Elias Wuoti

OHJELMOIJAN TYÖTEHOKKUUDEN MITTAUS

Informaatioteknologian ja viestinnän tiedekunta

Kandidaatintyö

Toukokuu 2019

TIIVISTELMÄ

Elias Wuoti: Ohjelmoijan työtehokkuuden mittaus
Kandidaatintyö
Tampereen yliopisto
Tietotekniikka
Huhtikuu 2019

Tässä työssä käydään läpi ohjelmoijan työtehokkuuden määritelmää ja esitellään keinoja seurata sitä. Työssä esiteltäviä keinoja ovat biologinen seuranta, ohjelmakoodin analysointi sekä seuranta käyttöjärjestelmästä. Eri keinoista esitellään niiden hyvät ja huonot puolet, minkä perusteella hahmotellaan paras optimaalinen toteutus ohjelmoijan työtehokkuuden seurannalle.
Työssä tarkastellaan myös erästä käytännön toteutusta työtehokkuuden seurannasta. Tämän ohjelman rakennetta ja toimintaa arvioidaan työssä ohjelmoijan työtehokkuuden seuraamisen kannalta. Toteutuksesta arvioidaan sen hyviä ja huonoja puolia, minkä jälkeen sitä verrataan muihin menetelmiin työtehokkuuden seurannalle. Tämän vertailun perusteella taas ehdotetaan käytännön toteutukseen parannuksia.
Työssä todetaan, että paras tapa seurata ohjelmoijan työtehokkuutta on yhdistellä työssä esiteltyjä menetelmiä. Työssä kuitenkin myös todetaan, että biologisen seurannan toteuttaminen ei ole realistista käyttäjien vastahakoisuuden takia. Tämän perusteella työssä todetaan, että paras ratkaisu on yhdistää ohjelmakoodin analysointi sekä käyttöjärjestelmästä analysointi.

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. TYÖTEHOKKUUDEN MÄÄRITELMÄ 2
3. TYÖTEHOKKUUDEN MITTAAMINEN 4
3.1 Biologinen mittaaminen 4
3.2 Ohjelmakoodin analysointi 4
3.3 Mittaaminen käyttöjärjestelmästä 6
3.4 Optimaalinen ratkaisu 6
4. TYÖTEHOKKUUTTA MITTAAVA OHJELMA 8
4.1 Ohjelman rakenne 8
4.2 Työtehokkuuden laskeminen 10
4.3 Ohjelman vertailu muihin seurantamenetelmiin 12
5. YHTEENVETO 14
LÄHTEET 15

LYHENTEET JA MERKINNÄT

EEG	Elektroenkefalografia eli aivosähkökäyrämittaus
IDE	Integrated Development Environment
Xorg	X11-pohjainen ikkunointijärjestelmä
PID	Process ID, prosessin tunniste

1. JOHDANTO

Ohjelmointi on ala, joka kärsii työmäärän vaikeasta mitattavuudesta. Etenkin yksilön saavutettavat työmäärät ovat hankalia mitata: pelkästään yksinkertainen ominaisuuksien valmistumisnopeus tai pelkkä koodin määrän analysointi ei riitä, sillä nopeammin kehitetty ohjelma on käytössä usein hitaampi kuin ohjelma, jonka optimointiin on kulutettu enemmän aikaa. [1]

Työn tarkoituksena on muodostaa kokonaiskuva ohjelmoijan työtehokkuudesta ja sen seuraamisen mahdollisista toteutustavoista. Kokonaiskuva saavutetaan vertailemalla eri seurantatapojen hyviä ja huonoja puolia sekä hahmottelemalla käytännön toteutusta seurannalle.

Luvussa 2määritellään työtehokkuuden käsite yleisesti ja sovelletaan sitä ohjelmistokehitykseen. Luvussa 3 esitetään kolme eri tapaa mitata ohjelmoijan työtehokkuutta. Ensimmäinen tarkasteltava metodi on biologinen seuranta, joka perustuu ihmisen kehosta fyysisesti saatavan tiedon käsittelemiseen esimerkiksi aivosähkökäyrämittauksen avulla. Toinen tarkasteltava metodi on tuotetun ohjelmakoodin analysointi, jossa mitataan ohjelmakoodin suoritustehokkuutta ja sen tuottamiseen kulunutta aikaa. Kolmas tarkasteltava metodi on seuranta käyttöjärjestelmän kautta, jolloin mitataan käyttäjän tekemiä toimintoja, esimerkiksi hiiren liikettä tai näppäimistön painalluksia. Luvussa 4 tarkastellaan yhtä esimerkkitoteutusta työtehokkuuden mittaamisesta ohjelmallisesti.

2. TYÖTEHOKKUUDEN MÄÄRITELMÄ

Yleisesti työtehokkuus voidaan määritellä yksilön tai organisaation kykynä muuntaa lähtötietoja tai raaka-aineita hyödykkeiksi tai tuotteiksi. Yksilön tapauksessa tätä voidaan mallintaa esimerkiksi yksittäisen työntekijän tuotoksiin tunnin välein mitattuna ja laatuun suhteutettuna. [2]

Tuotantoteollisuudessa voidaan tarkastella esimerkiksi autotehtaan kokoonpanolinjaston tehokkuutta: linjaston tehokkuus riippuu nimenomaan sen tuottamien autojen määrästä. Linjastolla yksittäisen työntekijän työtehokkuus riippuu taas tämän nopeudesta suorittaa tietty työvaihe. Tehokkuuteen pitää myös määritelmän mukaan suhteuttaa tuotannon laatu, mutta tuotantolinjalla tuotteet ovat joko laadunvalvonnan hyväksymiä tai hylkäämiä. Tämän perusteella voidaan vähentää hylätyt tuotteet kokonaistuotannon määrästä, tai vaihtoehtoisesti huomioida ne tarvittavien korjauksien jälkeen. Molemmissa tapauksissa kokoonpanolinjaston tehokkuudeksi voidaan määrittää valmistuneet tuotteet tunteihin suhteutettuina.[3]

Ongelma työtehokkuuden määrittämisessä kuitenkin ilmenee, kun tarkastellaan teollista suunnittelua: tietyn suunnitteluyksikön tehokkuutta ei oikeellisesti edusta yksikön tuottamien prototyyppien määrä. Tämä johtuu siitä, että suunnittelutyön laatukriteerit eroavat tuotantotyön yksikäsitteisistä laatukriteereistä. Tuottamalla vähemmän suunniteltuja prototyyppejä saadaan yksikön tuotanto määrällisesti nousemaan, mutta vähemmän suunnitellulla prototyypillä on huonompi laatu kuin prototyypillä, jonka suunnitteluun on kulutettu enemmän aikaa. Korkeimman työtehokkuuden pitäisi ilmetä tilanteessa, jossa suunnitteluun käytetty aika ja aikaansaadun tuotteen laatu ovat parhaassa mahdollisessa suhteessa, mutta tätä suhdetta on vaikea arvioida laadun monimutkaisuuden takia. [3]

Ohjelmointi on työnkuvaltaan hyvinkin samankaltainen teolliseen suunnitteluun: parhaan laadun ja määrän suhteen määrittäminen on samalla tavalla vaikeasti saavutettavissa. Esimerkiksi hitaammin tuotettu koodi on usein paremmin suunniteltua ja näin myös tehokkaampaa tuottamaan arvoa asiakkaalle.[4]

Ohjelmistoon liittyvää tehokkuutta voidaan tarkastella kolmesta eri näkökulmasta: itse ohjelmistokehittäjän näkökulmasta, ohjelmiston käyttäjän näkökulmasta sekä kehitystiimin johdon näkökulmasta [5]. Ohjelmistokehittäjän näkökulmasta ohjelmiston tehokkuus tarkoittaa tuotetun ohjelmiston määrää suhteutettuna sen tuottamiseen kuluneeseen aikaan. Ohjelmiston käyttäjän näkökulmasta tarkasteltuna taas tehokkuus tarkoittaa ohjelman kykyä tuottaa arvoa siinä tehtävässä, johon ohjelmaa käytetään [1]. Myös

tämän takia ohjelmoijan tehokkuus on vaikea käsite hallinnoida: mikäli tarkastellaan pelkästään tuotetun ohjelmakoodin määrää, ei välttämättä saada selville ohjelman hyödyllisyyttä asiakkaalle.

Tässä työssä keskitytään ohjelmistokehittäjän näkökulmaan työtehokkuudesta sen käytännön hyötyjen takia. Vaikka ohjelma olisi esimerkiksi erittäin tuottava käyttäjien hyödyntäessä sitä, ohjelmistoon on saattanut kulua sitä kehittäneen yrityksen puolesta niin paljon työtä, etteivät asiakkaalta saadut tulot kata kulunutta työpanosta. Tällöin ohjelmiston kehitystiimin johdon näkökulmasta tarkasteltuna ohjelmiston tehokkuus on huono [1]. Tämä laajempi näkökulma taas voidaan kohdistaa yksittäisen ohjelmoijan työtehokkuuteen, mikä on myös tämän työn päämäärä.

Ohjelmoijan työtehokkuuteen vaikuttaa muun muassa ohjelmoijan osaaminen suhteutettuna toteutettavan ohjelmiston monimutkaisuuteen ja vaikeuteen. Boehmin tutkimuksessa [6] havaittiin, että korkean osaamisen työntekijät yhdistettynä matalaan projektin monimutkaisuuteen tuottivat parhaat lopputulokset työtehokkuudessa. Mikäli taas ohjelmoijan osaaminen oli huonoa ja projekti monimutkainen, väheni työtehokkuus huomattavasti.

Monimutkaisuuteen vaikuttavat esimerkiksi toteutettavan ohjelmiston määrittelyn reunaehdot, kuten suorituskykyvaatimukset. Vosburghin tutkimuksessa [7] todettiin, että työtehokkuus laskee, kun ohjelmaa pitää optimoida esimerkiksi suorittimen tai muistin käytön suhteen. Tutkimuksessa huomattiin myös, että asiakkaan kokemuksella ohjelmiston määrittelyjen luonnissa on vaikutusta ohjelmoijan tuottavuuteen: mikäli asiakkaalla on paljon kokemusta määrittelyjen laatimisesta, on toteuttajan helpompi tulkita ohjeita ja muodostaa niistä käytännön toteutuksia.

Ohjelmoinnin tehokkuudesta saatavat hyödyt riippuvat kuitenkin huomattavasti kehitettävän ohjelmiston koosta. Esimerkiksi suurikokoisilla ohjelmistoilla, joiden koodirivien määrä sijoittuu asteikolle 50000-500000 riviä koodia, työtehokkuuden parannukset tuottavat rahallisesti paljon enemmän arvoa verrattuna pienikokoisen ohjelmiston kehityksen optimointiin [8]. Tämä taas tarkoittaa että mikäli työtehokkuuden parantamiseen käytetään tietty aika, saadaan sille enemmän arvoa suuren skaalan ohjelmistoprojekteissa.

3. TYÖTEHOKKUUDEN MITTAAMINEN

Tässä luvussa käydään eri tapoja seurata ohjelmoijan työtehokkuutta. Eri vaihtoehdoista esitellään niiden vahvuudet ja heikkoudet, joiden perusteella tutkitaan optimaalinen ratkaisu tehokkuuden valvontaan.

3.1 Biologinen mittaaminen

Eräs mahdollinen keino seurata ohjelmoijan työtehokkuutta on kehon fyysisten merkkien avulla. Yksi mahdollinen toteutus tälle on käyttää aivosähkökäyrämittausta (EEG) ohjelmoijan aktiivisuuden mittaamiseen, kuten S. Radevskin, H. Hatan ja K. Matsumoton tutkimuksessa [9]. Tutkimuksessa ohjelmoijia asetettiin tuottamaan ohjelmakoodia samalla pitäen EEG-laitetta päässään. Näistä mittauksista saatiin raakadata, jonka jaottelua varten toteutettiin koneoppista hyödyntävä luokittelija. Luokittelija kalibroitiin tunnistamaan hermoston eli tiloja saadusta datasta. Tilat luokiteltiin sellaisiin, joissa ohjelmoija tuotti ohjelmakoodia ja sellaisiin, joissa ohjelmoija ei tehnyt mitään. Näiden tilojen vaihtumista seuraamalla voidaan tarkastella tuottavien tilojen prosentuaalinen osuus ajan suhteen ja näin määrittää ohjelmoijan työtehokkuus kuluneelta ajalta.

Toinen mahdollisuus on seurata silmien liikettä ja hyödyntää sitä eri tilojen määrityksessä, kuten Leen ja Hooshyarin tutkimuksessa [10] tehtiin. Tutkimuksessa hyödynnettiin sekä EEG-laitetta että silmien seurantalaitetta muodostamaan raakadata, joka samalla tavalla syötettiin koneoppimiseen pohjautuvalle luokittelijalle. Tutkimuksessa luokittelijaa käytettiin hankkimaan tietoja ohjelmoijien asiantuntemuksen tasosta ja sen vaikutuksesta koettuun ohjelmointitehtävän vaikeuteen. Tämä tarkoittaa että näiden kahden esitellyn tutkimuksen tuloksia ei voida suoraan vertailla keskenään paremman biologisen mittausmenetelmän selvitykseen.

Toisaalta tärkein osuus biologisessa seurannassa ei ole pelkkä fyysinen toteutus seurantalaitteesta, vaan enemmänkin raakadatan luokittelun toteutus [10]. Koneoppiminen mahdollistaa syy- ja seuraussuhteiden muodostamisen biologisten ilmiöiden ja korkean tason konseptien toteutumisen välillä [11]. Tämän perusteella sekä EEG-mittaus että silmien liikkeiden seuranta ovat mahdollisia välineitä raakadatan hankinnalle, riippuen siitä, kumpi on seurannan toteuttajalle luontevampi toteuttaa.

Biologisten mittausmenetelmien ongelma on kuitenkin niiden monimutkainen toteutus ja tästä seurauksena lisääntynyt vaiva seurannan kohteelle: Radevskin tutkimuksessa [9] todettiin, että suurin osa EEG-laitteistolla toteutettuun seurantaan osallistuneista oh-

jelmoijista tunsi laitteen epämukavaksi. Ihmiset suhtautuvat myös seurantalaitteisiin kielteisesti: varsinkin erillisillä laitteilla tehtävä seuranta koetaan liian holhoavaksi ja tämän takia ahdistavaksi [12].

Laitteiston käyttöönotto kokonaiselle kehitystiimille vaatii myös lisäinvestointeja: yksi kappale tutkimuksessa käytettyjä laitteita maksaa 800 dollaria kappaleelta [13]. Toinen vaihtoehto olisi vaihtaa yrityksen tietokoneet sellaisiin, jotka tukevat esimerkiksi silmien seurantaa vakiona [14]. Kuitenkin laitteiston lisäksi tarvitaan mittausohjelmisto tukemaan siitä tulevia syötteitä, mikä vaikeuttaa edelleen päätöstä seurannan toteuttamisesta yrityksen näkökulmasta.

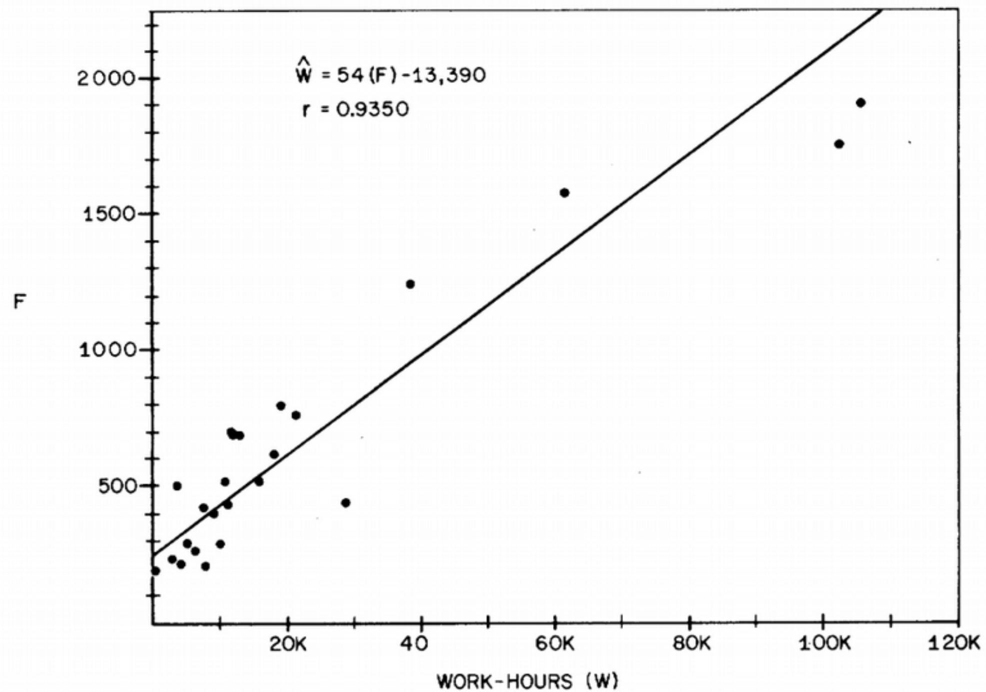
3.2 Ohjelmakoodin analysointi

Toinen mahdollinen lähde tuottavuuden mittaamiselle on ohjelmoijan tuottama materiaali eli itse ohjelmakooditiedostot. Näitä tutkiessa yksinkertaisin mittayksikkö on rivien määrä: mitä enemmän ohjelmoija tuottaa ohjelmakoodia, sitä enemmän rivejä on tiedostoissa. Tämän perusteella voidaan laskea tietyllä aikavälillä kirjoitetun koodin määrä ja sen perusteella määrittää ohjelmoijan työtehokkuus numeerisena arvona. [1]

Ongelmana rivien määrää mitattaessa on kuitenkin aiemmin mainittu ohjelmistojen tehokkuuden vaikea määrittely. Mikäli ohjelmoija tuottaa nopeasti ohjelmakoodia, hänen tuottavuutensa on rivien mukaan arvioituna hyvä, vaikka parempi lopputulos saataisiin tuottamalla huolellisempaa ohjelmakoodia. Paremmin suunnitellulla ohjelmakoodilla valmiin tuotteen tehokkuus on parempi, mikä tarkoittaa parempaa laatua. [1]

Tämän perusteella on järkevämpää tarkastella ohjelmakoodin suurempina kokonaisuuksina rivimäärän sijasta. Yleisesti näitä kokonaisuuksia mallinnetaan funktionaalisten pisteiden avulla, jotka ovat mittayksikkö ohjelmakoodin toiminnallisille osille, kuten funktioille ja luokille. Mittaamalla ohjelmakoodiin luotujen funktionaalisten pisteiden määrää tietylle aikavälille suhteutettuna voidaan edelleen laskea arvo ohjelmoijan työtehokkuudelle. [15]

Tutkitaan nyt erästä esimerkkiä funktionaalisten pisteiden määrästä verrattuna niiden tuottamiseen kuluneeseen aikaan. Albrechtin ja Gaffneyn tutkimuksessa [15] havaittiin, että funktionaalisten osien määrän kasvu korreloi selkeästi työmäärän kasvun kanssa. Havainnollistus korrelaatiosta on esitetty kuvassa 1.



Kuva 1: funktiopisteet suhteutettuna työtunteihin [15]

Kuvasta voidaan huomata, että varsinkin korkeilla työtuntimäärillä mittauspisteitä on varsin vähän. Esimerkiksi 1500 funktionaalisen pisteen kokoisilla ohjelmilla kehitykseen kuluvien työtuntien määrä vaihtelee kymmenillätuhansilla. Tämän takia funktionaaliset pisteetkään eivät takaa täysin tarkkaa menetelmää ohjelmiston koon laskennasta. [15]

Funktionaalisten pisteiden tarkkuutta haittaa myös niiden määrän korrelointi ohjelmakoodin rivien määrän kanssa. Tätä ilmiötä selvitettiin esimerkiksi Czarnacka-Chrobotin tutkimuksessa [16], jossa laskettiin yhden funktionaalisen pisteen arvo eri ohjelmointikielissä. Arvon mittana käytettiin keskimääräistä koodirivien määrää yhtä funktionaalista pistettä kohden. Tutkimuksen tietojen perusteella taulukkoon 1 on koottu eri ohjelmointikielien vaatimia koodirivimääriä yhtä funktionaalista pistettä kohden.

Taulukko 1: Esimerkki ohjelman tietokannan taulusta [16]

Ohjelmointikieli	**Keskimääräinen koodirivien määrä yhtä funktionaalista pistettä kohden**
Assembly	320
C	128
Lisp	64
C++	53
Java	53

Taulukosta huomataan, että korkeamman tason ohjelmointikielillä tarvitaan vähemmän ohjelmakoodirivejä yhden funktionaalisen pisteen tuottamiseen, mikä kuvaa myös ohjelmakoodin rivimäärän ja funktionaalisten pisteiden määrän korrelaatiota [16]. On siis mahdollista, että voidaan samaan tapaan tuottaa korkealla työtehokkuudella ohjelmakoodia, joka on laadultaan huonoa [11]. Esimerkiksi jakamalla ohjelmakoodin toiminnallisuutta tarpeettoman pieniin palasiin luomalla turhia luokkia ja funktioita ohjelmakoodista tulee vaikeasti seurattavaa, mutta työtehokkuus nousee funktionaalisten pisteiden määrän lisääntyessä.

Funktionaalisten pisteiden kautta mittaamisen huonona puolena on sen toteuttamisen skaalautuvuus: koska funktionaaliset pisteet ovat erilaiset joka ohjelmointikielelle ja ohjelmointikieliä on olemassa runsaasti erilaisia, mahdolliseen toteutukseen pitäisi tehdä tuki jokaiselle ohjelmointikielelle erikseen. Tämän takia pelkästään ohjelmakoodia analysoiva tuottavuuden laskentaohjelman kehittäminen veisi huomattavan paljon aikaa, mikäli siihen haluttaisiin mahdollisimman laaja kattavuus eri ohjelmointiprojekteihin. [15]

3.3 Mittaaminen käyttöjärjestelmästä

Käyttäjän aktiivisuutta voidaan seurata muillakin tavoin kuin suoraan kehosta eri antureita hyödyntäen. Eri käyttöjärjestelmät tarjoavat rajapintoja esimerkiksi ikkunoiden tietojen sekä hiiren ja näppäimistön seuraamiseen. Esimerkiksi Windows-ympäristöissä ikkunoiden tietoja voidaan hankkia Windows Controls-rajapinnalla [17], kun taas Linux-ympäristössä saman toiminnallisuuden voi saavuttaa Xlib-rajapinnalla [18]. Näitä hyödyntämällä voidaan mitata aikaa, jolloin käyttäjä on aktiivinen, suhteuttaa tämä kuluneeseen aikaan ja tämän perusteella laskea myös ohjelmoijan työtehokkuus.

Aktiivisen ajan mittaamista varten pitää ensin määritellä, milloin käyttäjä on aktiivinen. Suurinta osaa ohjelmistoista toteutetaan erillisissä kehitysympäristöissä, IDE:ssä. Tällöin voidaan olettaa, että ohjelmoijan siirtyessä johonkin toiseen ohjelmaan hän ei enää kirjoita ohjelmakoodia ja eikä näin ole enää sillä hetkellä tuottava. Voidaan myös olettaa, että mikäli ohjelmoija ei liikuta hiirtä tai anna näppäimistöllä syötettä tietokoneelle tietyn ajan kuluessa, ei hän ole silläkään hetkellä tuottava vaan tekee jotain muuta. Näiden tietojen perusteella voidaan määrittää tuottavat ja ei-tuottavat tilat samaan tapaan kuin biologisessa seurannassa, tosin erona se, ettei käyttöjärjestelmästä mittaaminen vaadi erillistä laitteistoa.

On kuitenkin huomioitava, että käyttöjärjestelmästä mittaaminen kärsii samoista ongelmista kuin tuotettujen ohjelmakoodin rivien määrän mittaaminen: mittausmenetelmä pi-

tää huonolaatuista mutta nopeasti kirjoitettua ohjelmakoodia suuremmassa arvossa kuin hitaasti kirjoitettua mutta laadukkaampaa ohjelmakoodia.

Toinen käyttöjärjestelmäpohjaisen seurannan heikkous on sen helppo huijattavuus: mikäli käyttäjä esimerkiksi automatisoi hiiren liikkumaan tietyn ajan välein ja pitää tarkkaillun ohjelman auki tietokoneellaan, pitää seurantamenetelmä häntä aktiivisena. Tämä johtuu siitä, että seurantamenetelmä laskee kaikki hetket, jolloin havaitaan hiiren ja näppäimistön liikkeitä tuottaviksi tiloiksi. Toisaalta tällaista ongelmaa ei ole biologisessa seurannassa lainkaan, koska tuottavat tilat määritellään hermoston tilojen perusteella, joita on huomattavasti vaikeampi huijata [9].

3.4 Optimaalinen ratkaisu

Vertaillaan nyt eri mittaustapojen hyötyjä ja haittoja ja määritellään niiden perusteella optimaalinen toteutus ohjelmoijan työtehokkuuden seuraamiseen. Biologinen seuranta vaatii sekä erillistä laitteistoa että oman ohjelmistonsa seurantadatan tulkintaa varten. Toisin kuin muut esitellyt seurantamenetelmät, laitteet ovat vaivalloisia käyttää epämukavuutensa takia, varsinkin kun verrataan mukavuutta puhtaasti ohjelmistopohjaisiin toteutuksiin. Myöskin työntekijöiden luottamus biologisiin seurantamenetelmiin on ongelmallinen: vaikka mittauslaitteet olisivatkin käyttömukavuudeltaan hyviä, niiden olemassaolo koetaan tarpeettomaksi ja painostavaksi työntekijöiden mielestä [12]. Biologinen seuranta on tämän perusteella niin epäkäytännöllistä, ettei siitä ole tarpeeksi hyötyä käytännön sovelluksiin.

Sekä ohjelmakoodin analysointi että käyttöjärjestelmän seuranta ovat toteutettavissa millä tahansa tietokoneella, joten paras mahdollinen toteutus olisi yhdistellä niitä. Tässä toteutuksessa ei tarvita erillistä laitteistoa, vaan kaikki toiminnot toteutetaan ohjelmiston kautta. Mahdolliseen ohjelmatoteutukseen kuuluisi esimerkiksi moduuli, jonka toimintona on käyttöjärjestelmän seuranta. Toinen moduuli voisi olla ohjelmakoodin analysoija, joka tutkisi määritettyjä ohjelmakooditiedostoja ja muodostaisi oman kuvan työtehokkuudesta näiden avulla. Ohjelmaan määriteltäisiin myös painoarvot kummankin moduulin analysoimalle työtehokkuudelle, minkä jälkeen painotetut arvot muodostaisivat yhdessä ohjelmoijan työtehokkuuden.

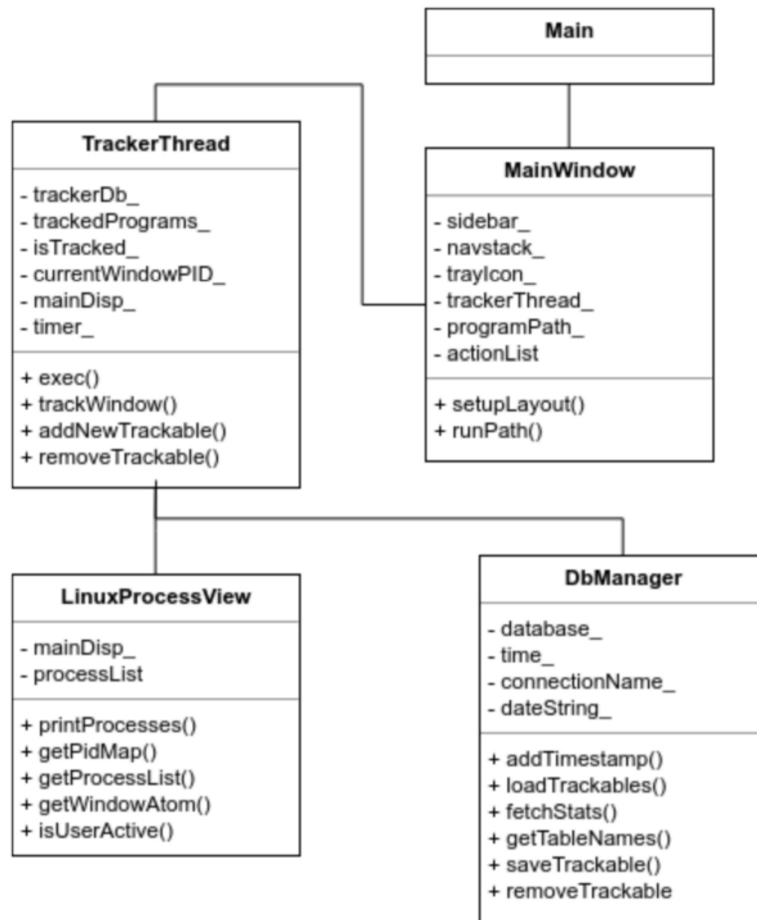
4. TYÖTEHOKKUUTTA MITTAAVA OHJELMA

Tässä luvussa tarkastellaan yhtä toteutusta työtehokkuuden seurantaan. Ohjelma on Linux-ympäristöön Qt:lla toteutettu ohjelma, joka tallentaa tietoja käyttäjän aktiivisuudesta tietokantaan ja muodostaa niistä kuvaajia käyttäjille. [19]

4.1 Ohjelman rakenne

Seuranta on toteutettu Linuxin ikkunointijärjestelmä Xorg:n rajapinnalla Xlib. Tätä hyödyntäen saadaan lista kaikista avoinna olevista ikkunoista sekä tämänhetkisestä etualalla olevasta ikkunasta [18]. Ohjelmassa hyödynnetään myös toista ikkunointijärjestelmän rajapintaa, XscreenSaver:ia. XscreenSaver on tarkoitettu näytönsäästäjien kehittämiseen ikkunointijärjestelmälle. Tämän takia se tarjoaa tiedon kuluneesta ajasta, jolloin käyttäjä ei ole antanut syötettä [20]. Syötteeksi lasketaan esimerkiksi näppäimistön tai hiiren painallus. Tätä ominaisuutta voidaan hyödyntää määrittämään tilat, jolloin käyttäjä on aktiivinen tai epäaktiivinen.

Ohjelman jakautuu päärakenteeltaan kahteen osaan: graafiseen käyttöliittymään ja taustaprosessiin. Graafinen käyttöliittymä kommunikoi käyttäjän kanssa näyttäen esimerkiksi tilastoja käyttäjän työtehokkuudesta, kun taas taustaprosessi seuraa käyttäjän toimintaa ja tallentaa tarvittavat tiedot työtehokkuuden kuvauksen muodostamiseen. Tässä työssä keskitytään tarkastelemaan taustaprosessin rakennetta, sillä graafinen käyttöliittymä keskittyy ainoastaan työtehokkuuden esittämiseen sekä seurattavien ohjelmien hallintaan. Tämän takia kaikki itse seurannan kannalta oleellinen voidaan havainnollistaa tarkastelemalla ohjelman taustaprosessista. Kuvassa 2 on esitetty ohjelman luokkakaavio siten, että vain taustaprosessin luokat ovat näkyvillä [19].



Kuva 2: ohjelman taustaprosessin luokkakaavio

Ohjelman käynnistyessä MainWindow-oliossa käynnistetään käyttäjän aktiivisuuden seurannan toteuttava TrackerThread-olio. TrackerThread:ssa on ajastin, joka suorittaa sekunnin välein trackWindow()-funktion, jonka tehtävänä on hankkia tieto tämänhetkisestä aktiivisesta ikkunasta. Funktio hakee LinuxProcessView-olion getWindowAtom()-funktiolla tiedon tämänhetkisestä aktiivisesta ikkunasta. Tämä funktio palauttaa useita ikkunaan liittyviä tietoja, joista yksi on prosessin numeraalimuotoinen tunniste, PID. Tätä verrataan edelliseen havaittuun tunnisteeseen, minkä avulla voidaan päätellä ikkunan vaihtuminen. Ikkunan vaihtuessa uuden ikkunan prosessin nimi kirjataan tietokantaan kutsumalla DbManager-olion addTimestamp()-funktiota. [19]

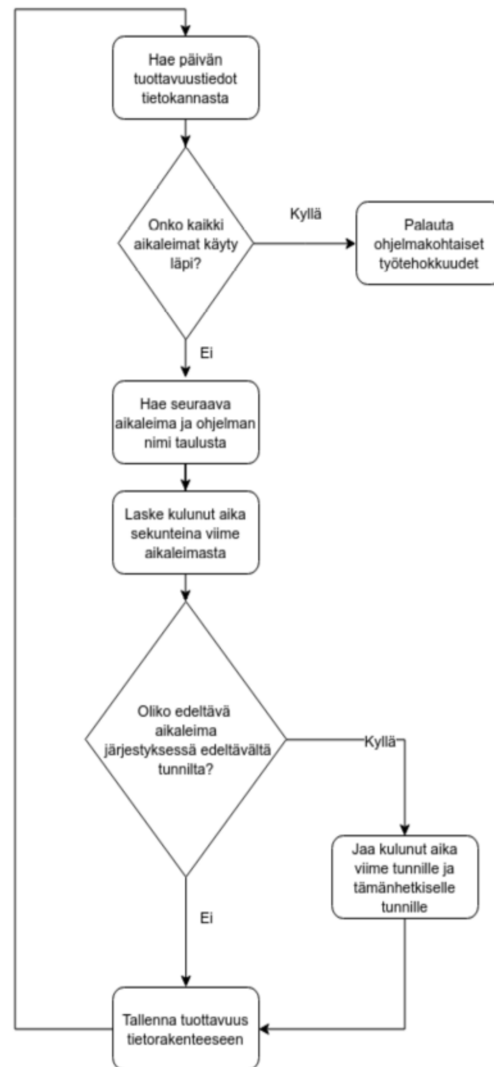
Aikaleimoja kirjataan myös käyttäjän aktiivisuustilan muuttuessa: XscreenSaver-rajapintaa hyödyntämällä voidaan tarkastella sitä, kuinka kauan aikaa on kulunut viimeisimmästä näppäimistön painalluksesta tai hiiren liikkeestä. Tämän perusteella voidaan asettaa tietokantaan merkintä epäaktiivisuudesta, vaikka ohjelma pysyisikin samana. Tämä parantaa tehokkuuden seurannan tarkkuutta, kun tilaa, jolloin käyttäjä ei tee mitään, havaita virheellisesti tuottavaksi.

4.2 Työtehokkuuden laskeminen

DbManager-olion fetchStats()-funktio hakee tietokannasta tiedot sille annetun päivämäärän perusteella. Koska aikaleimat on tallennettu aikajärjestyksessä, funktio tallentaa tietorakenteeseen kunkin tilan keston prosentuaalisen osuuden yhdestä tunnista. Tämä prosenttiarvo voidaan laskea kaavan 1 mukaisesti:

$$\sum_{i=1}^{N} \frac{t_i - t_{i-1}}{3600\,s} \quad , \tag{1}$$

jossa t_i on tietyn hetken aikaleima ja N on taulussa olevien aikaleimojen kokonaismäärä. Summa on ohjelma- ja tuntikohtainen ja kuvaa käyttäjän työtehokkuutta kyseisessä ohjelmassa. Koska työtehokkuuden määritelmä ohjelmassa on tilapohjainen, on se siis samankaltainen kuin EEG-mittauksessa käytetty työtehokkuuden määritelmä [9]. Funktion fetchStats() toiminnan yksinkertaistamiseksi muodostetaan sen ohjelmakoodin perusteella [19] algoritmin vuokaavio, joka on esitetty kuvassa 3.



Kuva 3: fetchStats()-funktion algoritmin vuokaavio

Tietokannassa aikaleimat on järjestetty siten, että kunkin päivän aikaleimoille on oma taulunsa. Jokainen taulu jakautuu kahteen sarakkeeseen, joista toinen on varattu ohjelman nimelle ja toinen itse aikaleimalle. Esimerkki tietokannan taulun rakenteesta on esitetty taulukossa 2.

Taulukko 2: Esimerkki ohjelman tietokannan taulusta

Ohjelma	**Aika**
Ohjelma1	15:43:01
NULL	15:45:15
Ohjelma2	15:51:43
NULL	16:10:32

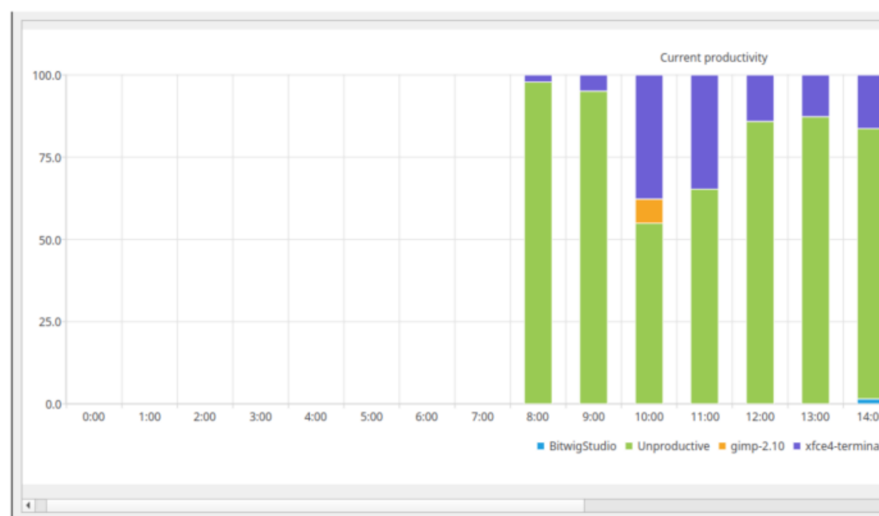
Funktio fetchStats() valitsee aluksi parametrina annetun päivämäärän perusteella tietokannan taulun. Tämän jälkeen aloitetaan silmukka, jossa käydään läpi jokainen taulus-

ta löytyvä rivi. Jokaisella iteraatiolla verrataan nykyistä aikaleimaa edelliseen aikaleimaan ja lasketaan niiden erotus, minkä jälkeen tilan kesto jaetaan 3600:lla ja summataan tietorakenteeseen ohjelman nimen perusteella kaavan 1 mukaisesti [19].

Poikkeustilanne kuitenkin syntyy, kun kahden aikaleiman erotus sijoittuu kahdelle eri tunnille. Esimerkiksi taulukossa 1 Ohjelma2 käyttö jatkuu 16:10:32 asti, mikä tarkoittaa, että osa työtehokkuudesta pitää kohdistaa tunnille 15 ja osa tunnille 16. Tämä tapahtuu siten, että algoritmi laskee aluksi kuluneen ajan välille 15:51:43 – 16:00:00, jakaa tämän 3600:lla ja lisää tämän tehokkuuden Ohjelma2:n tunnin 15 kohdalle. Tämän jälkeen lasketaan aika kulunut aika välille 16:00:00 – 16:10:32, jaetaan aika 3600:lla ja lisätään tämän tilan kesto taas Ohjelma2:n tunnin 16 kohdalle [19].

Taulukon 2 tietojen perusteella voidaan sanoa, että käyttäjä on ollut tuottava ohjelmassa Ohjelma1 kaksi minuuttia ja 14 sekuntia, kun taas epätuottavaa aikaa on kertynyt kuusi minuuttia ja 28 sekuntia. On tosin huomioitava, että todellisessa käytössä aikaleimoja olisi huomattavasti enemmän samalla aikavälillä, sillä käyttäjän aktiivisuutta tarkastellaan ohjelmassa tiheästi: TrackerThread-luokan funktio trackWindow() tarkastelee tilannetta sekunnin välein ja merkkaa aikaleiman *Null* käyttäjän oltua epäaktiivinen yhden minuutin ajan. [19]

Funktio fetchStats() jatkaa tietyn taulun läpikäymistä niin kauan kuin se löytää sieltä uusia aikaleimoja. Funktio palauttaa lopuksi tietorakenteen, joka sisältää määrätyn päivän eri ohjelmien tuntikohtaiset tehokkuudet. Nämä tehokkuuden ohjelma esittää graafisessa käyttöliittymässä pylväsdiagrammeina. Esimerkki mahdollisista tuloksista on esitetty kuvassa 4.



Kuva 4: esimerkkiesitys tunnin välein mitatusta tuottavuudesta

Ohjelma näyttää jokaisen tunnin kohdalla eri ohjelmien tuottavuuden sekä tuottamattoman ajan. Funktio fetchStats() laskee myös tuottamattoman ajan vähentämällä kaikki tuottavan ajan ohjelmien osuudet kuluneen ajan kokonaisosuudesta.

4.3 Ohjelman vertailu muihin seurantamenetelmiin

Kuten aiemmin mainittiin, ohjelman seuranta muistuttaa jonkin verran luvussa 2 käsiteltyä esimerkkiä biologisesta seurannasta. Ohjelman toteutus ei kuitenkaan vaadi lainkaan lisälaitteistoa, vaan toimii käyttöjärjestelmän rajapintoja käyttäen. Toisaalta ohjelman seurantaa on helppo huijata: vaikkei ohjelmoija tekisikään mitään tuottavaa, hän voi silti antaa syötteitä tietokoneelle hiirellä ja näppäimistöllä ja näin vaikuttaa tuottavalta ohjelman näkökulmasta. Biologisen toteutusta on taas huomattavasti vaikeampi huijata, koska tilat perustuvat hermoston syötteisiin, eivätkä rajoitu pelkästään tietokoneen saamiin syötteisiin [9].

Ohjelma ei myöskään sisällä lainkaan toteutusta funktionaalisten pisteiden analyysistä. Esitelty ohjelma on alun perin tarkoitettu yleispäteväksi tuottavuuden mittariksi, eikä vain pelkästään ohjelmoijan työtehokkuuden mittaajaksi [19]. Kuten luvussa 2 mainittiin, käyttöjärjestelmästä seuraamisen tarkkuutta voitaisiin parantaa huomattavasti lisäämällä ohjelmaan toteutus ohjelmakoodin analysoinnista. Tämä tosin vähentäisi ohjelman skaalautuvuutta muihin työtehokkuuden seurannan tilanteisiin.

Mikäli oletetaan, että esiteltyä ohjelmaa haluttaisiin muokata ainoataan ohjelmoijan työtehokkuutta seuraavaksi, eräs mahdollinen toteutus olisi lisätä moduuli, joka tutkii ohjelmoijan muokkaamia ohjelmakooditiedostoja ja etsii niistä funktionaalisia pisteitä. Tämän jälkeen tiedot pisteiden määrästä voitaisiin lisätä omaan tietorakenteeseensa tiedostokohtaisesti . Muodostettujen funktionaalisten pisteiden määrää voidaan käyttää painoarvona työtehokkuudelle, jolloin työtehokkuuden määrä kerrottaisiin funktionaalisten osien määrällä. Tämä mahdollistaisi tarkan hetkellisen seurannan, koska vaikka uusia funktionaalisia osia ei syntyisi ohjelmakoodiin, niin ohjelmoijan kehitystyökalujen käyttö tunnistettaisiin, ja voitaisiin arvioida ohjelmoijan tekevän ainakin jotain. Toisaalta painoarvon käyttö tasoittaisi myös huonosti suunnitellun koodin tuottavuutta: vaikka ohjelmoija olisi kehitysympäristössään aktiivinen ja tuottaisi paljon ohjelmakoodia, hänen työtehokkuutensa määritettäisiin silti huonoksi, jos kaikki ohjelmakoodi on laitettu vain yhteen funktioon tai luokkaan.

5. YHTEENVETO

Ohjelmoijan työtehokkuus on vaikeasti määriteltävä käsite: ohjelmistokehitys ei noudata samanlaisia määrän ja tuotetun arvon suhteita kuin perinteinen valmistusteollisuus. Ohjelmoijan työtehokkuuden eri lukemia on myös vaikea soveltaa: asiakkaan saama hyöty ohjelmistosta ei riipu suoraan ohjelmoijan työtehokkuudesta.

Ohjelmoijan työtehokkuuden seuraamiseen on useita eri tapoja, joista jokaisella on omat hyvät ja huonot puolensa. Biologisella seurannalla sekä käyttöjärjestelmäpohjaisella seurannalla on hyvä skaalautuvuus, mutta huono tarkkuus. Samoin pelkästään ohjelmakoodin rivien määrää mittaamalla ei saada hyvää tarkkuutta työtehokkuuden mittaukseen.

Funktionaaliset pisteet on muita tapoja tarkempi, mutta samalla huonommin skaalautuva eri ohjelmointikieliin: jokaisen ohjelmointikielen syntaksi eroaa muista, mikä tarkoittaa sitä, että jokaisen uuden kielen tukeminen tarkoittaa lisää työtä. Näiden tietojen perusteella paras mahdollinen toteutus ohjelmoijan työtehokkuuden seurantaan on yhdistää rivien määrän seuranta, funktionaalisten pisteiden seuranta sekä käyttöjärjestelmän seuranta yhdeksi toteutukseksi.

LÄHTEET

[1] Petersen K. Measuring and predicting software productivity: A systematic map and review. Inf Softw Technol [Internet]. 2011;53(4):317–43. Saatavissa: http://www.sciencedirect.com/science/article/pii/S0950584910002156

[2] Tokarčíková E. Measurement of Highly Qualified Employees Productivity. Ann Univ Dunarea Jos Galati Fascicle I, Econ Appl Informatics. 2013;19(3):5–10.

[3] Chew WB. No-Nonsense Guide to Measuring Productivity. Harv Bus Rev [Internet]. tammikuuta 1988;66(1):110–5. Saatavissa: http://search.ebscohost.com/login.aspx?direct=true&AuthType=cookie,ip,uid&db=bsu&AN=8800004433&site=ehost-live&scope=site&authtype=sso&custid=s4778523

[4] Solla M, Patel A, Wills C. New metric for measuring programmer productivity. Isc 2011 - 2011 IEEE Symp Comput Informatics. 2011;177–82.

[5] Johnson LF. On measuring programmer team productivity. Conf Proc IEEE Can Conf Electr Comput Eng Cat No98TH8341 [Internet]. 1998;2:701–5. Saatavissa: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=685594

[6] Chrysler E. Some Basic Determinants of Computer Programming Productivity. Commun ACM [Internet]. 1978;21(6):472–83. Saatavissa: http://doi.acm.org/10.1145/359511.359523

[7] Vosburgh J, Curtis B, Wolverton R, Albert B, Malec H, Hoben S, ym. Productivity Factors and Programming Environments. Teoksessa: Proceedings of the 7th International Conference on Software Engineering [Internet]. Piscataway, NJ, USA: IEEE Press; 1984. s. 143–52. (ICSE ’84). Saatavissa: http://dl.acm.org/citation.cfm?id=800054.801963

[8] Scacchi W. Understanding Software Productivity [Internet]. Vsk. 4. 1995. Saatavissa: https://www.researchgate.net/publication/2620175_Understanding_Software_Productivity

[9] Radevski S, Hata H, Matsumoto K. Real-time monitoring of neural state in assessing and improving software developers’ productivity. Proc - 8th Int Work Coop Hum Asp Softw Eng CHASE 2015. 2015;93–6.

[10] Lee S, Hooshyar D, Ji H, Nam K, Lim H. Mining biometric data to predict programmer expertise and task difficulty. Cluster Comput. 2017;1–11.

[11] Bednarik R, Vrzakova H, Hradis M. What Do You Want to Do Next: A Novel Approach for Intent Prediction in Gaze-based Interaction. Teoksessa: Proceedings of the Symposium on Eye Tracking Research and Applications [Internet]. New York, NY, USA: ACM; 2012. s. 83–90. (ETRA ’12). Saatavissa: http://doi.acm.org/10.1145/2168556.2168569

[12] Hobel J. Employee tracking a slice of Orwell’s 1984. Can HR Report [Internet]. 24. huhtikuuta 2000;13(8):4. Saatavissa:

https://libproxy.tuni.fi/login?url=https://search.proquest.com/docview/220805948?accountid=14242

[13] EMOTIV. EMOTIV EPOC+ 14 Channel Mobile EEG [Internet]. [viitattu 4. huhtikuuta 2019]. Saatavissa: https://www.emotiv.com/product/emotiv-epoc-14-channel-mobile-eeg/

[14] Acer Rolls Out Consumer Notebook with Tobii Eye Tracking. Prof Serv Close - Up [Internet]. 5. tammikuuta 2017; Saatavissa: https://libproxy.tuni.fi/login?url=https://search.proquest.com/docview/1855288003?accountid=14242

[15] Albrecht AJ, Gaffney JE. Software Function, Source Lines of Code and Development Effort Prediction. Ieee Tse. 1983;8(6):639–48.

[16] Czarnacka-Chrobot B. What Is the Cost of One IFPUG Method Function Point?- Case Study. Proc Int Conf Softw Eng Res Pract. 2012;(April):6.

[17] Microsoft. Windows Controls [Internet]. [viitattu 4. huhtikuuta 2019]. Saatavissa: https://docs.microsoft.com/fi-fi/windows/desktop/Controls/window-controls

[18] Tronche C. The Xlib Manual [Internet]. 2005 [viitattu 4. huhtikuuta 2019]. Saatavissa: https://tronche.com/gui/x/xlib/

[19] Wuoti E. Workaholic [Internet]. 2019 [viitattu 4. huhtikuuta 2019]. Saatavissa: https://gitlab.com/Wwuoti/Workaholic/

[20] Fulton J, Packard K. XScreenSaver [Internet]. [viitattu 4. huhtikuuta 2019]. Saatavissa: https://www.x.org/releases/X11R7.7/doc/man/man3/Xss.3.xhtml