyttäjän tarpeet onnistuneesti, tehokkaasti ja tyydyttävästi. Ohjelmistojärjestelmän käytettävyyden suunnittelu liittyy siihen, kuinka helppoa ja tehokasta järjestelmän käyttö on loppukäyttäjälle. Tässä kontekstissa, helppous tarkoittaa sitä, kuinka intuitiivisesti käyttäjä voi navigoida ohjelmiston läpi ja saavuttaa halutut tavoitteet. Tehokkuus taas viittaa siihen, kuinka nopeasti ja virheettö-

[7]kts. esim. Gillon `https://www.ht.lu.se/media/utbildning/dokument/kurser/FPRB01/20132/gillon.pdf`

[8]engl. usability

mästi käyttäjä voi suorittaa tehtävänsä. International Organization for Standardization (ISO) määrittelee käytettävyyden viiden keskeisen komponentin kautta: oppimisen helppous, tehokkuus, muistettavuus, virheiden määrä ja tyytyväisyys.[9] Nykyaikana kun ohjelmistotuotanto on hyvin voimakkaasti painottunut tuottamaan palveluita ja tuotteita tavallisten kuluttajien arkipäiväisiin tarpeisiin, olivat ne sitten taloudellisia, viihteellisiä tai harrastuksellisia, on käytettävyyden merkitys noussut huomattavasti tärkeämpään asemaan kuin 2000-luvun alussa tai ennen sitä, kun ohjelmistoja tehtiin ja käytettiin ensijaisesti hyvin spesifeissä työympäristöissä ja usein ohjelmistotekniikkaan perehtyneiden ihmisten toimesta. Tästä syystä nykyaikana käytettävyyden, käyttöliittymien ja käyttäjäkokemuksen suunnitteluun käytetäänkin huomattavasti enemmän resursseja ja käytännössä kaikilla isommilla ja monilla pienemmilläkin ohjelmistotuotannon toimijoilla työskentelee UX/UI (User Experience/User Interface) -suunnittelijoita jotka ovat erikoistuneet nimenomaisesti tähän ohjelmistokehityksen osa-alueeseen.

Välillä nykyäänkin tietotekniikan ja tietojenkäsittelytieteen opiskelijoiden parissa kohtaa haitallisia ennakkoluuloja siitä että tämä ohjelmistotekniikan haara olisi jotenkin hyödytön ja vähemmän tärkeä, mutta on kuitenkin kiistaton fakta että monet oman aikamme merkittävimmistä ohjelmistoteollisuuden menestystarinoista on rakennettu hyvin vahvasti intuitiivisen käytettävyyden ja mielyttävän käyttökokemuksen varaan, mistä selkeimpänä yksittäisenä esimerkkinä voidaan pitää Applen tuotteiden ekosysteemiä ja yleisemmin graafisten käyttöliittymien käytännössä standardinomaista asemaa. Mikäli eläisimme yhä tilanteessa jossa tietokoneita käytettäisiin pääasiassa komentoriviltä ei tietotekniikka olisi koskaan saavuttanut sellaista yhteiskunnallista läpimurtoa jonka olemme nähneet viimeisten kahden vuosikymmenen aikana, ensin graafisten käyttöliittymien ja hiirellä tapahtuvan navigoinnin ja myöhemmin kosketusnäyttöjen muodostuttua normeiksi. Tästä syystä käytettävyysasioita ei tule sivuuttaa ohjelmistotuotannossa, vaan niihin tulee suhtautua samalla vakavuudella kuin näennäisesti teknisempiin osa-alueisiin kuten varsinaiseen ohjelmointiin ja järjestelmäsuunnitteluun. Turun yliopistossa käytettävyyssuunnitteluun voi erikoistua tietojenkäsittelytieteen vuorovaikutusmuotoilun maisterilinjalla, jonka tarjoamilla kursseilla voi perehtyä syvällisemmin tähän aiheeseen ja sen erilaisiin tekniikoihin ja teorioihin.

[9] ISO 9241-11:2018

1. Oppimisen helppous: Kuinka intuitiivista ja helppoa on uusien käyttäjien oppia käyttämään ohjelmistoa? Mitä vähemmän aikaa ja vaivaa vaaditaan järjestelmän toiminnan oppimiseen, sitä parempi sen käytettävyys on.
2. Tehokkuus: Kun käyttäjä on oppinut ohjelmiston käyttämisen, kuinka nopeasti ja vaivattomasti hän pystyy suorittamaan halutut tehtävät? Tehokkuus voidaan mitata esimerkiksi toimintojen suorittamiseen kuluneen ajan tai vaadittujen toimintojen määrän avulla.
3. Muistettavuus: Kuinka helposti käyttäjät voivat palata ohjelmiston pariin pitkän poissaolon jälkeen ja jatkaa sen käyttöä ilman uuden oppimisprosessin tarvetta?
4. Virheiden määrä: Kuinka moni käyttäjä tekee virheitä ohjelmiston käytön aikana, kuinka vakavia nämä virheet ovat, ja kuinka helposti he voivat toipua näistä virheistä?
5. Tyytyväisyys: Onko ohjelmiston käyttö miellyttävää, ja ovatko käyttäjät tyytyväisiä ohjelmiston toimintaan ja ulkoasuun?

Käyttötilanteiden suunnittelussa pyritään määrittelemään, kuinka ohjelmistoa tullaan käyttämään reaalimaailman tilanteissa. Tämä prosessi sisältää eri käyttäjäroolien määrittelyn, heidän tehtävänsä, tavoitteensa, ja tarpeensa. Nämä “käyttäjätarinat” auttavat muovaamaan ohjelmiston rakennetta ja toiminnallisuutta niin, että ne palvelevat käyttäjän tavoitteita mahdollisimman hyvin.

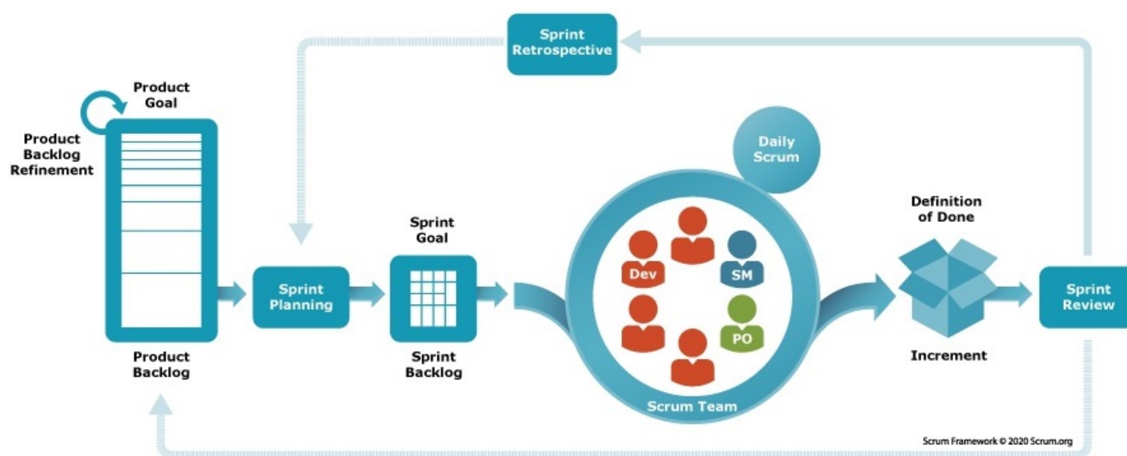
On hyvä huomata, että käytetävyys on riippuvaista käyttäjästä ja käyttötilanteesta: käytettävyys ei ole kaikille yksilöille tai ryhmille sama. Käytettävyyssuunnittelussa kohderyhmän tuntemus ja ymmärrys ovatkin keskeisessä roolissa. On tärkeää muistaa, että sekä käytettävyyden että käyttötilanteiden suunnittelu ovat jatkuvia prosesseja ohjelmistokehityksessä. Käyttäjäpalaute ja testaus ovat olennaisia työkaluja näiden suunnitteluprosessien parantamiseksi, ja niitä tulisi hyödyntää säännöllisesti koko ohjelmiston elinkaaren ajan. Käytettävyystutkimuksen parissa onkin muodostunut joitakin metodologioita, esimerkiksi Co-Design Principle[21], jotka on tarkoitettu parantamaan ymmärrystä ohjelmiston loppukäyttäjän näkemyksistä ja osallistamaan käyttäjiä itse suunnitteluprosessiin jotta heidän tarpeensa ja mielipiteensä asiasta tulisivat tarpeeksi hyvin kuulluiksi.

4.7 Kehitystyön koordinointi ja johtaminen

Nykyaikaisen ketterän ohjelmistokehityksen onnistumiseksi tehokkaimmin on ohjelmistotuotannossa kehittynyt ja osittain omaksuttu muilta teollisuuden aloilta erilaisia tuotannonohjauskäytäntöjä ja järjestelmiä, joiden avulla prosessin sujuvuus ja tehokkuus pystytään optimoimaan. Tässä luvussa käymme näistä läpi kolme hyvin yleisesti käytössä olevaa ja toisiinsa tietyillä tavoilla linkittyvää metodia, jotka ovat Scrum, Kanban ja Lean. Kaikki näistä metodologioista edelsivät ketterän kehitysparadigman muodostumista jossain määrin, mutta ovat sittemmin muodostuneet keskeisiksi työkaluiksi sen toteuttamisessa.

4.7.1 Scrum

Ohjelmistokehitysprosessin tehokas koordinointi on ensiarvoisen tärkeää työn etenemisen kannalta. Tähänkin asiaan on kehitetty erilaisia teoreettisia viitekehyksiä ja niitä tukevia käsitteellisiä työkaluja, joista varmaankin yleisimmin käytetty nykyaikana on Scrum. Se on työnjohtamisen malli, joka yksinkertaisesti ilmaistuna on sitä että kokoustetaan asioista ja päätetään mitä tehdään. Ja sitten se tehdään. Ja sitten kokoustetaan lisää. Scrumin viitekehyksessä tätä kokous-toteutus-kokous-toteutus -sykliä sanotaan sprintiksi. Sprintin pituus voi olla mitä tahansa päivästä kuukauteen ja Scrum tekniikkana onkin kehitetty tukemaan ketterän kehityksen hektistä ja joustavaa työtahtia. Tulee kuitenkin huomioida että Sprintin pituus ei perinteisesti koskaan ole enempää kuin kuukausi.



Kuva 4.2: Scrum framework (Lähde: scrum.org)

Scrumiin liittyy paljon omaa jargoniaan, joka on tarpeen avata tässä jotta aihetta voi ollenkaan käsitellä[10]:

- **Scrum:**

  Käsitteellä Scrum tarkoitetaan koko viitekehyksen lisäksi varsinaisia sprinttien välisiä kokouksia, joissa säädetään Sprint Goalit ja joiden jälkeen alkaa Sprint.

- **Sprint:**

  Scrumin toteutusjaksoa sanotaan "sprintiksi", eli pyrähdykseksi. Sprintti voi olla periaatteessa minkä vain pituinen, mutta Scrumin luonteen vuoksi yleensä sprintti on ennemmin lyhyt kuin pitkä.

- **Sprint Goal:**

  Sprint Goal on, kuten nimestä voi ehkä päätellä, sprintin päämäärä eli "maali". Tämä tarkoittaa siis sitä kehitystavoitetta joka on asetettu edellisessä Scrumissa, esim. jonkin uuden ominaisuuden implementointia.

- **Daily Scrum:**

  Daily Scrumit ovat päivittäisiä lyhyitä palavereja joissa käydään nopeasti läpi viimeisen vuorokauden aikana saavutetut asiat.

- **Product Backlog:**

  Product Backlog on lista työtehtävistä jotka pitää toteuttaa. Product Owner on vastuussa tämän listan ylläpitämisestä.

Scrum-tiimit jakautuvat perinteisesti kolmeen pääasialliseen rooliin:

- **Developer:**

  Tätä roolia ei ehkä tarvitse selittää enempää, koska ohjelmistokehittäjien työskentelyä on jo käsitelty aiemmin tässä dokumentissa.

[10]Scrumin käsitteet ovat saaneet vaikutteita rugbyn käsitteistä jossa sillä kuvataan peliä edeltävää tilannetta jossa joukkueen pelaajat kokoontuvat painamaan päänsä yhteen ja tekemään nopeaa taktikointia.

- **Scrum Master:**

  Scrumissa on työnjohtajan tai tiiminjohtajan rooli lähtökohtaisesti jaettu kahtia Scrum Masterin ja Product Ownerin välille. Näistä Scrum Master on vastuussa tiimin organisoimisesta, siitä että Scrumin sprintit pysyvät aikataulussa ja muutenkin ns. aktiivisesta työn johtamisesta.

- **Product Owner:**

  Product Owner on Scrum -jargonia sellaiselle henkilölle, jonka vastuulla on projektin onnistuminen ja työtehtävien määrittäminen backlogiin. Voisikin sanoa että Product Owner on Scrum Masteria enemmän taustalla toimiva hahmo, jonka tehtävänä on valvoa koko projektin onnistumista ja pitää huolta siitä että backlogiin on kirjattuna tärkeitä työtehtäviä siten että kehittäjät ja Scrum Master voivat niitä sieltä poimia. Product Ownerina usein toimii henkilö jolla on itsellään teknistä osaamista, mutta tämä ei ole välttämätöntä, vaan lähinnä Product Ownerilta vaaditaan abstraktia ymmärrystä kokonaiskuvasta ja siitä miten juuri tämän tiimin "tuote", toisinsanoen se ohjelmiston osa jota tiimi työstää, integroituu laajempaan ohjelmistoprojektiin.

Planning Poker on Scrumin avuksi kehitetty pelillistetty työkalu, jossa pokerin ideaa on sovellettu projektin työtehtävien suunnitteluun. Pelissä siis pelaajat käyttävät kortteja, jotka esittävät erilaisia ohjelmistokehityksen konsepteja ja työtehtäviä, ja relatiivisia arvoja niihin tarvittavalle ajalle. Pelin korttien numerointi noudattaa Fibbonaccin kaavaa, eikä sitä ole tarkoitettu kuvaamaan eksaktia työmäärää esim. päivissä tai viikoissa, vaan eri työtehtävien työmäärän vertautuvuutta toisten työtehtävien työmäärään. Tämän kaiken pointtina on että kun jokainen pelaa lyö "pimeänä" oman esityksensä pöytään, ilman että muut osallistujat voivat etukäteen tietää mitä kukin pelaaja on valinnut, tämä ei vaikuta toisten ajatuksiin siitä miten kannattaisi toimia jossain yhteydessä. Pelissä ei myöskään voi kukaan voittaa, vaan tarkoitus on saavuttaa konsensus siitä miten tulisi toimia seuraavan sprintin aikana.

Tämän tekniikan kehitti alunperin James Grenning vuonna 2002 ja sen popularisoi ohjelmistokehityksen parissa Mike Cohn joitakin vuosia myöhemmin. Koska tämän idean toimintaa voi olla hivenen hankala ymmärtää vapaan selityksen pohjalta, on alapuolelta luettavissa miten Planning Poker jakautuu eri pelin vaiheisiin.

- Moderaattorina toimiva henkilö, joka ei osallistu peliin, toimii puheenjohtajana.
- Tiimin Product Owner esittää jonkinlaisen käyttötapauksen jonka toteutusta pitäisi arvioida. Tiimin muut jäsenet voivat tässä vaiheessa kysyä selventäviä kysymyksiä ja keskustella keskenään aiheesta, mutta he eivät tässä vaiheessa saa vertailla omia arvioitaan seuraavan vaiheen resurssien suuruuksista. Moderaattorina toimiva henkilö voi tehdä näistä keskusteluista tiivistelmän.
- Jokainen pelaajista valitsee kortin jonka arvo kuvastaa hänen arviotaan käytettävien resurssien tarpeesta ja asettaa sen kuvapuoli alaspäin pöydälle eteensä. Arvo voi tässä tapauksessa tarkoittaa esim. vaadittujen työpäivien määrää. Pelaajat eivät tässäkään vaiheessa saa kommunikoida korttiensa arvoja toisilleen.
- Jokainen paljastaa korttinsa samanaikaisesti.
- Ensin omaa arviotaan saavat esitellä ne joiden arvio oli korkein ja matalin, perustellakseen arviotaan. Tämän jälkeen muut saavat puhua vapaasti.
- Keskustelua eri vaihtoehdoista jatketaan kunnes konsensus asiasta on saavutettu. Moderaattori voi myös pyrkiä neuvottelemaan konsensuksen pelaajien kesken mikäli sellaista ei muuten synny.
- Jotta keskustelu pysyisi jonkinlaisessa koheesiossa eivätkä kierrokset venyisi liian pitkiksi, voi Moderaattori tai Product Owner ottaa koska tahansa käyttöönsä ajastimen, jonka tullessa täyteen vuoro loppuu ja pelataan uusi kierros pokeria.

Kuten ylläolevasta kuvauksesta voi päätellä, Planning Poker voi olla hivenen haastava ottaa tehokkaasti käyttöön. Sitä onkin kritisoitu jonkin verran, erityisesti siitä että peli on rakennettu relatiivisten aika-arvioiden varaan, mutta ihmismieli ei ole kovin hyvä arvioimaan asioita relatiivisesti, vaan pelaajat usein alkavat miettiä pelin numeerisia arvioita päivissä ja tunneissa, jotka ovat konkreettisia ajanmääreitä. Toinen asia josta peliä on kritisoitu on sääntöjen vaikeus ja monitulkintaisuus, mikä tekee varsinkin uusille käyttäjille siitä vaikean omaksua.[11] Planning

[11] Why I stopped using Planning Poker

Pokeria on myös kritisoitu siitä että se voi viedä kohtuuttoman paljon aikaa, erityisesti jos konsensusta ei saada muodostettua käsiteltävistä aiheista, jos jotkut tiimin jäsenet eivät pidä sitä järkevänä työskentelymetodina tai jos projektin skaala on liian suuri.[12] Näistä ongelmistaan huolimatta Planning Pokeria käytetään varsin laajasti Scrumin apuvälineenä.

4.7.2 Lean

Toinen oleellinen tällainen systeemi on ns. Lean -ajattelu, joka on autovalmistaja Toyotan kehittämä. Leanissa ideana on toiminnan tulosten jatkuva mittaaminen, näiden mittaustulosten tuominen käytännön työnjohtotyöhön ja "hukan", eli negatiivista tulosta tuottavien toimintatapojen karsiminen pois organisaatiosta. Metodologian nimi tuleekin pitkälti siitä että siinä "trimmataan" turhuutta pois tuotantoprosesseista, jolloin tehokkuus itsestään paranee. Suurimpana "hukkana" Lean -ajattelussa nähdään se, että ihmisten tuotantopotentiaalia ei käytetä mahdollisimman tehokkaasti, joskin järjestelmä tunnistaa muitakin hukkaanheitetyn potentiaalin muotoja.

Ohjelmistokehityksen näkökulmasta Leanissä voidaan tunnistaa hukaksi muunmuassa ohjelmiston hitaan toiminnan, kommunikaatio-ongelmat, turhien varmuuskopioiden pitämisen, innovaatioiden ja uusien ideoiden ohittamisen ja sen että kehittäjät eivät tee sitä missä ovat tehokkaimpia. Näiden hukkaelementtien hallintaan ja poistamiseen käytetään Leanissä metodologiaa jota kutsutaan arvovirtakuvaukseksi[13], jossa nimensä mukaisesti esitetään tuotantoprosessin vaiheet ja materiaalit visuaalisena diagrammina, josta on helppo nähdä tuotannon yksilölliset osa-alueet ja niiden lopputulokset ja sen pohjalta tehdä päätöksiä siitä mistä kohtaa prosessia pitäisi trimmata tehokkaammaksi.

Hukan lisäksi kaksi muuta keskeistä käsitettä Lean -ajattelun ymmärtämiseksi ovat virtaus ja "epätasaisuus"[14], jotka ovat toistensa vastakohtia. Virtaus on ideaali, johon tulee pyrkiä; se että tuotantoprosessin vaiheet soljuvat häiriöttömästi eteenpäin, ja epätasaisuus taas ovat ne asiat jotka häiritsevät virtausta. Epätasaisuus ei kuitenkaan ole sama asia kuin hukka, vaan se koostuu pikemminkin tuotannon vaiheista jotka voivat aiheuttaa pullonkauloja virtauk-

[12] Planning Poker Common Challenges and Solutions

[13] Value-stream Mapping

[14] jap. mura

sen etenemiselle, kun taas hukka tarkalleen ajateltuna ymmärretään asioiksi jotka heikentävät virtauksen voimakkuutta. Jos keskitytään pelkästään hukkaan, voi käydä niin että prosessin epätasaisuus aiheuttaa muita ongelmia. Ohjelmistokehityksen näkökulmasta esimerkkinä tästä voitaisiin käyttää sitä, että tiimi kehittää jonkin ohjelmiston tai sen osa-alueen käyttäen sellaista ohjelmointikieltä joka on heille mahdollisimman tuttu ja hyvin ymmärretty, jolloin hukan määrä on olematon, mutta jolla ei ole tarpeeksi hyvää API -standardia jonka kautta yhteistyökumppanit ja asiakkaat voisivat tehokkaasti ottaa sen käyttöön, jolloin epätasaisuuden määrä on valtava.

4.7.3 Kanban

1940-luvulla keksitty kanban[15], joka myös alunperin keksittiin Toyotan autotehtaiden tarpeisiin, on järjestelmä, joka alunperin kehittyi Leanin osaksi, mutta jota nykyään käytetään myös Scrumin kanssa. Kanban on aikoinaan kehitetty supermarkettien asiakaslogiikan pohjalle: Asiakas saa ainoastaan sitä mitä on tarjolla, ja kauppiaan tehtävä on huolehtia siitä että tuotteita tulee lisää hyllyihin sitä mukaa kun niitä ostetaan ja että ne eivät loju varastotilassa tarpeettomasti. Ideana on siis inventaarion managerointi siten, että varastossa ei ikinä ole turhaan mitään, mutta toisaalta kaikkea on tarpeeksi saatavilla kun sitä tarvitaan. Kun tämä prosessi käännetään työelämän ohjausmenetelmäksi, se tarkoittaa käytännössä sitä että työn määrää hallinoidaan siten että työtehtäviä ei pinoudu odottamaan "varastoon", vaan sitä mukaan kun tekijät suorittavat tehtäviä "kaupan hyllyltä" uusia tehtäviä siirretään "hyllyille" poimittaviksi.

Kun kanbania sovelletaan ohjelmistokehitykseen käytännössä hyllystä otettavat "tavarat" ovat siis erilaisia kehitysprosessin työtehtäviä, ja "hylly" itsessään jonkinlainen tiketointijärjestelmä johon tehtäviä laitetaan esille. Yleensä puhutaan kanban -tauluista[16], joka on joko fyysinen tai digitaalinen taulu johon merkitään kanban-korteilla työtehtäviä ja resursseja. Tarkkaavainen lukija saattaa havaita tässä vaiheessa tietynlaista yhtäläisyyttä Scrumin toimintaprosesseihin, ja kanbania käytetäänkin tästä syystä paljon Scrumin aputyökaluna. Käytännössä Product Owner päivittää kanban -taululle työtehtäviä, joita sitten Scrum Master jakaa kehittäjille.

[15] suom. visuaalinen signaali

[16] What is a kanban board?

4.8 Suunnitteluvirheet ja kuviot

Design smells on käsite jolla tarkoitetaan vihjeitä, jotka viittaavat siihen että järjestelmän suunnittelussa ja toteutuksessa on tapahtunut jotain lähtökohtaisesti virheellistä. Tämä ei välttämättä tarkoita sitä että järjestelmä ei toimisi, mutta se tarkoittaa aina sitä että järjestelmä toimii jossain määrin epäoptimaalisesti ja että jossain kohtaa on joko vedetty mutkia suoriksi väärällä tavalla tai vähintäänkin toteutustiimin välinen kommunikaatio ei ole aina pelannut vaadittavalla tasolla. Pahimmassa tapauksessa design smellsit aiheuttavat jossain kohtaa järjestelmän elinkaarta merkittäviä ongelmia kun alkaa ilmaantumaan erilaisia sivuvaikutuksia ja bugeja näiden suunnitteluvirheiden takia.

Käsitteellä viitataan spesifisti eräänlaiseen intuitiiviseen tunteeseen siitä että kaikki ei ole niinkuin pitäisi, eikä näitä virheitä aina pystykään yksilöimään mihinkään tiettyyn asiaan, mutta on myös kehitetty erilaisia tarkastelutapoja joilla voidaan arvioida onko tuntemus design smellseistä oikea. Kevyimmillään design smellit ovat lähinnä epäoptimaalisia koodin rakenteita, jotka voidaan korjata helposti uudelleenfaktoroinnilla, ja jotka suorastaan vaativat tällaista käsittelyä. Pahimmillaan taas on kysymys siitä, että esimerkiksi eri osissa järjestelmää käsitellään samaa informaatiota eri nimillä, joskus myös eri yksiköillä (esim. toisessa kohdassa määritellään sama lukuarvo prosentteina ja toisessa desimaaleina), mikä voi pahimmassa tapauksessa aiheuttaa katastrofaalisia seurauksia.

Joitakin yleisiä design smellejä ovat muun muassa:

- **Abstraktiotason puuttuminen**, eli se että käytetään ylenmääräisen paljon irrallisia muuttujia ja muita datamöykkyjä sen sijaan että ne olisi koherentisti määritelty luokkien sisälle.
- **Monitahoinen abstraktio**, eli se että luokka tekee monia, toisistaan irrallisia asioita, sen sijaan että se olisi keskittynyt tietyn rajatun toiminnallisuuden tarjoamiseen.
- **Moninkertainen abstraktio**, eli että on monia luokkia jotka tekevät samoja asioita.
- **Rikkonainen modularisaatio**, eli tilanne jossa toisiinsa liittyvien data-esineiden pitäisi olla saman luokan alaisina, mutta ne ovatkin jaettuna useille eri luokille.

Design Patterns: Design Patternit[17] ovat ohjelmistotuotannossa yleisesti hyödyllisiksi huomattuja rakenteita ja toistuvia kaavoja, joiden tunteminen edistää merkittävästi ohjelmistokehittäjän osaamista. Osa näistä kaavoista on niin perustavanlaatuisia osia nykyaikaista ohjemointityötä, esim. Factory Method, että monet niitä käyttävät kehittäjät eivät välttämättä varsinaisesti edes tiedosta käyttävänsä jotain sellaista mallia joka mielletään Design Patterniksi.

4.9 Tee yksikkö- ja muut automaattiset testit

Yksikkötestaus on työskentelymetodi, jota käytetään standardinomaisesti ohjelmistokehityksessä varmentamaan koodin toimivuus. Nimensä mukaisesti yksikkötestauksessa käydään automatisoidusti läpi ohjelmiston "yksiköitä" eli pienimmän tason komponentteja, jotka useimmissa kielissä ovat metodeja tai funktioita, siten että kirjoitetaan toinen funktio jonka on tarkoitus testinomaisesti ajaa testattava funktio tietyillä parametreilla ja vertailla sen palauttamaa tulostetta ennalta määriteltyyn, testattavan funktion logiikan mukaiseen lopputulokseen. Yksikkötesti palauttaa arvona joko TRUE tai FALSE sen mukaisesti vastasiko testin kohteena olevan funktion antama palautusarvo sitä mitä sen piti.

Nykyaikana yksikkötestaus on siinä määrin merkittävä osa ohjelmistokehitystä että sitä käytetään yleisesti ottaen kaikissaa ohjelmistokehityksen skaaloissa, pienistä massiivisiin projekteihin, ja varsinkin isommissa yrityksissä on usein nimenomaan yksikkötestien suunnittelua ja toteutusta varten omat ohjelmistokehittäjänsä. Pienemmissä tiimeissä yksikkötestit kirjoittaa yleensä sama kehittäjä joka on kirjoittanut itse testattavan koodinkin.

Tarkastelkaamme yksikkötestien logiikkaa yksinkertaisella esimerkillä. Oletetaan Python -kielinen funktio:

```
def substract_two_numbers(x, y):
    return x - y
```

Tätä funktiota varten kirjoitettaisiin yksikkötestit:

```
def test_substract_positives():
    result = substract_two_numbers(5, 40)
    assert result == -35
```

[17] 22 Classic Design Patterns

```
def test_substract_negatives():
    result = substract_two_numbers(-4, -50)
    assert result == 46


def test_substract_mixed():
    result = substract_two_numbers(5, -5)
    assert result == 10
```

Yksikkötestausta voidaan lähteä toteuttamaan muutamalla eri tavalla, jotka eroavat toisistaan jossain määrin. Esimerkiksi "Test-driven development" on ohjelmistokehityksen metodologia, jossa yksikkötestausta käytetään aktiivisesti toiminnallisten vaatimusten todentamisessa. Ideana on siis se, että testit kirjoitetaan ensin, niin että niiden ehdot vastaavat ohjelmistolle asetettuja toiminnallisia vaatimuksia, jolloin sitten kun varsinainen ohjelmakoodi kirjoitetaan testi ei pelkästään testaa sen toimivuutta, vaan toimii todistuksena siitä että ohjelmisto noudattaa tarkasti niitä vaatimuksia jotka sille on asetettu. Päinvastaisessa ajattelumallissa taas yksikkötestit kirjoitetaan heti varsinaisen koodaustyön jälkeen, jotta koodin toimivuus voidaan todentaa mahdollisimman nopeasti. Yksikkötestaus on myös oleellinen osa CI/CD -putkistojen toimintaa, siten että yksikkötestaus on kiinteästi integroitu automaattiseen julkaisuputkeen. Yksikkötestausta ja muita ohjelmistokehityksen laadunvalvontamenetelmiä käsitellään laajemmin kurssilla Software Testing and Quality Assurance.[18]

[18] Kurssi: Software Testing and Quality Assurance

V -malli

Agilen ja vesiputousmallin väliin mahtuu joitakin muitakin ohjelmistokehityksen filosofioita, joista yksi on niinsanottu V -malli[19], joka on sovellus laajemmassa kontekstissa järjestelmien suunnitteluun tarkoitetusta V-mallista.[20] Tässä pelkkää ohjelmistokehitystä laajemmassa kontekstissa V-mallia käyttävät mm. Saksan ja USAn liittovaltiotason hallintokoneistot erilaisten projektien suunnittelussa. Nimensä tämä malli on saanut siitä että sen kehitysvaiheiden voidaan nähdä asettuvan ikäänkuin V-kirjaimen muotoiseen muodostelmaan kun ne esitetään loogisessa yhteydessä toisiinsa, siten että V:n vasemmalle puolelle asettuvat suunnittelu- ja kehitysvaiheet ja oikealle puolelle niiden kanssa korreloivat testaus- ja ylläpitovaiheet. Mallin ideana on että kutakin suunnittelun vaihetta mittaavat testit kirjoitetaan siinä vaiheessa kun suunnittelua tehdään,V-mallia käytetään ohjelmistokehityksen piirissä nykyaikana pääsosin terveysteknologisten laitteiden kehityksessä, koska tietyistä syistä johtuen ketterä kehitysfilosofia ei ole optimaalinen tällaiseen työskentelyyn.[22]

Muut testausmetodologiat: Test-driven developmentin lisäksi käytetään myös muita metodologioita, joissa ohjelmiston testaaminen yhdistetään kehitystyöhön hivenen eri tavoilla.

4.10 Implementoi

Joskus jopa hiukan koodataan. Koska tämä ei kuitenkaan ole koodauskurssi, käsittelemme tässä luvussa sellaisia koodaamiseen liittyviä ohjelmistokehityksen työskentelykäytäntöjä, joiden ymmärtäminen, sisäistäminen ja omaksuminen osaksi omaa työskentelyä on oleellista ohjelmistokehittäjän roolissa.

Aiemmilla kursseilla on sivuttu koodauksen opetuksen lomassa sitä, että koodin oikeanlainen sisentäminen, kommentointi ja muuttujien nimeämiskäytännöt ovat oleellisia asioita kokonaisuuden kannalta. Näiden käytäntöjen merkitystä voi kuitenkin olla vaikea ymmärtää mikäli oma koodauskokemus on vielä vähäistä ja erityisesti jos suurin osa siitä kokemuksesta koostuu yksin tapahtuneesta henkilökohtaisten projektien työstämisestä. Silloin kun kirjoittaa koodia jota käytännössä kukaan muu ei koskaan lue on helppo totuttaa aivonsa siihen että sisennykset voivat mennä ihan miten sattuu, kommentointia on ehkä joskus jossain ja nimeämiskäytännötkin loistavat lähinnä koherenssin puutteellaan. Koodi kuitenkin toimii ja tiedät itse miksi se tekee mitäkin, koska olet sen itse kirjoittanut alusta loppuun. Vaikka monissa ohjelmointikielissä

ja viitekehyksissä on hyvinkin tarkasti määritellyt "best practicet" jotka yleensä myös koskettavat näitä asioita, ja vaikka jotkut kielet myös pakottavat toimimaan ainakin jossain määrin tietyllä tavalla (esim. Pythonin sisennys), nämä eivät kuitenkaan koodista tee luettavampaa jos koodari ei näitä käytäntöjä noudata.

Mutta mietitäänpä tilannetta jossa työskennellään vähän isommalla porukalla, vähän isomman projektin parissa ja vähän isommilla panoksilla, yleensä siten että joku on maksanut siitä koodin kirjoittamisesta ja että sen koodin pitää olla valmista tiettyyn ajanhetkeen mennessä. Mikäli tällaisessa tilanteessa kaikki kehittäjät toimivat yläpuolella kuvaillulla tavalla, ei työstä tule mitään. Ja jos edes yksi toimii mainitulla tavalla, hän todennäköisesti saa potkut jos ei muuta toimintaansa, koska muille koodin kanssa työskenteleville koituu kohtuutonta ja tarpeetonta vaikeutta siitä että he joutuvat elämään toisen koodarin huonojen työkäytäntöjen kanssa. Jos työajasta menee merkittävä osa toisen tekijän huonosti sisentämän, sekavasti nimetyn ja kommentoimattoman koodin pelkkään tulkitsemiseen, ei se suoranaisesti edistä työmotivaatiota tai tehokkuutta.

Tästä syystä käytännössä kaikki ohjelmistokehitystä tekevät yritykset haluavat, että kaikki heille työskentelevät koodarit noudattavat samoja käytäntöjä kommentoinnissa, sisennyksessä ja muuttujien nimeämisessä. Jotkut yritykset pyrkivät tässä jopa niin suureen yhdenmukaisuuteen, että koodista ei olisi havaittavissa mitään sen kirjoittaneen koodarin yksilöiviä tyylipiirteitä. Tämä kaikki yksinkertaisesti sen takia, että tällaisten koodin luettavuuteen ja selkeyteen vaikuttavien toimintatapojen vaikutus työtehoon ja sen kautta tulokseen on merkittävä. Tästä syystä näiden käytäntöjen päättäminen yhteisesti tiimin kanssa, kaikkien työntekijöiden sitoutuminen niihin ja se että niitä noudatetaan kunnes ne muuttuvat automaatioiksi joita ei edes tarvitse miettiä koskaan onkin yksi ohjelmistokehityksessä oppimisen merkittävä osa-alue.

4.11 Hanki valmiit komponentit ja konfiguroi

Suuri osa nykyaikaisesta ohjelmistokehityksestä perustuu valmiin koodin hyötykäyttöön. Tämän osuus tietysti vaihtelee ohjelmointikielestä ja tarkoituksesta toiseen, mutta karkeasti ottaen on olemassa hyvin vähän sellaisia asioita joita kukaan toinen ei olisi jo jollain tavalla toteuttanut ja luonut tästä toteutuksesta kirjastoa, jota käyttämällä voidaan säästää massiivinen määrä

työtä ja aikaa ja harmaita hiuksia. Hyötyohjelmien puolella tämä tarkoittaa useimmiten erilaisten valmiiden moduulien hyödyntämistä, mutta esim. peliohjelmoinnissa se voi tarkoittaa kokonaisten pelimoottorien käyttämistä oman luomuksen pohjana.

Jotkin viitekehykset ja kielet painottuvat enemmän tällaiseen valmiiden kirjastojen käyttöön, esim. React ja Java ja näitä kieliä käytettäessä suuri osa koko ohjelmistoprojektista voikin olla valmiiden palikoiden käyttöä tai vähintäänkin näiden valmiiden osien päälle rakennettujen toimintojen luomista. Tämä johtuu pitkälti siitä että molempien kielien ympärille on muodostunut valtava kehittäjäkulttuuri, joka on luonut varteenotettavan määrän ilmaisia kirjastoja ja moduuleja jotka on äärimmäisen helppo omaksua ja ottaa käyttöön.

Mitä erikoistuneempaan tai laajempaan käyttöön kirjasto on tarkoitettu ja mitä monimutkaisempaa teoreettista osaamista sen kirjoittaminen on vaatinut, sitä todennäköisempää on että kyseisen kirjaston käytöstä joutuu maksamaan jotain. Joskus, esimerkiksi isojen pelimoottorien kuten Unreal Enginen tai Unityn kohdalla maksullisuus perustuu siihen miten suurta voittoa näitä työkaluja hyödyntämällä tehdään, siten että ilmaisia tai lähes ilmaisia pelejä voi rakennella ihan miten paljon tahansa, mutta heti kun pelin myynnistä saadut tulot ylittävät tietyn pisteen ottaa pelimoottorin luonut yritys siitä oman provisionsa. Toinen yleinen laskutusmalli on myös se että lisenssimaksut skaalautuvat suoraan kehittäjätiimin jäsenmäärän mukaan. Joissakin tapauksissa maksamalla saa myös muita palveluja moduulin tai kirjaston kehittäneeltä taholta, esimerkiksi teknistä tukea ja konsultaatiota mahdollisten ongelmatilanteiden tullessa vastaan.

4.12 Hallitse lähdekoodia ja muita assetteja versionhallinnassa (GIT)

versiohallintajärjestelmien keskeinen idea on nimensä mukaisesti siinä, että ohjelmistokoodin kehitystä voidaan hallita tehokkaasti, vanhoihin versioihin voidaan tarvittaessa palata helposti ja useat kehittäjät voivat vaivattomasti työstää saman ohjelmiston eri osa-alueita olematta edes suoraan yhteydessä toisiinsa. Nykyaikana kaikki ohjelmistokehitystä ammatillisesti tekevät tahot käyttävät versiohallintajärjestelmiä, useimmiten Git -ohjelmaa, ja tästä syystä tämän teknologian tuntemus onkin oleellista kaikille ohjelmistokehitystä opiskeleville ja tekeville hen-

kilöille. Tulee myös huomioida että varsinaisen versionhallinnan lisäksi oman koodirepositorion kasvattaminen GitLabiin tai GitHubiin toimii myös todisteena sille että henkilöllä on koodauskokemusta, minkä vuoksi varsinkin aloittelevien ohjelmistokehittäjien kannattaakin lähtökohtaisesti ladata kaikki tekeleensä, aina koulutöistä harrastusprojektehin, johonkin tällaiseen palveluun. Tätä kautta omaa koodausportfoliota on helppoa esitellä esim. linkittämällä sen ansioluetteloon.

Kuten aiemmin olemme jo todenneet, yleensä kun nykyään puhutaan versiohallintajärjestelmistä tarkoitetaan Git -ohjelmistoa[21], jonka kehittämisen suomalainen Linus Torvalds aloitti vuonna 2005. Ennen tätä erilaisia versiohallintajärjestelmiä oli ollut olemassa jossain muodossa jo 1960 -luvulta lähtien[22], eikä Gitin kehityksen taustalla varsinaisesti ole mitään muuta syytä kuin se että Torvalds tarvitsi versiohallintajärjestelmän Linux -projektin tarpeisiin eikä halunnut maksaa muiden tällaisten järjestelmien lisenssimaksuja.

Gitistä on viimeisen 19 vuoden aikana muodostunut versionhallinnan standardi, jota käytetään käytännössä kaikilla tietotekniikan osa-alueilla ja jonka ympärille on muodostunut palveluntarjoajia kuten GitHub ja GitLab, jotka ylläpitävät massiivisia koodivarantoja. Joidenkin arvioiden mukaan Gitin osuus kaikesta versionhallinnasta oli vuonna 2022 jo 93.3 %[23], mikä tarkoittaa sitä että se on käytännössä hävittänyt kaiken kilpailun näiltä markkinoilta. Syitä tähän hyvin nopeasti tapahtuneeseen ja hyvin totaaliseen markkina-aseman valloitukseen voidaan arvailla, mutta yleisesti ottaen ohjelmiston ilmaisuus, tietoturvan korkea taso, hajautettu repositoriostruktuuri ja tehokkuus jopa verrattain suuria koodivarantoja käsitellessä ovat asioita joita pidetään sen menestyksen taustalla vaikuttavina tekijöinä.

Gitin käyttöönotto on hyvin yksinkertaista, eikä oikeastaan tarvitse muuta kuin ohjelmiston lataamisen sille koneelle jossa sitä halutaan käytettävän. Tämä johtuu pitkälti siitä että Git perustuu hajautettuun varantorakenteeseen. Toisinsanoen, Git on lähtökohtaisesti suunniteltu siihen että jokainen kehittäjä ylläpitää omaa koodivarantoaan lokaalisti omalla koneellaan, minkä ansiosta mitään keskusvarantoa ei välttämättä tarvita. Isoissa ohjelmistoprojekteissa

[21] Git Download

[22] Version Control Systems

[23] Wikipedia: Git

tietysti käytetään tällaista keskusvarantoa, johon kaikki projektissa työskentelevät kehittäjät lisäävät koodia.

Gitin keskeinen hyödyllisyys tulee siitä, että sitä käytettäessä koodi voidaan haarauttaa[24] eri kehityslinjoihin, jotka voidaan sitten myöhemmässä vaiheessa sulauttaa[25] takaisin yhdeksi linjaksi. Tämä mahdollistaa sen että eri osa-alueita koodista voidaan muokata täysin toisistaan eristetysti erillisissä kehityshaaroissa, mikä mahdollistaa tehokkaasti esimerkiksi toisistaan eriävien rinnakkaisten versioiden luomisen samasta ohjelmistosta ja erilaisten kokeilujen tekemisen turvallisesti niin että päälinjan versiohistoria pysyy puhtaana. Myös koko repositorio voidaan monistaa[26] helposti, jos halutaan esimerkiksi samasta pohjakoodista lähteä rakentamaan kahta merkittävästi erilaista ohjelmistoa.

4.13 Integroi ja asenna (CI/CD)

Kuten käsittelimme jo lyhyesti luvussa 3.8.2, CI/CD -putkistojen käyttö on oleellinen osa nykyaikaista ohjelmistokehitystä. Tämän teknologian avulla ohjelmiston päivittäminen, eli vikojen korjaaminen ja uusien ominaisuuksien vieminen tuotantoon voidaan automatisoida lähes täysin, niin että ihmistyön määrä koko prosessissa voidaan minimoida käytännössä pelkästään koodin kirjoittamiseen, joskin tässä on putkistokohtaisia eroja. Kun puhutaan Continuous Delivery -putkistosta, tarkoitetaan järjestelmää jossa ihminen kuitenkin loppujenlopuksi kuittaa muutokset ennen niiden julkaisua, kun taas Continuous Deployment -putkistossa tämäkin osuus on täysin automatisoitu. Tässä luvussa perehdymme hivenen yksityiskohtaisemmin CI/CD -putkiston toimintaperiaatteisiin ja siihen miten tällainen järjestelmä otetaan käyttöön. Kuten jo nimestäkin voi päätellä, jakautuu tämä teknologia kahteen osa-alueeseen, Continuous Integrationiin, eli jatkuvaan integraatioon ja Continuous Delivery/Deploymentiin, eli jatkuvaan jakeluun/käyttöönottoon.

Jatkuva integraatio pitää sisällään sen osan järjestelmästä, jossa uudet muutokset koodissa testataan, yhdistetään ohjelmiston pääasialliseen runkoon ja kootaan[27] valmiiksi ohjelmaksi.

[24] branch

[25] merge

[26] fork

[27] engl. build

Koko tämän prosessin ideana on se että se tapahtuu täysin automaattisesti, niin että ihmisen täytyy ainoastaan kirjoittaa koodia, putkiston hoitaessa kaiken muun.

Jatkuva jakelu/käyttöönotto taas on vaihe joka seuraa jatkuvaa integraatiota, siten että kun koodi on koottu toimivaksi ohjelmaksi se siirretään suoraan joko testi- tai tuotantoympäristöön ja otetaan käyttöön. Kuten on tullut jo todettua, tässä vaiheessa on hienoinen ero jakelun (engl. Delivery) ja käyttöönoton (engl. Deployment) välillä, koska ensimmäisessä tapauksessa ihminen varmistaa integraatiovaiheen tulosten oikeellisuuden ja manuaalisesti antaa käskyn ohjelman siirtämiseksi käyttöön, kun taas jälkimmäisessä nämäkin toiminnot on täysin automatisoitu.

Käytännössä tällaisen järjestelmän rakentaminen voi tapahtua erilaisilla työkaluilla, jotka on todennäköisimmin päätetty organisaatio- tai tiimitasolla ja konfiguroitu jonkun tämäntyyppiseen työhön erikoistuneen tekijän toimesta, niin että ohjelmistokehittäjän tarvitsee vain opetella käyttämään järjestelmää. Pienemmissä projekteissa ja yrityksissä voi tietysti olla niin että tämänkin työn tekee joku kehittäjä muiden töidensä ohella. Esimerkiksi sekä GitLab että GitHub tarjoavat omat työkalunsa tähän työhön, jotka voidaan yhdistää täysin projektin versionhallinnan kanssa, mutta muitakin CI/CD -putkien toteuttamistapoja on olemassa.

4.14 Kommunikointi ja koordinointi

4.14.1 Kokouskäytännöt

Jotta tiimityöskentely toimisi järkevästi, pitää tiimin jäsenten välisen kommunikaation toimia. Käytännössä kaikissa organisaatioissa yksi keskeisimmistä työskentelytavoista tämän saavuttamiseksi ovat säännölliset kokoukset ja palaverit joissa työn suunnittelu suureksi osaksi tapahtuu ja jossa voidaan keskustella kullakin hetkellä oleellisista asioista ja pitää muut tiimin jäsenet tietoisina aikatauluista ja työn etenemisestä. Projektin alkuvaiheessa tuleekin määritellä säännölliset käytännöt ja aikataulut kokousten pitämiselle, sekä sopia siitä mihin kokouksiin kenenkin on välttämätöntä osallistua. Joihinkin työskentelymetodeihin kuten Scrumiin jatkuva kokoustaminen kuuluu oleellisesti, ja sitä sovellettaessa voidaan pitää jopa päivittäin lyhyitä kokouksia (engl. Daily Scrum), jossa vaihdetaan oleelliset tiedot asioiden päiväjärjestyksestä. Ilmankin Scrumin mukaista työn rytmitystä on kuitenkin melko yleistä että vähintään kerran

viikossa on kokous jossa raportoidaan omasta etenemisestä, ja vähintään kerran kuukaudessa vähän pidempi suunnittelusessio.

4.14.2 Viestintä- ja tiketointiohjelmistot

Oleellinen osa tehokasta tiimityöskentelyä on kommunikaation järjesteleminen niin että se kuluttaa mahdollisimman vähän resursseja itse työnteolta. Kokoustamisen lisäksi tätä varten on nykyaikana tarjolla useita erilaisia työkaluja jotka mahdollistavat niin jatkuvan tiedonvälityksen kuin työtehtävien organisoimisen siten että niiden status on reaaliaikaisesti kaikkien tekijöiden tiedossa. Ensimmäiseen osioon kuuluvat lukuisat viestisovellukset kuten Slack, Teams ja Mattermost yms. jotka on nimenomaisesti suunniteltu työkontekstiin, ja jälkimmäiseen osioon erilaiset tiketointijärjestelmät joilla työtehtäviä voidaan tehokkaasti jakaa tiimin sisällä.

Erilaiset viestijärjestelmät ovat nykyaikana täysin arkipäiväisiä, mutta tämänkin tyyppisissä työkaluissa on erikseen joukko sellaisia ohjelmistoja jotka on tarkoitettu ensisijaisesti työpaikkojen ja muiden virallisten organisaatioiden viestintätarpeisiin. Toisin kuin ensisijaisesti vapaa-ajan viestintään tarkoitetuissa systeemeissä kuten WhatsAppissa tai Signalissa on sellaisissa järjestelmissä kuten Slack tai Mattermost mietitty käytettävyyden kannalta ominaisuuksia jotka edistävät nopeiden kommunikaatioyhteyksien luomista muiden tiimin jäsenten kanssa, viestiketjujen haarautumista erillisiin aiheisiin ja viestihistorian varmaa saatavuutta, mikä mahdollistaa sen että käyttäjät voivat palata kohtuullisen yksinkertaisesti vanhoihinkin aiheisiin. Joissakin tällaisissa sovelluksissa on myös sisäänrakennettuja videopuheluominaisuuksia, jolloin kaikki viestintä voidaan tehokkaasti keskittää yhden sovelluksen alle ja esim. videokokouksia varten ei tarvita omaa ohjelmistoaan.

Suurin osa työkäyttöön tarkoitetuista viestisovelluksista on maksullisia, mutta osittain juuri tämän takia yritykset preferoivat niitä sisäisessä viestinnässään, koska vastakohtaisesti sovellukset jotka ovat “ilmaisia” perustuvat käytännössä aina siihen että niissä liikkuvat tiedot myydään mainostajille, tekoälyjen opetusmateriaaliksi tai muihin kolmannen osapuolen tarkoituksiin. Tästä syystä niitä ei voida käyttää kun käsitellään liikesalaisuuksia tai muita sellaisia tietoja jotka eivät saa päätyä ulkopuolisten käsiin, mikä tarjoaa markkinaraon sellaisille viestintäohjelmistoille jotka perustuvat johonkin toiseen ansaintamalliin.

Tiketointiohjelmat ovat sovelluksia, joiden avulla työtehtäviä voidaan allokoida tehokkaasti

ja dynaamisesti ja seurata tehtävien toteutumista minimaalisella vaivalla. Yleisesti ottaen näitä järjestelmiä on alunperin käytetty lähinnä asiakaspalvelupuolella, mutta ne ovat siirtyneet myös IT -alan arkipäiväksi mahdollistamansa kommunikaation helppouden vuoksi. Monet nykyaikaiset tiketointijärjestelmät voidaan integroida esim. Slackin kanssa siten, että tiketit menevät suoraan Slackin kautta halutulle käyttäjälle, mikä tehostaa tiketointiprosessia entisestään.

4.15 Asiakkaan tarpeiden oppiminen iteratiivisesti

Vaikka projektin pohjustustyö vaatimusmäärittelyineen olisi tehty asiallisesti, on varsin tavallista että sitä mukaa kun työ etenee ja välitavotteita, esimerkiksi sprinttien tuloksia ja muita saavutettuja milestoneja esitetään asiakkaalle, nousee esille uusia päämääriä joita ei aiemmin ole tultu otetuksi huomioon.

Tämä johtuu pitkälti siitä että siinä vaiheessa kun asioita suunnitellaan vasta paperilla eikä niiden toimintaa ole vielä nähty käytännössä voi asiakkaalla olla hyvinkin vääränlainen kuva siitä mitä oikeastaan halutaan ja sitten kun siitä on tehty ensimmäinen versio juuri kuten asiakas tarpeensa kuvaili, asiakas onkin sitä mieltä että ei tämä nyt ollutkaan sitä mitä haluttiin.

Toisaalta, johtuen ohjelmistojen monimutkaisesta luonteesta voi projektin alkuvaiheiden suunnittelutyössä yksinkertaisesti unohtua sellaisia asioita, jotka kuitenkin ovat hyvinkin välttämättömiä ohjelmiston toiminnalle, tai ainakin järkevälle käyttäjäkokemukselle.

4.16 Ohjelmistokehityksen mittaaminen

Ohjelmistokehityksen mittaamisella tarkoitetaan tässä tapauksessa sekä käytettävissä olevan ajan ja resurssien suhteen, että projektin tulosten mittaamista jollain sellaisella tavalla joka on projektin luonteen puolesta järkevä. Projektin etenemisen mittaaminen ei ole tärkeää pelkästään sen takia että kehitystiimi pysyy kärryillä siitä missä vaiheessa työ etenee, vaan selkeillä mittauskäytännöillä viestitään myös stakeholdereille ja muille sidosryhmille projektin etenemistä. Projektin edeistymisen mittaamiseen on kehitetty erilaisia käsitteitä ja työkaluja, esim. Burndown Charteja ja Milestoneja, jotka käymme läpi pääpiirteittäin.

- **Burndown Chart[28]:** Graafi, jolla on tarkoitus kuvata vielä jäljellä olevan työn määrää

[28] Burndown Chart

suhteessa käytössä olevaan aikaan. Käytännössä tämä voi tarkoittaa esim. sitä että verrataan Sprint Backlogin instanssien määrää käytettävissä olevien työpäivien määrään, mistä voidaan arvioida se kuinka monta työtuntia mihinkin tehtävään on mahdollista käyttää.

- **Milestones[29]:** Kotoisammin virstanpylväät ovat ohjelmistoprojektille määriteltyjä väliaikatavoitteita, joiden saavuttaminen kertoo siitä että projekti on edennyt tiettyyn vaiheeseen. Milestonejen käyttö on nykyään täysin standardinomainen projektin kehityksen mittaamistapa, jota sovelletaan laajalti myös muilla teollisuuden aloilla. Milestonet määritellään yleensä ennen projektin alkua, mutta ne voivat muuttua projektin aikana ketterän kehitysparadigman vaatimusten mukaisesti. Tietyssä mielessä monia tällä kurssilla käsiteltyjä ohjelmistoprojektin osa-alueita voisi itsessään pitää yleisen tason milestoneina, joiden saavuttaminen tapahtuu tietyssä järjestyksessä.
- **Team Velocity[30]:** Käsite joka kytkeytyy oleellisesti käyttäjätarinoihin, eli abstraktioihin joilla kuvataan ohjelmiston käyttötilanteita. Team Velocity mittaakin sitä kuinka moneen tällaiseen käyttäjätarinaan edellinen kehitysiteraatio, oli se sitten Scrumin sprintti tai jonkin muun työnjohtojärjestelmän jakso, loi vastauksen. Näiden toteutettujen käyttäjätarinoiden pohjalta voidaan sitten luoda arvioita siitä kuinka pitkään projektissa vielä suunnilleen menee.

4.17 Retrospektiivit eli työskentelytavan mukauttaminen

Sitä mukaa kun ohjelmistoprojekti etenee, nähdään selvemmin se onko valittu lähestymistapa järkevä juuri tähän projektiin, vai pitääkö toimintamalleja muuttaa jollain tavalla optimaalisemman lopputuloksen aikaansaamiseksi. Periaatteessa tämänkaltaisessa mukautuvuudessa, eli siinä että tarkastellaan asioita “jälkikäteen” ts. retrospektiivissä, on koko ketterän kehitysfilosofian ydin; siinä että asioita ei lyödä ehdottomasti lukkoon etukäteen, vaan että toimintamalleja, lähestymistapoja ja työskentelymetodeja voidaan vaihtaa sen mukaan mikä on järkevintä käsillä olevien ongelmien ratkaisemiseksi.

Retrospektiivien merkitys ohjelmistokehityksen toimivuudelle on siinä määrin hyödylliseksi

[29] Milestones

[30] Team Velocity

tunnustettu tosiasia, että ne on kodifioitu osaksi virallista Scrum -rutiinia[31]. Scrumissa retrospektiivin ideana on ensisijaisesti koota tiimi yhteen keskustelemaan siitä olivatko edellisen sprintin toteutukseen käytetyt työkalut päämääriin soveltuvia ja miten asiaa pitäisi jatkossa kehittää. Retrospektiivin ideana ei siis ole löytää syyllisiä siihen mikä meni pieleen tai ylipäänsä muutenkaan arvioida itse työn laatua, vaan ainoastaan tarkastella sitä olivatko käytetyt työkalut tehtävään sopivia.

4.18 Seuraa järjestelmän kehityksen liittyviä riskejä ja reagoi niihin

Ohjelmiston kehityskaareen liittyy monia riskitekijöitä, joita pitää tarkkailla ja joihin pitää puuttua mahdollisimman varhaisessa vaiheessa, jota nämä riskitekijät eivät muutu oikeiksi ongelmiksi myöhemmin kehitysvaiheessa. Vaikka ongelmia voidaankin aina ratkoa, on niiden torjunta huomattavasti helpompaa oikeanlaisella ennakoinnilla ja valmistautumisella.

Periaatteessa riskejä on kahdenlaisia, teknisiä ja projektin hallintaan liittyviä. Käymme seuraavassa läpi joitakin yleisimpiä tällaisia riskitekijöitä.

Tekniseen toteutukseen liittyviä yleisiä riskejä:

- **Suoritustekniset riskit[32]:** Tarkoittavat käytännössä sitä, että ohjelmiston suunnittelussa tai toteutuksessa on mahdollista tehdä sellaisia virheellisiä valintoja, jotka johtavat ohjelmiston epäoptimaaliseen toimintaan. Tämä ei tarkoita pelkästään sitä että ohjelmistossa on bugeja, vaan myös sitä että vaikka ohjelmisto teoriassa toimiikin oikein, se esimerkiksi vaatii merkittävästi enemmän laskenta- tai tiedonsiirtokapasiteettia kuin mitä se oikein optimoituna vaatisi.

- **Tietoturvariskit:** Oikeastaan kaikissa ohjelmiston kehitysvaiheissa on mahdollista tehdä virheitä, jotka johtavat tietoturva-aukkoihin, joita potentiaalisesti voidaan käyttää hyökkäyspolkuina.

- **Integraatioon liittyvät riskit:** Integraatiot muiden järjestelmien ja osajärjestelmien

[31] Sprint Retrospective

[32] engl. Performance Risks

kanssa pitää suunnitella huolellisesti, jotta ne eivät aiheuta yhteensopivuusongelmia tai muodosta pääsyä vääränlaiseen dataan.

Projektin hallintaan liittyviä yleisiä riskejä:

- **Laajuusriskit:**[33] Jos projektin vaatimusmäärittelyjä ei ole tehty asianmukaisesti, on vaarana että uusia tavoitteita tulee jatkuvasti lisää kun kehitystyön aikana havaitaan että jotain oleellista puolta ohjelmiston toiminnassa ei ole otettu huomioon. Tämä voi johtaa hyvin helposti tilanteeseen jossa projektin skaala karkaa käsistä ja työmäärä kasvaa mahdottomaksi toteuttaa.

- **Resurssiriskit:** Jos projektin budjetointia ja työmäärän arviointia ei ole tehty asianmukaisesti, on mahdollista että tekijöistä, rahasta, ajasta tai jopa vaadituista laitteistoista, esim. serverikapasiteetista, tulee pulaa jossain kriittisessä vaiheessa.

- **Kommunikaatio-ongelmat:** Toimiva kommunikaatio projektin eri osapuolien, eli toteutustiimin, sidosryhmien, asiakkaiden ja päätöksiä tekevien tahojen välillä on oleellista ohjelmistoprojektin toteutumiselle. Mikäli kommunikaatiossa on ongelmia, eivätkä kaikki ole samalla kartalla siitä mitä ollaan tekemässä, voi tämä heijastua negatiivisesti työn etenemiseen ja lopputuloksiin.

Riskienhallintaa pitää tehdä aktiivisesti koko ohjelmiston elinkaaren ajan. Käytännössä tämä tarkoittaa sitä, että pidetään säännöllisiä palavereja kaikkien osapuolien välillä, joissa käydään läpi potentiaalisia riskitekijöitä ja päätetään siitä miten niitä torjutaan tehokkaimmin. Muita keinoja riskienhallintaan ovat erilaiset laatuarvioinnit, prototyyppien kehittäminen uusille ominaisuuksille ennen kuin niitä lähdetään toteuttamaan ja kokeneiden kehittäjien ja suunnittelijoiden ottaminen mukaan projektiin tarkastelemaan ohjelmiston kokonaisuutta ja antamaan ulkopuolisen asiantuntijan arvion siitä mitä riskejä on havattavissa.

4.19 Osajärjestelmien integrointi

Nykyaikana oikeastaan kaikki ohjelmistot koostuvat useista eri osista, jotka jollain tavalla integroidaan toistensa kanssa niin että lopputulos toimii kuin yksi järjestelmä. Jopa yksinkertaiset

[33] engl. Scope Risks

verkkosivut on nykyaikana useimmiten toteutettu niin, että käyttäjälle näkyvä osa on toteutettu yhdellä tekniikalla, ja käyttäjälle näkymättömät mekanismit toisella, ja nämä on jollain kolmannella tekniikalla “integroitu” toisiinsa niin että ne toimivat yhdessä. Tämä on kuitenkin jossain määrin eri asia kuin se “integraatio” josta puhutaan kun laitetaan kaksi tai useampi isoa järjestelmää keskustelemaan keskenään ja tekemään yhteistyötä.

Käytännössä isommissa ohjelmistoja tekevissä yhtiöissä on jokaista tällaista osajärjestelmää varten vähintään yksi, joskus useampia järjestelmäarkkitehteja, jotka muun suunnittelun ohessa yhteistyössä muiden osajärjestelmien arkkitehtien kanssa varmistavat että osajärjestelmät osaavat keskustella keskenään. Tällöin puhutaan skaalasta jossa jokaista tällaista osajärjestelmää kehittää oma tiiminsä, tai jopa tiimien joukko. Pienemmissä kuvioissa, joissa osajärjestelmät ovat sekä yksinkertaisempia että niitä on vähemmän voi yksikin henkilö suunnitella kaikkien integraatioiden toiminnan.

4.20 Betatestaus, järjestelmän skaalautuvuus ja kuormituksen kesto

Ennen kuin ohjelmisto voidaan julkaista loppukäyttäjille, sen toimintaa ja erityisesti sen sietokykyä oikeille käyttövolyymeille pitää testata laajamittaisesti. Käytännössä tämä tarkoittaa sitä, että ohjelmistoa ajetaan testiympäristössä simuloiduilla käyttäjämäärillä joiden oletetaan vastaavan tulevaa käyttöä, ja vähän enemmänkin, jotta voidaan varmistaa että kaikki järjestelmän osa-alueet kestävät kuormitusta jonka oikea käyttäjämassa tulee aiheuttamaan.

Julkaisua edeltävää testausta voidaan tehdä myös oikeita käyttäjiä vastaavilla testaajilla, jotka rekrytoidaan oman prosessinsa kautta, ja jotka saavat rajoitetun pääsyn testiympäristöön joka mahdollistaa sen että he voivat käyttää ohjelmistoa kuten sitä on tarkoitus käyttää. Tällöin puhutaan alpha- ja betatestauksesta, jossa on tarkoitus testaajien avulla löytää bugeja ohjelmiston toiminnasta ja yleisesti ottaen selvittää laajamittaisesti se että kaikki ohjelmiston ominaisuudet toimivat kuten niiden pitäisi.

Tällaista alpha- ja betatestausprosessia voidaan myös käyttää osana ohjelmiston markkinointia, jonka kautta valikoidulle käyttäjäjoukolle annetaan “etuoikeutettu” pääsy uuteen ohjelmistoon, millä voi olla merkittävä painoarvo ohjelmiston tulevaan kaupalliseen menestykseen

sen luoman julkisuusvaikutuksen myötä. Facebook, joka oli alunperin rajoitettu pelkästään yhdysvaltalaisten yliopisto-opiskelijoiden käyttöön, on tästä loistava esimerkki. Facebook levisi varhaisessa vaiheessa vuosien 2004–2010 välillä ensin rajoittamalla käyttäjäkunnan yliopistojen, lukioiden ja valikoitujen yritysten käyttäjille, ja tämän saavutettua merkittävää suosiota avasi palvelun kaikille vuonna 2006. Merkittävää on kuitenkin se, että "betatestaukseen" rekrytoiminen aloitettiin vasta 2010, yhtiön jo kasvettua merkittäväksi toimijaksi, käytännössä siis täysin kehitysvaiheessa olevalla ohjelmistolla.

Peliteollisuudessa tällaisesta testausvaiheesta on muodostettu myös kaupallista toimintaa, koska usein *Early Access* -tilassa julkaistut pelit, joista käyttäjät kuitenkin maksavat itse, ovat käytännössä betatestausvaiheessa olevia raakileita, joiden toiminta ei ole välttämättä millään tavalla varmaa. Käytännössä nämä loppukäyttäjät ovat testaajia, jotka maksavat siitä että pääsevät tekemään työtä. Vielä kyseenalaisemman tästä käytännöstä tekee se, että jos peli ei menestykään tässä vaiheessa sen kehitystyö voidaan yksinkertaisesti lopettaa, eikä siitä maksaneilla kuluttajilla ole mitään takeita siitä että he koskaan saisivat rahoilleen todellista vastinetta tai rahojaan takaisin. Pelien jäämisestä ikuiseen *Early Access* -limboon onkin muodostunut jossain määrin ilmiö. Jotta tämä ei kuulostaisi aivan liian raadolliselta, voidaan kuitenkin todeta että varsinkin pienemmille ja aloitteleville pelejä kehittäville yrityksille *Early Access* -julkaisun tuomat rahavarat voivat olla ainut keino saada tarpeeksi resursseja viedä kehitystyö loppuun asti.

4.21 Käytettävyystestaus

Käytettävyystestaus on prosessi, jossa ohjelmistoa testataan oikeiden käyttäjien, tai näitä vastaavien toimijoiden, avustuksella, tavoitteena selvittää, kuinka helppoa ja intuitiivista ohjelmiston käyttö on. Tämä auttaa kehittäjiä ymmärtämään paremmin, miten heidän suunnittelemansa järjestelmä vastaa todellisten käyttäjien tarpeita, odotuksia ja käyttötapoja.

Käytettävyystestaus on empiirinen, käyttäjäkeskeinen prosessi, joka keskittyy käyttäjän kokemukseen. Testissä käyttäjät suorittavat tyypillisesti joukon tehtäviä ohjelmistolla, ja heidän suoritustaan seurataan ja arvioidaan. Käytettävyyden testauksessa voidaan mitata esimerkiksi

tehtävien suorittamiseen kulunutta aikaa, virheiden määrää, tehtävän suorittamisen onnistumista ja käyttäjien subjektiivista kokemusta.

Käytettävyystestauksen avulla pyritään tunnistamaan ohjelmiston mahdolliset ongelmat ja puutteet, jotka vaikuttavat negatiivisesti käyttäjän kokemukseen tai estävät häntä suorittamasta tehtäviä tehokkaasti. Se on osa käyttäjäkeskeistä suunnitteluprosessia ja sen tavoitteena on varmistaa, että ohjelmisto on käyttäjälle helppokäyttöinen, tehokas ja tyydyttävä.

Testauksen aikana voidaan käyttää erilaisia työkaluja ja menetelmiä, kuten haastatteluja, havainnointia, kyselyitä ja silmänliikkeiden seurantaa, riippuen siitä, mitä aspekteja käytettävyydestä halutaan tutkia. Testaajat voivat olla ohjelmiston todellisia käyttäjiä, tai he voivat olla käyttäjäprofiilin mukaisesti valittuja henkilöitä. Tärkeää on, että testauksessa otetaan huomioon käyttäjien monimuotoisuus, jotta saadaan mahdollisimman kattava kuva ohjelmiston käytettävyydestä erilaisille käyttäjille.

Käytettävyystestauksen tavoitteena on jatkuva ohjelmiston parantaminen ja käyttäjäkokemuksen optimointi. Kun käytettävyystestaus on integroitu osaksi ohjelmistokehitysprosessia, jokainen uusi versio tuotteesta on parempi kuin edellinen, ja lopullinen tuote on käyttäjälle miellyttävä ja helppokäyttöinen. Käytettävyystestauksen tulokset voivat osoittaa, mitkä ohjelmiston osat toimivat suunnitellusti ja mitkä eivät. Testauksen avulla voidaan myös tunnistaa käyttäjän tarpeet ja odotukset, jotka eivät välttämättä ole olleet selvillä kehitysprosessin aikana.

Käytettävyystestauksen tulokset auttavat ohjelmiston kehittäjiä ymmärtämään, missä ohjelmiston käyttöliittymässä on parantamisen varaa. Nämä tulokset voivat olla hyödyllisiä ohjelmiston kehityksen eri vaiheissa, aina alustavasta prototyypistä valmiiseen tuotteeseen. Parhaat tulokset saadaan, kun käytettävyystestaus integroidaan osaksi jatkuvaa kehitysprosessia.

On myös tärkeää muistaa, että käytettävyystestauksen pitäisi tapahtua mahdollisuuksien mukaan realistisessa ympäristössä ja käyttökontekstissa, jotta saadaan mahdollisimman luotettavat tulokset. Käyttäjien erilaiset taustat, taidot ja tarpeet tulisi myös ottaa huomioon testaajien valinnassa, jotta saadaan kattava kuva ohjelmiston käytettävyydestä.

4.22 Teknisen velan hallinta

Käsittelimme teknisen velan käsitettä yleisellä tasolla jo aiemmin, luvussa 3.5. Tässä luvussa keskitymme yksityiskohtaisemmin siihen mistä teknisessä velassa ja sen muodostumisessa oikeastaan on kysymys, ja miten teknisen velan hallinnointi kannattaa toteuttaa.

Teknisestä velasta puhuttaessa tulee ymmärtää se, ehkä hivenen paradoksaalinen ilmiö, että kuten rahallista velkaa myös teknistä velkaa voidaan "ottaa" tarkoituksella, ja että se voi olla joissakin tilanteissa jopa positiivinen ilmiö. Esimerkki tämänkaltaisesta velanotosta on vaikkapa se, jos keskitytään saamaan kaikki toiminnalliset vaatimukset valmiiksi tietyssä ajassa, mutta sen seurauksena katsotaan laatuvaatimuksia hivenen läpi sormien. Teknistä velkaa kuitenkin kertyy myös tahattomasti, pääosin syistä joita käsittelimme jo aiemmin. Nämä syyt voidaan pitkälti kiteyttää siihen että tehdään päätöksiä jotka eivät perustu oikeaan tietoon, esim. siten että vaatimusmäärittelyt eivät ole loppuun asti hiottuja siinä vaiheessa kun ohjelmistoa aletaan toteuttamaan. Toinen syy teknisen velan tahattomaan kertymiseen ovat ympäristötekijät, eli käytännössä se miten ohjelmistoekosysteemit kehittyvät ja tämän vuoksi esimerkiksi ohjelmointikielen versio saattaa "vanhentua" uudempien versioiden tuodessa uusia ja tehokkaampia ominaisuuksia kieleen.

Rahallisen velan käsitteistöä voidaan käyttää myös teknisen velan ominaisuuksien kuvailemiseen.

- **Pääoma:** Pääoma on teknisen velan näkökulmasta se määrä työtä, joka vaaditaan siihen että järjestelmästä saadaan optimaalisesti toimiva. Toisinsanoen se on alkuperäinen, täydellisestä tilanteesta poikkeava toteutus.

- **Korko:** Teknisen velan näkökulmasta korko ovat ne kaikki negatiiviset ilmiöt ja vaikutukset jotka kasvavat jatkuvasti kun aikaa kuluu alkuperäisen toteutuksen käyttöönotosta.

- **Koron realisoitumistodennäköisyys:** Tällä käsitteellä tarkoitetaan sitä todennäköisyyttä, jolla joudutaan tilanteeseen jossa epäoptimaalisen ratkaisun vaikutukset heijastuvat laajemmin koko järjestelmään jonka osana velkaa kerryttänyt ohjelmisto toimii.

4.23 Pidä huolta dokumentaatiosta ja käyttöohjeista

Kuten on sanottu jo aiemmin alaluvussa 4.10, on ohjelmiston dokumentointi ensiarvoisen tärkeää niin varsinaisen kehitystyön sujuvuuden, kuin myöhempien päivitysoperaatioiden kannalta. Huonosti dokumentoitu koodi voi olla kryptistä, jopa lähes mahdotonta ymmärtää jälkeenpäin, erityisesti niille kehittäjille jotka eivät ole juuri sitä koodia kirjoittaneet. Se voi olla sitä kirjoittajalleenkin, jos saman koodin äärelle palaa vasta jonkin ajan kuluttua, työskenneltyään välillä muissa projekteissa. Tästä syystä kommentointikäytäntöjen yhtenäisyys on oleellista, niin että kaikki kehitystyötä tekevän organisaation kehittäjät sitoutuvat käyttämään ilmaisutapoja, jotka ovat selkeästi ymmärrettävissä muille. Kommentoinnin avuksi on myös kehitetty erilaisia automaattisia järjestelmiä, jotka pystyvät jossain määrin generoimaan kommentteja koodin pohjalta ilman että kehittäjän täytyy käyttää niihin aikaa.

Toinen oleellinen kirjallinen puoli joka liittyy ohjelmistokehitykseen on käyttöohjeiden kirjoittaminen. Nämä on tarkoitettu ensisijaisesti ohjelmiston loppukäyttäjille, mistä syystä niissä esitetty teknisten yksityiskohtien tarkkuus ja käytetty kieliasu tulee suunnitella sen mukaan, minkälaista ohjelmistoa ollaan tekemässä ja minkälaiset ihmiset näitä ohjeita todennäköisesti tulevat lukemaan. Oleellista käyttöohjeissa on se että ne toimivat helposti ymmärrettävänä referenssinä käyttäjille tilanteissa joissa tapahtuu jotain odottamatonta tai käytetään jotain sellaista ohjelmiston piirrettä jonka aktivaatioaste on vähäinen, ts. jolloin käyttäjillä ei välttämättä ole aiempaa kokemusta tai selkeää muistikuvaa siitä miten juuri kyseistä ominaisuutta tulisi käyttää. Käyttöohjeiden tulee kuitenkin kattaa myös järjestelmän yleisetkin ominaisuudet, jotta niitä voidaan käyttää uusien käyttäjien kouluttamisessa.

Käyttöohjeiden suhteen tulee muistaa mahdolliset käännökset muille kielille, jotka määräytyvät pitkälti ohjelmiston asiakasorganisaation tarpeiden mukaan. Koska varsinkin teknisten termien kohdalla on suuri mahdollisuus virheisiin jos käännöstyötä tekee joku jolla ei ole aiempaa kokemusta käyttöohjeiden kääntämisestä, on tämä osuus työstä mahdollisesti järkevintä ulkoistaa sellaisille käännöspalveluja tarjoaville yrityksille jotka ovat erikoistuneita teknisten dokumenttien kääntämiseen. Nykyaikana erilaiset tekoälyt voivat myös kääntää tekstejä melko hyvin, joskaan eivät täydellisesti, mistä syystä ihmiskääntäjiä tarvitaan vielä vähintäänkin vir-

heenkorjaukseen ja sisältöjen tarkastamiseen, varsinkin mitä teknisemmästä materiaalista on kyse.

Ohjelmiston itsensä osalta kääntämistyötä tarvitaan usein lokalisaation, eli sisällön ja käyttöliittymän kääntämisen jollekin tietylle kielelle, kanssa. Vaikkakin englanti on muodostunut maailman yleiskieleksi ja enemmistö ohjelmistojen käyttäjistä pärjää englanninkielisten sisältöje ja valikkojen kanssa, on lokalisaatio silti merkittävä osa ohjelmistokehitystä nykyaikana ja sen suosio on ollut jatkuvassa kasvussa viime vuosikymmenenen aikana. Lokalisaation käytännön toteuttaminen riippuu ohjelmasta, ja voi vaatia sitä että mekanismit joilla kielen vaihtaminen tapahtuu vaativat kehittäjien työtä, mutta monille viitekehyksille ja sisällönhallintajärjestelmille on myös olemassa valmiita lokalisointikirjastoja ja työkaluja joita käyttäessä lokalisaatio ei oikeastaan vaadi muuta kuin sen että käännetyt sisällöt ja valikkojen tekstit ovat jossain tietokannassa. Tämäkin käännöstyö tehdään jo nyt ja tullaan todennäköisesti tulevaisuudessa enenevässä määrin tekemään tekoälyjen avulla, niin että tekstejä kääntävien ihmisten määrä vähenee huomattavasti.

4.24 Tukijärjestelmät ja -toimet

Ohjelmistot eivät elä tyhjiössä, vaan ekosysteemissä jossa ne ovat riippuvaisia lukuisista muista ohjelmista, jotka tukevat niiden toimintaa. Tällaisia muita ohjelmia ovat erilaiset markkinointikanavat, asiakastuki- ja palautejärjestelmät, lisenssinhallintaan tarkoitetut järjestelmät ja niin edelleen. Vaikka loppukäyttäjän näkökulmasta nämä usein vaikuttavatkin olevan osa varsinaista ohjelmistoa, ovat ne todellisuudessa useimmiten omia ohjelmistojaan, jotka kommunikoivat varsinaisen ohjelmistotuotteen kanssa.

Näiden tukijärjestelmien kehittäminen on usein oma projektinsa, joka asiakkaan tarpeiden mukaan voi tapahtua joko samanaikaisesti varsinaisen ohjelmiston kehityksen kanssa, tai vasta sen jälkeen kun pääasiallinen ohjelmistotuote on jo julkaistu. On olemassa myös valmiita ratkaisuja joilla näitä toimintoja voidaan toteuttaa, esimerkiksi monesti verkkosivuilla nähtävät palautelaatikot, käyttäjille suunnatut kyselyt, chattiohjelmat ja muut vastaavat ovat kolmansien osapuolien tarjoamia palveluita. Valmiit ratkaisut eivät kuitenkaan aina ole soveltuvia juuri haluttuun käyttötarkoitukseen ja silloinkin kun ne ovat ne vaativat oman konfiguraa-

tionsa. Useimmiten tämä on kuitenkin huomattavasti pienempi työ kuin näiden ohjelmistojen kehittäminen alusta lähtien itse.

4.25 Käyttäjien koulutus

Siinä vaiheessa kun ohjelmisto on valmis käytettäväksi, mieluusti ennen kuin se laitetaan lopulliseen tuotantoon, pitää sen tuleville käyttäjille pitää perehdytyksiä, joissa ohjelmiston toiminta selitetään hyvin yksityiskohtaisesti läpi ja suorastaan kädestä pitäen opetetaan loppukäyttäjille miten ohjelmaa tulee käyttää oikein, mitä tehdään virhetilanteissa ja mihin toimintoihin kullakin käyttäjäryhmällä on pääsy. Tässä vaiheessa kannattaa olla hyvin tehdyt kirjalliset ohjeet myös valmiina, mutta niiden varaan asiaa ei voi jättää, vaan koulutus pitää käytännössä tehdä kokousmuotoisesti niin että asiakkaan edustajat voivat esittää selventäviä kysymyksiä.

Tämä tietysti koskee vain sellaisia ohjelmistoja jotka on räätälöity jonkun tietyn asiakkaan tarpeita vaativiksi. Jos tehdään pelejä tai verkkosivuja tai ylipäänsä jotain minkä ihmiset voivat ostaa itselleen valmiina tuotteena, koulutuksen hoitaa jonkinlainen tutoriaali tai nettisivujen kohdalla ihan yleinen käyttöliittymän tuttuus. Kuitenkin, jos nettisivujen kautta on tarkoitus käyttää jotain muuta järjestelmää joka on tehty asiakkaan vaatimusten mukaan, on koulutus tällöin melkolailla pakollinen.

Tulee huomioida että myös käyttäjien kouluttaminen on laskutettavaa työtä; mitä enemmän sitä pitää tehdä, sitä enemmän siitä pitää laskuttaa. Jos asiakas ei ole halukas tai kykenevä budjetoimaan tarpeeksi rahaa tähän osaan projektia, on järkevin tapa koota asiakasorganisaatiosta avainasemassa olevia henkilöitä, joilla optimaalisesti sellaisia joilla on keskimääräistä paremmat tietotekniset taidot, joille järjestelmän toiminta opetetaan. Nämä voivat sitten kouluttaa muita oman organisaationsa käyttäjiä ohjelmiston toimintaan.

4.26 Julkaisu

Peliteollisuudessa on tapana sanoa, että “on vain yksi julkaisupäivämäärä”. Tällä tarkoitetaan sitä että julkaisun toimivuuden vaikutus ohjelmiston tulevalle menestykselle on niin merkittävä, että sen yhden ja ainoan päivän kun ohjelmisto julkaistaan täytyy onnistua niin hyvin kuin on mahdollista. Serverit eivät saa kaatua, bugit eivät saa hyppiä silmille, ihmisten pitää päästä

kirjautumaan palveluun ja luomaan käyttäjätunnuksia ja niin edelleen. Kaikista aiemmissa luvuissa käsitellyistä työ- ja testausvaiheista huolimatta todellisuus on kuitenkin se, että mikään ohjelmisto ei ole täydellinen siinä vaiheessa kun se vihdoin julkaistaan loppukäyttäjille; oleellista onkin se että se vaikuttaa täydelliseltä, minkä varmistamiseen pitää varata runsaasti työvoimaa ja resursseja sitä silmällä pitäen että kaikki mahdolliset ongelmatilanteet saadaan ratkottua lennosta jos ja kun seinät alkavat kaatua päälle maaliviivalla.

Julkaisun saaminen toimimaan vaatii koordinointia markkinointi- ja myyntiosastojen kanssa, mikäli ollaan rakennettu kaupalliseen myyntiin menevää ohjelmistotuotetta, tai vastaavasti sen organisaation jolle ohjelmisto on rakennettu mikäli kyse on järjestelmästä joka on räätälöity tietyn asiakasorganisaation tarpeisiin.

5 Ohjelmistotuotteen ja -palvelun ylläpito

Valmiin ohjelmiston elinkaaren loppuvaihetta kutsutaan ylläpitovaiheeksi. Ylläpitovaihe voi kestää useita, jopa kymmeniä vuosia, joten ohjelmistokehittäjän työ ei suinkaan lopu ohjelmiston valmistumiseen. On hyvä huomioida, että ylläpidon osuus ohjelmistokehityksen kokonaiskustannuksista ja ajasta on usein huomattavan suuri. On arvioitu, että jopa 80 prosenttia kokonaiskustannuksista ja ajasta kuluu ylläpitotehtäviin.

Ketterissä projekteissa on usein parempi puhua ylläpidon sijasta jatkuvan kehittämisen vaiheesta, joka kuvaa paremmin nykyaikaisen ohjelmistokehityksen luonnetta. Jatkuvan kehittämisen etuna on uusien ominaisuuksien, korjausten ja muutosten tekeminen pienemmissä paloissa, jolloin säästytään laajojen yksittäisten ohjelmistopäivitysten aiheuttamilta ongelmilta. Ajan saatossa muuttuviin vaatimuksiin ja teknologisten ympäristöjen muutoksiin voidaan reagoida paremmin, kun päivittäminen on jatkuvaluonteista.

5.1 Järjestelmän julkaisu

Järjestelmän julkaisu on vaiheittain etenevä prosessi. Vaiheistuksen tärkeimpänä tavoitteena on mahdollistaa aikainen virheiden tunnistaminen ja korjaaminen, sekä ohjelmiston toiminnallisuuksien testaus. Lopullista julkaisua edeltäviä kehitysvaiheita/kypsyystasoja kutsutaan yleensä alphaksi ja betaksi. Alpha-julkaisu rajoittuu yleensä ohjelmiston kehittäjiin ja sisäisiin testaajiin. Tässä vaiheessa ohjelmisto voi olla epävakaa ja sisältää vain osan suunnitelluista ominaisuuksista. Kehittäjät käyttävät tätä vaihetta pääasiassa suurimpien bugien ja suorituskykyongelmien korjaamiseen.

Beta-julkaisuvaiheessa hyödynnetään ulkopuolisia testaajia, kuten beta-testaajia tai ennakkokäyttäjiä. Joissakin tapauksissa, erityisesti ketterässä kehityksessä, voi olla useita betaversioita tai vaihtoehtoisia testaus- ja julkaisusyklejä. Beta-testauksessa ohjelmisto on lähempä-

nä lopullista versiota ja sen tavoitteena on kerätä laajempaa palautetta ohjelmiston toiminnallisuuksista, käyttäjäkokemuksesta ja mahdollisista bugeista. Beta-vaiheessa ohjelmistoon voidaan vielä tehdä merkittäviä muutoksia. Beta-julkaisu on yleensä vakaa, mutta se saattaa sisältää joitakin tunnistamattomia virheitä.

Lopullinen julkaisu merkitsee ohjelmiston siirtymistä kehitys- ja testausvaiheesta tuotantokäyttöön. Ennen lopullista versiota julkaisuprosessia voidaan vielä vaiheistaa käyttämällä julkaisuehdokasta (engl. *release candidate*). Lopullisessa julkaisussa tiedetyt virheet ja ongelmat on korjattu ja ohjelmiston uskotaan olevan riittävän vakaa ja toimiva laajan yleisön käyttöön. Kehittäjät voivat edelleen päivittää ohjelmistoa uusilla ominaisuuksilla tai korjata ilmeneviä ongelmia, mutta lopullisen julkaisun katsotaan täyttävän kaikki keskeiset vaatimukset.

5.2 Datan keruu

Ohjelmistoista ja niiden käytöstä voidaan kerätä monenlaista tietoa. Suuret tietovarannot ilman jäsentelyä eivät ole kovin arvokkaita, mutta kerätyn datan analysoinnilla voidaan tuottaa mielekästä tietoa niin järjestelmän toiminnasta kuin sen tuottamasta liiketoimintahyödystäkin. Parhaimmillaan ohjelmiston tuottama tieto voi mahdollistaa organisaation tietopohjaista päätöksentekoa ja vahvistaa datan ja tietojohtamisen välistä vuorovaikutussuhdetta.

Ohjelmiston julkaisun jälkeinen seuranta keskittyy ohjelmiston suorituskyvyn, käytettävyyden ja turvallisuuden varmistamiseen. Bugien seurantajärjestelmät mahdollistavat virheiden raportoinnin, priorisoinnin ja seurannan. Käyttäjäpalautteen kerääminen ja analysointi taas antaa tietoa ohjelmiston käytettävyydestä ja mahdollisista parannuskohteista.

Huomiolaatikko: IT-infrastruktuurin seurantaa, suorituskyvyn analysointia ja verkon valvontaa voidaan automatisoida käyttämällä tarkoitusta varten kehitettyjä työkaluja. Jokaisella työkalulla on omat erityispiirteensä ja ne sopivat erilaisiin käyttöympäristöihin ja vaatimuksiin. Eräs esimerkki yleisesti käytössä olevasta seurantaohjelmistosta on Grafana, joka on avoimen lähdekoodin ohjelmisto pääasiassa aikasarjatietojen visualisointiin ja hälytyksiin. Ohjelmisto mahdollistaa useiden tietolähteiden (mm. Prometheus, InfluxDB, Elasticsearch) datan tuonnin ja visualisoinnin reaaliaikaisesti. Sen avulla käyttäjät voivat luoda erilaisia näkymiä järjestelmätiedoista kaavioina, graafeina ja taulukoina.

Huomiolaatikko: Huomiothan myös ylläpitovaiheessa tietojen keräämiseen liittyvät lakitekniset seikat. Datan keräämisessä tulee huomioida tietosuoja-asetukset, kuten EU:n yleinen tietosuoja-asetus (GDPR) tai vastaavat kansalliset lait eri maissa. Tietojen keräämisen tulee olla perusteltua ja sen tarkoitus selkeästi määritelty. Pseudonymisoituja lokitietoja voidaan edelleen käyttää suorituskyvyn seurantaan, käyttäytymisanalyysiin ja järjestelmän parannusten suunnitteluun ilman, että yksittäisten käyttäjien yksityisyyttä loukataan.

5.3 Monitorointi

Järjestelmän teknisen toimintakyvyn monitorointi on keskeinen osa ylläpitovaihetta. Monitoroinnin käytännön vaiheet voidaan esittää vaikkapa seuraavanlaisena tehtävälistana:

1. **Suunnittelu ja konfigurointi:** Määrittele seurattavat metriikat ja tapahtumat: Tämä voi sisältää mm. palvelimen resursseja (CPU, muisti, levytila), verkon suorituskykyä, sovellusten vasteaikoja ja tietokantakyselyjen tehokkuutta. Aseta hälytyskynnykset ja määrittele toimintaprosessit hälytysten sattuessa.

2. **Tietojen keräys ja aggregointi:** Kerää metriikoita ja lokitietoja järjestelmän eri komponenteista. Aggregoi data keskitettyyn monitorointijärjestelmään analyysin ja kokonaiskuvan muodostamiseksi.

3. **Analysointi ja visualisointi:** Analysoi kerättyä dataa trendien, poikkeamien ja potentiaalisten ongelmien tunnistamiseksi. Käytä visualisointityökaluja, kuten kojelautoja ja kaavioita esittämään data ymmärrettävässä muodossa.

4. **Hälytykset ja reagointi:** Määritä hälytysjärjestelmä lähettämään automaattisia ilmoituksia, kun kriittiset kynnykset ylittyvät tai tunnistetaan poikkeamia. Organisoi toimenpiteet hälytysten hallintaan, mukaan lukien automaattiset korjaustoimenpiteet tai manuaalinen puuttuminen.

5. **Jatkuvat parannukset:** Hyödynnä monitoroinnista saatuja tietoja järjestelmän suorituskyvyn ja vakauden jatkuvan parantamisen pohjana. Päivitä monitorointistrategiaa vastaamaan järjestelmän muutoksia ja uusia vaatimuksia.

5.4 Ekosysteemin seuranta

Ekosysteemin seurannalla tarkoitetaan järjestelmän hyödyntämien taustateknologioiden kehityksen ajantasaista seurantaa. Migraatioiden valmisteleminen ja toteuttaminen sekä esim. kolmansien osapuolien komponenttien päivitykset ovat keskeinen osa tuotteenhallintaa. (Tai muuten käy näin: How one programmer broke the internet by deleting a tiny piece of code)

5.5 Ylläpidä tietoturvaa

Ohjelmistotuotteet koostuvat useista erilaisista komponenteista, joiden monimutkaisuusaste vaihtelee. Erityisesti isommissa projekteissa kaikki komponentit eivät välttämättä ole omaa työtä, vaan osa komponenteista tuotetaan alihankintana, ostetaan tai hyödynnetään vapaasti käytettäviä avoimen lähdekoodin tuotteita. Tietoturvan kannalta on keskeistä tuntea omassa tuotteessa käytössä olevat komponentit. Komponentit muodostavat tuoterakenteen, jossa on huomioitava useita tietoturvaseikkoja. Jotkin komponentit ovat turvallisuuskriittisempiä kuin toiset ja joihinkin komponentteihin kiinnitetään turvallisuuden kannalta enemmän huomioita. Valitettavasti myös turvalliseksi mielletyissä ohjelmissa ja kirjastoissa voi olla kohtalokkaita haavoittuvuuksia, joita peruskäyttäjä ei välttämättä osaa ottaa huomioon. On järkevää kohdella jokaista komponenttia turvallisuuden kannalta kriittisenä. Tietoturvanäkökulmasta ohjelmisto kannattaa pitää mahdollisimman yksinkertaisena ja poistaa sen toiminnan kannalta toisarvoiset komponentit kokonaan käytöstä. Myös asiakkaita tulee opastaa tuotteen turvalliseen elinkaarenhallintaan. [23]

Komponenttien turvallisuutta voidaan arvioida esimerkiksi Kyberturvallisuuskeskuksen laatiman tarkastuslistan mukaan [23]:

- Komponentit hyödyntävät eri teknologioita ja alustoja. Ollaanko tietoisia niiden vaikutuksista tuotteen turvallisuuteen?
- Eri komponenteille on määritettävä erilaiset käyttöoikeudet. Kuinka komponentteihin voidaan soveltaa mahdollisimman suppeiden valtuuksien periaatetta?

- Eri komponenteilla on eripituiset päivitysvälit. Kuinka komponenttien päivitykset synkronoidaan tuotteen päivitysten kanssa?

- Komponenttien toiminnallisuudet voivat poiketa paljonkin tuotteen ydintoiminnallisuudesta. Tarvitaanko komponentteja varten erillisiä testausmenetelmiä ja muita laadunvarmistustoimia?

5.6 Korjaa ohjelmointivirheet hallitusti

Ohjelmointivirheiden vakavuus voi vaihdella järjestelmän käytön kokonaan estävästä virheestä käyttäjää ärsyttävään, kosmeettiseen yksityiskohtaan. On myös arvioitu, että n. 5 prosettia ohjelmavirheistä jää kokonaan havaitsematta, ja ohjelma toimii näistä huolimatta [4]. Ohjelmointivirheiden korjaamista kannattaa lähestyä systemaattisesti ja hallitusti. Ensimmäisenä selvitetään, mistä ja miksi virhe on syntynyt. Korjausten ja muutosten jälkeen on testattava, että korjaustoimenpiteet eivät aiheuta uusia ongelmia. Korjaukset siirretään tuotantoympäristöön kontrolloidusti, minimoiden mahdollisuuksien mukaan käyttökatkot ja muut häiriöt käyttäjille.

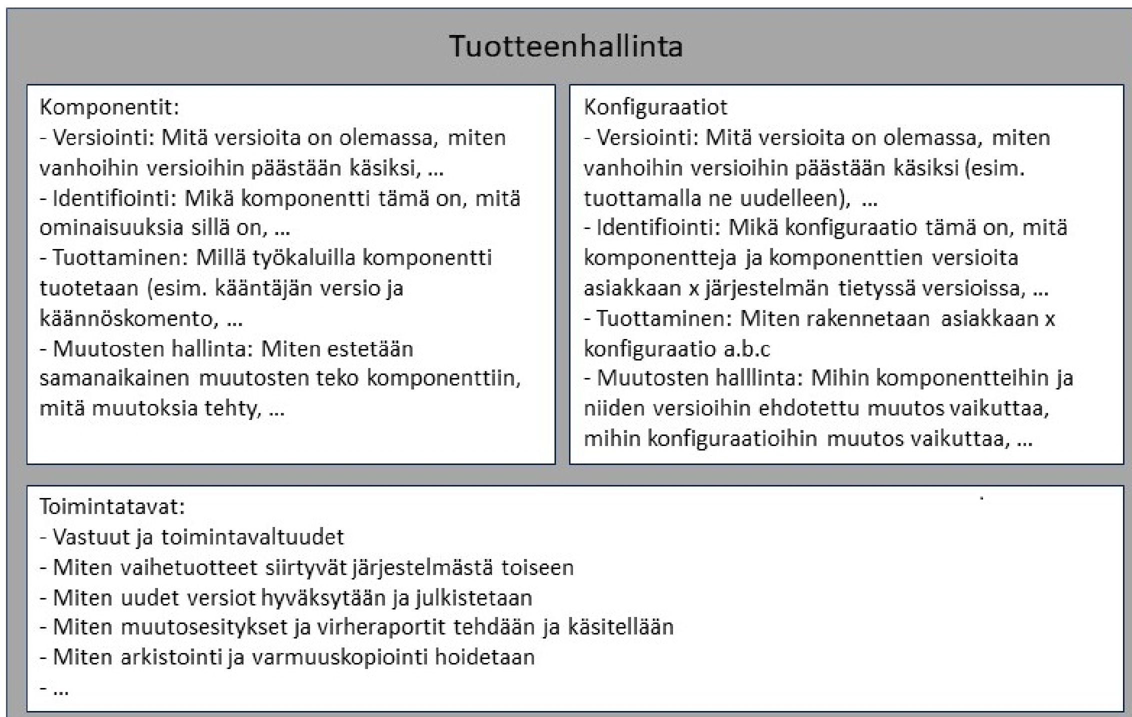
5.7 Elinkaaren aikainen päivittäminen

Ohjelmiston komponentit ja konfiguraatiot muuttuvat ja kehittyvät ohjelmiston elinkaaren aikana, kun virheitä korjataan, uusia ominaisuuksia lisätään ja komponenteista tehdään uusia variaatioita. Joissain tapauksissa voi olla tarpeen jatkokehittää myös komponenttien ja konfiguraatioiden vanhoja versioita, joiden tulee olla saatavilla tarvittaessa. Ylläpidon aikaiseen päivittämiseen liittyviä keskeisiä asioita on selvittää muutosten vaikutukset kokonaisuuteen, mikä tarkoittaa, että uutta toiminnallisuutta kannattaa lisätä hallitusti. Jatkuvan integroinnin kautta voidaan pakottaa muutoksen niin pieniksi, että versionhallinnassa oleva kokonaisuus on aina valmis suoritettavaksi. Teknisestä näkökulmasta ohjelmistotuotteen hallinta on komponenttien ja konfiguraatioiden hallintaa, joka voidaan jakaa kolmeen erilaiseen osa-alueeseen (kuvio 5.1 sivulla 95) [4].

- menetelmät, joilla saman komponentin eri versiot hallitaan

- menetelmät, joilla konfiguraatioita ja niiden versioita hallitaan

- versioita ja konfiguraatioita luotaessa ja muutettaessa noudetattavat toiminta-tavat



Kuva 5.1: Tuotteenhallinnan osa-alueet [4]

5.8 Jatkuva testaaminen

Jatkuva testaus on osa CI/CD-prosessia. Ylläpitovaiheessa jatkuva testaus keskittyy erityisesti olemassa olevan ohjelmiston toimintavarmuuden varmistamiseen, uusien virheiden havaitsemiseen ja korjaamiseen sekä järjestelmän yhteensopivuuden säilyttämiseen uusien päivitysten ja muutosten yhteydessä. On hyvä huomioida, että mitä korkeammalla V-mallin testaustasolla ollaan, sitä kalliimpaa virheiden korjaus on. Jos esimerkiksi järjestelmätestauksessa havaitaan ongelma, voi tämän virheen korjaus vaikuttaa useisiin komponentteihin. Ylemmän tason virheenkorjausta vaikeuttaa se, että virheiden korjaus voi aiheuttaa muita virheitä. Tämän takia jatkuva testaus on tärkeää.

5.9 Päivitä käyttäjää häiritsemättä

Jokainen projekti määrittelee omat parhaat käytäntönsä päivitysten ajamiselle ja näitä voidaan säätää projektin edetessä. Tärkeää on suunnitella ja hallita päivitykset niin, että niistä

on mahdollisimman vähän haittaa käyttäjälle. Käytännössä päivityksiä kannattaa tehdä esimerkiksi öisin tai muuten toimistoaikojen ulkopuolella aikaerot huomioiden. Lue elinkaaren aikaisesta päivittämisestä tarkemmin alaluvuista 5.6 ja 5.7.

5.10 Ylläpito

Ylläpitäminen voi tarkoittaa esimerkiksi eri alustoilla toimivien sovellusten päivittämistä sekä taustajärjestelmien, kuten tietokantojen ja palvelimien hallintaa. Järjestelmän ylläpitoon kuuluu myös varmuuskopioinnista huolehtiminen, häiriöiden ennaltaehkäisy ja käytön aikana havaittuihin muutospyyntöihin reagoiminen. Elinkaaren aikana tulee kiinnittää erityistä huomiota tietoturvaympäristössä tapahtuviin muutoksiin ja huolehtia, että ohjelmiston tietoturva pysyy ajan tasalla. Ylläpitoon kuuluu olennaisena osana myös dokumentaation ja tukimateriaalien ajantasaisuus, sekä järjestelmän käyttäjiin vaikuttavista muutoksista tiedottaminen.

5.11 Elinkaaren päätös

Ohjelmiston elinkaaren hallintaan kuuluu olennaisena osana ohjelmiston hallittu alasajo (ks. myös luku 3.5). Järjestelmällisellä alasajoprosessilla pyritään minimoimaan mahdolliset negatiiviset vaikutukset asiakkaan toimintaan ja tietoturvaan. Alasajon ensimmäinen vaihe on tunnistaa elinkaarensa päässä olevat palvelut. Syitä alasajolle voivat olla esimerkiksi alentunut käyttöaste, korkeat ylläpitokustannukset, vanhentunut teknologia tai asiakkaan muuttuneet liiketoimintavaatimukset.

Kun palvelu on tunnistettu alasajettavaksi, seuraava vaihe on suunnitella alasajon käytännön toteutus. Tähän sisältyy yksityiskohtainen suunnitelma siitä, miten palvelu poistetaan käytöstä mahdollisimman vähällä häiriöllä. Suunnitelman tulee sisältää prosessin aikataulu, tehtävät ja vastuuhenkilöt. On tärkeää myös varmistaa, että kaikki sidosryhmät ovat tietoisia suunnitelmasta ja sen vaikutuksista.

Alasajoprosessin toimenpiteet riippuvat järjestelmästä. Ne voivat pitää sisällään mm. datan siirtämisen tai arkistoinnin, riippuvuuksien hallinnan ja mahdollisten korvaavien palveluiden määrittelyn. Tietoturva on syytä huomioida, sillä usein tuen, korjausten ja päivitysten loppuminen aiheuttaa tietoturva-aukkoja, jotka voivat jättää ohjelmiston haavoittuvaksi. Myös

arkaluonteisten tietojen siirtäminen järjestelmästä toiseen vaatii huolellisuutta. Lisäksi on hyvä varmistaa, että kaikki lisensointiin ja sopimuksiin liittyvät seikat hoidetaan asianmukaisesti alasajon myötä. Ohjelmiston käytöstäpoiston vaiheet dokumentoidaan osana elinkaaren hallinnan prosessikäytänteitä.

Elinkaaren päätökseen ja tietoturvaan liittyviä tapausesimerkkejä:

Kyberturvallisuuskeskus: Käytöstä poistuvien palveluiden alasajo tulee tehdä huolella

6 M.STUP -menetelmäopas

6.1 M.STUP -menetelmäopas

Tutustu. Käytä. Omaksu.

> ”M.Stup on opiskelijaprojekteja varten suunniteltu ketterä ohjelmistokehitysmenetelmä, joka pyrkii noudattamaan ketterän ohjelmistokehityksen julistuksen arvoja sekä periaatteita. Ajatuksena on tuoda ryhtiä ja rakennetta pienen ryhmän tai jopa yksittäisen opiskelijan projektityön suorittamiseen pitäen kuitenkin menetelmän aiheuttaman ylimääräisen kuorman mahdollisimman pienenä. Menetelmä noudattelee hyvin löyhästi suositun SCRUM-ohjelmistokehitysmenetelmän ajatuksia ja yleistä toimintarakennetta. Lisäksi menetelmä olettaa tiimin käyttävän GIT-versionhallintajärjestelmää ja GitLab-projektialustaa sitoen työmenetelmät konkreettisesti näihin tiimityökaluihin.”

Lähdeluettelo

[1] H. W. Rittel ja M. M. Webber, ”Dilemmas in a general theory of planning”, *Policy sciences*, vol. 4, nro 2, s. 155–169, 1973.

[2] J. F. Dooley, *Software Development, Design and Coding.* Apress, 2017. doi: `10.1007/978-1-4842-3153-1`.

[3] J. F. Dooley, *Software Design Problems: Wicked or Tame?*, 2018. viitattu 18. kesäkuuta 2024. url: `https://www.apress.com/gp/blog/all-blog-posts/software-design-problems-wicked-or-tame/15558942`.

[4] I. Haikala ja T. Mikkonen, *Ohjelmistotuotannon käytännöt.* Talentum Media Oy, 2011, isbn: 978-952-14-1754-2.

[5] J. Desjardins, *How many millions of lines of code does it take?*, 2017. url: `https://www.visualcapitalist.com/millions-lines-of-code/`.

[6] O. Celkee Tivia, *Tietojärjestelmien hankinta Suomessa 2013.* Tivia, 2013.

[7] W. W. Royce, ”Managing the development of large software systems: concepts and techniques”, teoksessa *Proceedings of the 9th international conference on Software Engineering*, 1987, s. 328–338.

[8] T. Mäkilä, *Software development process modeling. Developers perspective to contemporary modeling techniques.* Turku: TUCS Dissertations No 148, 2012, isbn: 978-952-12-2790-5.

[9] S. E. Biable, N. M. Garcia, D. Midekso ja N. Pombo, ”Ethical Issues in Software Requirements Engineering”, *Software*, vol. 1, nro 1, s. 31–52, 2022, issn: 2674-113X. doi: `10.3390/software1010003`. url: `https://www.mdpi.com/2674-113X/1/1/3`.

[10] S. Vallor, A. Narayanan, B. Regnell, C. Jones ja R. Skipper, ”An Introduction to Software Engineering Ethics”, teoksessa *Applied Ethics*. Santa Clara, CA, USA: Santa Clara University, 2015, s. 1–60.

[11] B. Cook. ”Engineering: Everything You Need to Know About Software Engineering Ethics”, viitattu 17. huhtikuuta 2023. url: `https://fellow.app/blog/engineering/engineering-everything-you-need-to-know-about-software-engineering-ethics/`.

[12] F. Scale, *Software Developer Ethics: Avoiding Ethical Issues in Software Development*, Full Scale, 2024. viitattu 18. kesäkuuta 2024. url: https://fullscale.io/blog/ethical-issues-in-software-development/.

[13] M. Murgia. ”Google Pauses AI Image Generation of People After Diversity Backlash”, viitattu 17. huhtikuuta 2024. url: https://www.ft.com/content/979fe974-2902-4d78-8243-a0cff68e630a.

[14] A. C. M. de Souza, ”Social Sustainability Approaches for a Sustainable Software Product”, *ACM SIGSOFT Software Engineering Notes*, vol. 48, s. 38–43, 2023. DOI: 10.1145/3573074.3573085.

[15] L. Lannelongue, J. Grealey ja M. Inouye, ”Green Algorithms: Quantifying the Carbon Footprint of Computation”, *Advanced Science*, vol. 8, nro 12, s. 2 100 707, 2021. DOI: https://doi.org/10.1002/advs.202100707. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202100707. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/advs.202100707.

[16] L. A. Wright, S. Kemp ja I. Williams, ”'Carbon footprinting': towards a universally accepted definition”, *Carbon Management*, vol. 2, nro 1, s. 61–72, 2011. DOI: 10.4155/cmt.10.39. url: https://doi.org/10.4155/cmt.10.39.

[17] M. Z. Hauschild, R. K. Rosenbaum ja S. I. Olsen, *Life cycle assessment*. Springer, 2018.

[18] G. G. Protocol, ”Greenhouse gas protocol”, *Sector Toolsets for Iron and Steel-Guidance Document*, 2011.

[19] Apriorit, *The Importance of Accessibility & Inclusiveness in UI/UX Design*, https://www.apriorit.com/dev-blog/design-accessibility-in-ui-ux, 2023. viitattu 18. kesäkuuta 2024.

[20] WCAG. ”WCAG 101: Understanding the Web Content Accessibility Guidelines”, viitattu 18. kesäkuuta 2024. url: https://wcag.com/resource/what-is-wcag/.

[21] E. B.-N. Sanders ja P. J. Stappers, ”Co-creation and the new landscapes of design”, *Co-design*, vol. 4, nro 1, s. 5–18, 2008.

[22] M. McHugh, F. McCaffery ja V. Casey, ”Barriers to Adopting Agile Practices When Developing Medical Device Software”, teoksessa *Software Process Improvement and Capability Determination*, A. Mas, A. Mesquida, T. Rout, R. V. O'Connor ja A. Dorling, toim., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 141–147, ISBN: 978-3-642-30439-2.

[23] Kyberturvallisuuskeskus. ”Turvallinen tuotekehitys: Kohti hyväksyntää”, viitattu 18. kesäkuuta 2024. url: https://www.kyberturvallisuuskeskus.fi/sites/default/files/media/publication/Turvallinen_tuotekehitys_Suomi_J003_2018.pdf.