Tampereen yliopisto

Toivo Knuutinen

TYYPPIJÄRJESTELMÄT VERTAILUSSA

JavaScript ja TypeScript

Kandidaatintutkielma
Informaatiotekniikan ja viestinnän tiedekunta
Tarkastaja: Maarit Harsu
Toukokuu 2024

TIIVISTELMÄ

Toivo Knuutinen
Kandidaatintutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma, tietotekniikka
Toukokuu 2024

JavaScript on yksi maailman suosituimmista ohjelmointikielistä, ja sitä käytetään erityisesti web-kehityksessä melkein jokaisella verkkosivulla. Se mahdollistaakin web-sivuilla interaktiivisen käyttökokemuksen. Usein JavaScriptin ajatellaan kuitenkin vaikeuttavan ohjelmistokehitystä sen heikon tyyppijärjestelmän takia. Vaihtoehtona JavaScriptille on luotu TypeScript, joka sisältää staattiset tyyppitarkistukset sekä estää implisiittisiä tyyppimuunnoksia.

Tutkielma on toteutettu kirjallisuuskatsauksena ja sen tavoitteena on tutkia, miten TypeScript vaikuttaa JavaScriptiin verrattuna ohjelmistoihin ja ohjelmistojen kehitysprosessiin. Vertailussa käytetään kolmea ohjelmistokehityksen vertailukriteeriä: ohjelmointivirheiden esiintyvyyttä, ohjelmistojen ylläpidettävyyttä ja ohjelmistojen kehitysnopeutta. Tutkielmassa verrataan sekä TypeScriptiä ja JavaScriptiä suoraan, että muita samat tyyppijärjestelmät omaavia kieliä.

Vaikka TypeScript tuo mukanaan monia hyötyjä, se ei poista kaikkia JavaScriptiin liittyviä ongelmia, erityisesti kun kyse on DOM-vuorovaikutuksesta web-ohjelmoinnissa. Tullaan tulokseen, että TypeScript ja staattiset tyyppijärjestelmät parantavat kuitenkin koodia ainakin ohjelmointivirheiden esiintyvyyden ja ohjelmistojen ylläpidettävyyden näkökulmista. Vaikuttaa myös siltä, että pienissä ohjelmointitehtävissä, ainakin niissä, joissa vaaditaan tyyppimuunnoksia, dynaamisilla tyyppijärjestelmillä kehitys on nopeampaa. Suuremmissa ohjelmistoissa tyyppimuunnoksien vaikutus kehitysnopeuteen voi kuitenkin jäädä muiden seikkojen varjoon. Havaittiin myös, että TypeScript nopeutti ohjelmistojen kehitystä isommissa ohjelmointitehtävissä.

Tutkimuksen perusteella TypeScript on hyvä vaihtoehto perinteiseen JavaScript-ohjelmistokehitykseen. TypeScript sallii sekä dynaamisen että staattisen tyypityksen vähittäisen tyypityksen kautta. Siksi se mahdollistaa pienempien prototyyppien kehittämiseen nopeasti dynaamisesti tyypitettynä, sallien kuitenkin hiljattain siirtymisen staattiseen tyypitykseen myöhemmin. JavaScriptin käyttö voi olla perusteltua pienissä sovelluksissa sen joustavuuden vuoksi, mutta muut tyypintarkastajien tarjoamat edut puoltavat TypeScriptin käyttöä myös pienissä projekteissa.

Avainsanat: TypeScript, JavaScript, tyyppijärjestelmät, ohjelmistokehitys

SISÄLLYSLUETTELO

1. JOHDANTO

Ohjelmistokehitys on moniulotteinen prosessi, jossa ohjelmointikielillä on merkittävä vaikutus sekä itse kehitysprosessiin että sen lopputulokseen eli syntyvään ohjelmistoon. JavaScript on erityisesti web-kehityksessä käytetty ohjelmointikieli, jonka avulla voidaan nopeasti luoda interaktiivisia verkkosivuja. JavaScript tuo dynaamisen ja heikon tyypityksen ansiosta joustavuutta ohjelmistojen kehitykseen. Usein dynaaminen tyypitys nähdään kuitenkin myös sen ongelmana. Staattisten tyyppitarkastusten puuttuminen voi johtaa sovelluksen suorituksen aikaisiin virheisiin, joita ei usein huomata ennen sovelluksen käyttöönottoa. Tällaisten virheiden korjaaminen jälkikäteen voi olla aikaa vievää ja turhauttavaa. Myös heikko tyypitys voi aiheuttaa vaikeasti jäljitettävissä olevia virheitä. Vaikka JavaScriptiä pidetään virhealttiina, on se yksi suosituimmista ohjelmointikielistä. Muiden vaihtoehtojen tutkiminen voi auttaa kehittäjiä luomaan laadukkaampia ohjelmistoja.

Yksi vaihtoehto tyypittämättömälle JavaScriptille on vahvan ja staattisen tyypityksen omaava ja Microsoftin kehittämä ohjelmointikieli TypeScript. Tämän tutkielman tavoitteena on tutkia TypeScriptin vaikutusta ohjelmistoihin ja niiden kehitysprosessiin verrattuna perinteiseen, tyypittämättömään JavaScript-kehitykseen. Tutkielmassa suoritetaan kirjallisuuskatsaus, jonka avulla vertaillaan JavaScriptin ja TypeScriptin ominaisuuksia ja vahvuuksia. Vertailussa käytettäviä kriteerejä ovat ohjelmistojen ohjelmointivirheiden esiintyvyys, ylläpidettävyys ja kehitysnopeus.

Kirjallisuuskatsaus suoritettiin hakemalla TypeScriptiä ja JavaScriptiä sekä eri tyyppijärjestelmiä vertailevia tutkimuksia hakupalveluista ja tutkimustietokannoista kuten Andor ja IEEE Xplore. Useassa tutkimuksessa annettiin ohjelmointitehtäviä suoritettavaksi eri tyyppijärjestelmillä sekä tarkasteltiin tehtäviin kuluvaa aikaa. Lisäksi muun muassa vertailtiin eri kieliä käyttäviä GitHub-repositorioita tai tarkasteltiin koodin laatua staattisen analyysin työkaluilla. Yhdessä tutkimuksessa myös lisättiin tyypit TypeScriptillä virheitä sisältävälle JavaScript-koodille, tavoitteena saada korjausapua virheisiin.

Luvussa 2 määritellään vertailun kannalta olennaiset tyyppijärjestelmät ja luvussa 3 esitellään käytetyt vertailukriteerit ja kerrotaan, miksi niitä käytetään tutkielmassa. Luvussa 4 esitellään JavaScript ja TypeScript. Luvussa 5 kieliä vertaillaan ohjelmointivirheiden

esiintyvyyden, ohjelmiston ylläpidettävyyden ja kehitysnopeuden näkökulmista. Luvussa 6 kootaan tutkielman tulokset ja luvussa 7 suoritetaan yhteenveto.

2. TYYPITYKSESTÄ YLEISESTI

Tyyppijärjestelmät ohjelmointikielessä asettavat säännöt muuttujien tyyppien käsittelylle sekä niiden välisille operaatioille. Tässä luvussa esitellään ja määritellään tutkielman kannalta keskeiset tyyppijärjestelmät.

2.1 Staattinen ja dynaaminen tyypitys

Dynaaminen tyypitys ohjelmoinnissa tarkoittaa, että muuttujien tyypit tarkistetaan ohjelman suorituksen aikana. Tämä antaa joustavuutta muuttujien käyttöön, koska tyyppejä ei tarvitse määritellä etukäteen. Dynaamisen tyypityksen käyttö voi kuitenkin johtaa siihen, että tyyppivirheet havaitaan vasta ohjelman suorituksen aikana. [1], [2] Tämä voidaan nähdä dynaamisen tyypityksen huonona puolena, sillä ohjelmointivirheet voidaan havaita vasta pitkän ajan päästä, kun tyyppivirheinen koodi ajetaan.

Staattisessa tyypityksessä muuttujien tyypit puolestaan määritellään ennen ohjelman suoritusta, ja niiden yhteensopivuus varmistetaan jo käännösvaiheessa, ei suorituksen aikana. [1], [2] Tämä on myös staattisen tyypityksen rajoite, sillä ohjelman tyypillinen oikeellisuus on varmistettava aina. Tämä johtaa siihen, että ohjelman kääntäminen voidaan estää, vaikka se olisikin suoritettavissa ilman tyyppivirheitä. [1]

Kuitenkin myös staattisissa kielissä joudutaan usein käsittelemään dataa, jonka tyyppiä ei kääntäessä tiedetä. Kun ollaan vuorovaikutuksessa ulkoisten tietolähteiden, kuten tietokantojen tai muiden ohjelmien kanssa, tarvitaan jonkinasteisia ajonaikaisia tarkistuksia tyyppiturvallisuuden varmistamiseksi, koska tietotyypit ovat tuntemattomia kääntämisaikana. [3] Staattinen tyypitys ei siis pysty suorittamaan tyyppitarkistuksia ohjelman ulkopuolelta tulevaan dataan ennen ohjelman suoritusta.

2.2 Vähittäinen tyypitys

Vähittäinen tyypitys yhdistää ohjelmointikielessä staattisen ja dynaamisen tyypityksen. Se antaa mahdollisuuden hallita tyyppitarkistuksen tiukkuutta antamalla ohjelmoijalle mahdollisuuden valita, mitkä ohjelman osat tarkistetaan käännösaikana ja mitkä suoritusaikana. On yleistä, että ohjelmistojen kehitys aloitetaan dynaamisesti tyypitetyllä kielellä ja kesken kehityksen vaihdetaan staattisesti tyypitettyyn kieleen. Vähittäinen tyypitys mahdollistaakin nopean prototyypin kehityksen dynaamisesti tyypitettynä ja nimensä mukaisesti vähittäisen siirtymisen staattiseen tyypitykseen samaa kieltä käyttäen. [4]

Usein vähittäinen tyypitys toteutetaan kääntämällä tyypitetty ohjelmointikieli johonkin dynaamisesti tyypitettyyn kieleen. Tästä esimerkkeinä ovat tyypitetty Clojure, Gradualtalk, Reticulated Python ja tässä tutkielmassa tarkasteltu TypeScript. [5] Tällaisia kääntäjiä, jotka eivät käännä ohjelmointikieltä konekieleksi, vaan ohjelmointikielestä toiseen ohjelmointikieleen kutsutaan transkääntäjiksi (*transpiler*).

2.3 Vahva ja heikko tyypitys

Vahvalla ja heikolla tyypityksellä ei ole laajalti hyväksyttyä akateemista määritelmää. Kirjallisuudessa määritelmiä ”vahva tyypitys” ja ”staattinen tyypitys” käytetään joskus synonyymeinä [2]. Usein kuitenkin määritellään, että heikossa tyypityksessä tyyppimuunnokset eri tyyppien välillä tehdään implisiittisesti [6], [7]. Tällöin yhteensopimattomat tyypit muunnetaan automaattisesti yhteensopiviksi siten, että niiden välillä voidaan suorittaa joitain operaatioita. Vahvan tyypityksen voidaan ajatella olevan tämän vastakohta. Vahvassa tyypityksessä tyyppejä ei muunneta automaattisesti, vaan ohjelmoijan on itse suoritettava tyyppimuunnokset eksplisiittisesti ennen muuttujien välisiä operaatioita [6], [7]. Näitä määritelmiä käytetään myös tässä tutkielmassa tyypityksen luonteiden tarkastelun tukena.

Vaikka heikko tyypitys yhdistetään usein dynaamisiin ohjelmointikieliin ja vahva tyypitys staattisiin, eivät ne aina esiinny yhdessä. Esimerkiksi Python, joka on dynaaminen ohjelmointikieli, kieltää monet automaattiset tyyppimuunnokset, mikä viittaa vahvaan tyypitykseen. Toisaalta C, joka on staattinen ohjelmointikieli, voi toteuttaa useita automaattisia tyyppimuunnoksia, mikä on tyypillistä heikolle tyypitykselle [8].

3. VERTAILUKRITEERIT

Tutkielmassa vertailtavat osa-alueet on rajattu kolmeen: ohjelmointivirheiden esiintyvyyteen, ohjelmiston ylläpidettävyyteen ja kehitysnopeuteen. Nämä valittiin vertailuun, koska ne ovat kriittisiä ohjelmiston laadun ja kehitysprosessin tehokkuuden kannalta.

ISO/IEC 25010:2011-standardissa määritellään ohjelmistojen laatu ja siinä tunnistetaan kahdeksan päälaatuominaisuutta: toiminnallinen sopivuus, tehokkuus, yhteensopivuus, käytettävyys, luotettavuus, turvallisuus, ylläpidettävyys ja siirrettävyys. [9] Näitä kriteerejä voidaan käyttää ohjelmistojen laadun mittaamiseen.

Ylläpidettävyys on ohjelmiston laadun ulottuvuus, joka viittaa siihen, kuinka helposti ohjelmistoa voidaan muuttaa korjaamaan virheitä, parantamaan suorituskykyä tai muuten päivittämään ohjelmistoa vastaamaan muuttuvia vaatimuksia. Standardissa siihen määritellään kuuluvan modulaarisuus, uudelleenkäytettävyys, analysoitavuus ja testattavuus [9]. Ylläpidettävyyttä voidaan verrata monella tapaa, kuten koodihajujen määrällä [10] tai mittaamalla aikaa, joka ylläpitotehtäviin kuluu [11].

Koodin ymmärrettävyys puolestaan kuvaa, kuinka helposti kehittäjät voivat ymmärtää ohjelmiston koodia. Ymmärrettävyyttä mitataan esimerkiksi koodirivien tai loogisten koodirivien määrällä. Toisena kriteerinä voidaan käyttää McCaben syklomaattista kompleksisuutta, joka laskee mahdolliset itsenäiset polut koodin sisällä. Vuonna 2018 SonarSource julkaisi uutena mittarina kognitiivisen kompleksisuuden. [12] Ohjelmistokehittäjän näkökulmasta koodin ymmärrettävyyden voidaan ajatella olevan osa ohjelmiston ylläpidettävyyttä.

Ohjelmointivirheet viittaavat koodissa oleviin virheisiin, jotka aiheuttavat toimintahäiriöitä tai ohjelmiston toimimista muilla odottamattomilla tavoilla. Ne voivat johtaa esimerkiksi virheilmoituksiin, suorituskyvyn heikkenemiseen, tietoturvaongelmiin tai ohjelmiston kaatumiseen. Ohjelmointivirheet voivat vaikuttaa useaan laatutekijään ja niitä tarkastellaan tässä tutkielmassa irrallaan standardin kriteereistä. Ohjelmointivirheiden esiintyvyyttä voidaan tarkastella esimerkiksi vertaamalla virheiden korjaukseen liittyvien muutosten ja muutosten kokonaismäärän suhdetta [10], [13].

Kehitysnopeus kuvaa nimensä mukaisesti sitä, kuinka nopeasti ohjelmistokehittäjä tai -kehittäjät pystyvät tuottamaan uutta koodia tai päivittämään olemassa olevaa ohjelmistoa. Kehitysnopeus vaikuttaa suoraan ohjelmistoprojektin aikatauluun. Kehityksen ollessa nopeaa tiimit voivat luoda uusia ominaisuuksia sovelluksiin nopeammin ja vastata

paremmin muuttuviin vaatimuksiin. Kehitysnopeutta tutkitaan useimmiten antamalla koehenkilöille suoritettavaksi ohjelmointitehtäviä ja tarkastelemalla niihin kuluvaa aikaa [11], [14], [15].

4. VERTAILTAVAT OHJELMOINTIKIELET

Tässä luvussa esitellään työssä vertailtavat ohjelmointikielet eli JavaScript ja TypeScript. Kerrotaan kielten historiasta ja ominaisuuksista, mihin niitä käytetään ja minkä tyyppijärjestelmien piirteitä niissä on.

4.1 JavaScript

Brendan Eich kehitti ensimmäisen version JavaScriptistä vuonna 1995. Netscape Communications Corporation ja Sun Microsystems julkaisivat kielen 4. joulukuuta 1995. Se luotiin ensisijaisesti käytettäväksi asiakaspuolen skriptikielenä verkkosivustoilla ja otettiin ensimmäisen kerran käyttöön Netscape Navigator-selaimessa, joka oli tuolloin suosituin selain. [16], [17]

JavaScript tunnettiin ensimmäisen version aikaan nimellä Mocha. Kielen nimeksi vaihdettiin LiveScript osana yhteistyötä Sun Microsystemsin kanssa, ja pian sen jälkeen se nimettiin uudelleen JavaScriptiksi, koska markkinoinnissa haluttiin hyödyntää Javan suosiota. JavaScript ja Java ovat kuitenkin teknisesti kaukana toisistaan. Tavaramerkkiongelmien vuoksi JavaScriptin standardisoitu versio nimettiin ECMAScriptiksi. [17] Nykyään JavaScript on yksi maailman suosituimmista ohjelmointikielistä, ja arviolta jopa 95 % verkkosivuista käyttää sitä jossain muodossa. JavaScriptiä käytetään myös esimerkiksi palvelinpuolen kehityksessä ja perinteisissä työpöytä- ja mobiilisovelluksissa. [16]

JavaScript on dynaamisesti tyypitetty ja tulkattu ohjelmointikieli. Dynaamisen tyypityksen lisäksi JavaScriptissä monia yhteensopimattomia muuttujia muunnetaan implisiittisesti eli käytetään heikkoa tyypitystä. [7] Alla on esimerkkiohjelma JavaScriptin heikosta ja dynaamisesta tyypityksestä käytännössä.

```
1   // Dynaamisesti tyypitetyt muuttujat
2   let numero = 10;
3   let merkkijono = 5;
4
5   console.log(typeof merkkijono) // Tuloste: number
6   merkkijono = '5'; // Vaihdetaan muuttujan tyyppi merkkijonoksi
7   console.log(typeof merkkijono) // Tuloste: string
8
9   // Implisiittinen tyyppimuunnos numeroksi aritmeettisessa operaatiossa
10  let tulos = numero - merkkijono; // Tulos: 5
```

Ohjelma 1. *Dynaaminen ja heikko tyypitys JavaScriptissä*

JavaScript vie heikon tyypityksen kuitenkin hyvin pitkälle. Mikäli esimerkiksi suoritetaan yhteenlaskuoperaatio numeron ja listan välillä, muuntaa JavaScript molemmat merkkijonoksi ja antaa tuloksena merkkijonon, kuten alla olevassa ohjelmassa. [7]

```
3 + [1]; // Tulos: "31"

(3).toString() + [1].toString() // Tulos: ”31"
```

Ohjelma 2. *Ongelmaesimerkki heikosta tyypityksestä JavaScriptissä [7]*

JavaScript mahdollistaa yhdessä Document Object Modelin (DOM) kanssa verkkosivujen elementtien dynaamisen muokkaamisen asiakaspuolella. DOM on web-dokumenttien ohjelmointirajapinta. DOM:ia käyttämällä ohjelmoijat voivat hallinnoida verkkosivun sisältöä, rakennetta ja tyylejä JavaScriptin avulla, mahdollistaen HTML-elementtien lisäämisen, muokkaamisen ja poistamisen.

JavaScript mahdollistaa myös datan haun palvelimelta AJAX-tekniikoiden avulla. AJAX on lyhenne sanoista Asynchronous JavaScript And XML. Se on joukko web-kehitystekniikoita, joiden avulla verkkosivut voivat kommunikoida palvelimen kanssa ilman, että koko sivua tarvitsee ladata uudelleen. Tämä parantaa verkkosivujen käyttäjäkokemusta ja tekee niistä enemmän perinteisten sovellusten kaltaisia. [18]

4.2 TypeScript

Microsoft julkaisi TypeScriptin vuonna 2012 vaihtoehtona JavaScript-kehitykselle suuria sovelluksia varten [19]. TypeScript on JavaScriptin ylijoukko, mikä tarkoittaa, että syntaksiltaan kaikki JavaScript-ohjelmat ovat myös kelvollisia TypeScript-ohjelmia.

TypeScript on staattisen ja vahvan tyypityksen omaava versio JavaScriptistä. Vahvan tyypityksen ansiosta TypeScript ei useimmissa tapauksissa salli implisiittisiä tyyppimuunnoksia. TypeScript-kääntäjä estäisi esimerkiksi JavaScript-kappaleessa esitellyn ohjelman, jossa suoritettiin yhteenlaskuoperaatio kokonaisluvulle ja listalle, kääntämisen ilman eksplisiittisiä tyyppimuunnoksia toString()-metodilla. [7] Alla esimerkkiohjelma, jossa esitellään staattisen ja vahvan tyypityksen käyttöä TypeScriptissä.

```
// Staattisesti tyypitetyt muuttujat
let numero: number = 10;
let merkkijono: string = '5';
merkkijono = 5; // Tyyppi 'number' ei ole määritettävissä tyypille ...

// Implisiittinen tyyppimuunnos epäonnistuu:
let tulos = numero - merkkijono; // Aritmeettisen operaation oikean
// puolen on oltava tyyppiä 'any', 'number', 'bigint' tai ...

// Vahvan tyypityksen vaatima eksplisiittinen tyyppimuunnos:
let tulos = numero - Number(merkkijono); // Tulos: 5
```

Ohjelma 3. *Staattinen ja vahva tyypitys TypeScriptissä*

Tyyppitarkistus voidaan ohittaa käyttämällä any-tyyppiä, jolloin muuttuja käyttäytyy perinteisen dynaamisesti tyypitetyn JavaScriptin tavoin. TypeScript toteuttaa staattisen tyypityksen vähittäisen tyypityksen kautta. [7] TypeScript toteuttaa vähittäisen tyypityksen siten, että se käännetään dynaamisesti tyypitettyyn kieleen [5], eli tässä tapauksessa JavaScriptiin.

5. VERTAILU

Tässä luvussa vertaillaan JavaScriptin ja TypeScriptin ominaisuuksien vaikutusta ohjelmistokehityksen eri osa-alueisiin olemassa olevan kirjallisuuden pohjalta. Tarkastellaan myös kirjallisuutta yleisesti liittyen aiemmin esiteltyihin tyyppijärjestelmiin.

5.1 Ohjelmointivirheiden esiintyvyys

Ocariza Jr. et al. kertovat tutkimuksessaan, että JavaScript on ohjelmointivirheille altis kieli. Tutkimuksen tarkoituksena oli selvittää, mistä JavaScriptin alttius ohjelmointivirheille johtuu ja miten se vaikuttaa ohjelmoijiin. Tutkimuksessa 68 % havaituista ohjelmointivirheistä liittyivät JavaScriptin vuorovaikutukseen Document Object Modelin (DOM) kanssa. DOMiin liittyvistä virheistä 38 % oli myös tyyppivirheitä. Kaikista virheistä tyyppivirheitä oli 33 %. Näistä tyyppivirheistä 72 % oli tilanteita, joissa ohjelma odotti jotain JavaScriptin natiivia asiakaspuolen luokkatyyppiä mutta sai arvon *null* (muuttujalla ei ollut arvoa ja se oli tarkoituksellisesti tyhjä) tai *undefined* (muuttuja oli määritelty, mutta sille ei ollut asetettu arvoa). Esimerkkeinä natiiveista luokkatyypeistä annettiin Function ja Element. Kerrottiin, etteivät tyypintarkastajat voi yleisesti tietää JavaScriptin sisäänrakennettujen metodien paluuarvoja eikä melkein mikään tyypintarkastaja tarkastele DOM:ia, joten siihen liittyviä tyyppivirheitä ei voida ohjelmaa kääntäessä tarkistaa. Muun muassa edellä mainittujen syiden takia pääteltiin, etteivät TypeScript tai muut vahvan tyyppijärjestelmän omaavat kielet välttämättä poista suurinta osaa JavaScriptiin liittyvistä ongelmista, vaikka niiden muita hyötyjä kuitenkin tiedostettiin. [20]

J. Bognerin ja M. Merkelin tutkimuksessa suoritettiin laatuvertailu 604 GitHub-projektista, joissa oli yli 16 miljoonaa koodiriviä. Projekteista 299 käyttivät JavaScriptiä ja 305 TypeScriptiä. Havaittiin odotusten vastaisesti, että TypeScript-projekteissa ohjelmointivirheiden korjaukseen vaadittiin noin 33 päivää verrattuna JavaScriptin noin 32 päivään ja TypeScript-projekteissa 20,6 % commiteista liittyi ohjelmointivirheiden korjauksiin, kun taas JavaScript-projekteissa 12,6 %. Havaintojen kerrottiin viittaavat siihen, että TypeScriptin vaikutus koodin virheherkkyyteen voi olla aiemmin ajateltua monimutkaisempi. [10]

Z. Gaon et al. tutkimuksessa lisättiin tyypit ohjelmointivirheitä sisältävälle JavaScript-koodille TypeScriptillä ja toisella JavaScriptin staattisella tyypintarkistimella Flowilla siinä toivossa, että nämä antaisivat virheilmoituksen, jonka avulla ohjelmointivirheet voidaan korjata. Tutkielmassa kiinnitettiin huomiota siihen, että tämän tyyppinen arviointi aliarvioi

ohjelmointivirheiden havaitsemisen normaalissa kehityksessä, sillä arvioitu koodi on julkista ja jo kertaalleen arvioitua ja hyväksyttyä. Flow ja TypeScript havaitsivat tästä huolimatta ohjelmointivirheistä 15 %. [21]

B. Rayn et al. tutkimuksessa suoritettiin GitHub-laatuvertailu 728 projektille. Yhteensä tarkasteltiin 63 miljoonaa lähdekoodiriviä ja 17 eri ohjelmointikieltä. Virheiden esiintyvyyttä tarkasteltiin vertailemalla ohjelmointivirheiden korjaamiseen liittyvien committien ja committien kokonaismäärän suhdetta. Tulokset viittasivat siihen, että staattisesti tyypitetyt kielet ovat vähemmän virhealttiita kuin dynaamisesti tyypitetyt, erityisesti funktionaalisissa kielissä. Vahva tyypitys myös suoriutui heikkoa tyypitystä paremmin. Tämän tutkielman kontekstissa on hyvä huomioida, että TypeScript-projektit sisälsivät suhteessa committien määrään kaikista kielistä vähiten virheenkorjauksiin liittyviä committeja. TypeScriptin osalta kuitenkin korostettiin, että vaikka se on tarkoitettu käytettäväksi staattisesti tyypitettynä kielenä, käyttävät kehittäjät jopa noin puolissa muuttujista heikon ja dynaamisen tyypityksen sallivaa any-tyyppiä. [13]

5.2 Ohjelmiston ylläpidettävyys

J. Bognerin ja M. Merkelin GitHub-laatuvertailututkimuksessa koodin laatua tutkittiin muun muassa koodihajuilla. Koodihajut ilmoittavat koodin matalasta laadusta, joka voi vaikuttaa koodin ylläpidettävyyteen. Koodihajujen etsimiseen käytettiin staattisen analyysin työkalua SonarQubea. 7 496 726 JavaScript-koodirivissä havaittiin 217 957 koodihajua ja 8 683 600 TypeScript-koodirivissä 95 132. Tuhatta koodiriviä kohden JavaScript-ohjelmissa oli noin 12 koodihajua enemmän kuin TypeScript-ohjelmissa. Tultiin tulokseen, että TypeScript parantaa merkittävästi koodin laatua verrattuna JavaScriptiin. Havaittiin myös, TypeScriptin any-tyypin käyttö nosti koodihajujen määrää. Myös koodin ymmärrettävyyttä mitattiin SonarQubella tarkastelemalla koodin kognitiivista kompleksisuutta. TypeScript paransi ohjelmiston laatua tästäkin näkökulmasta, ja any-tyypin käyttö jälleen huononsi koodin ymmärrettävyyttä. [10] Lavazza et al. kuitenkin väittävät, ettei koodin kognitiivinen kompleksisuus merkittävästi ennusta ymmärrettävyyttä paremmin verrattuna perinteisiin mittareihin, kuten McCabenin syklomaattiseen kompleksisuuteen ja loogisten koodirivien määrään. [12]

S. Kleinschmagerin et al. tutkimuksessa suoritettiin koe, jossa kehittäjille annettiin ohjelmointitehtäviä, joita kuvailtiin mahdollisiksi ohjelmiston ylläpitotehtäviksi. Ylläpidettävyyttä arvioitiin tarkastelemalla, kuinka pitkään tehtävien suorittamisessa kestää. Tehtävät suoritettiin sekä staattisesti että dynaamisesti tyypitetyissä ympäristöissä. Kun tehtävät vaativat uusien, vain lähdekoodissa dokumentoitujen luokkien käyttöä, vähensi staattinen tyypitys tehtävään tarvittua aikaa neljässä viidestä tehtävästä. Staattinen tyypitys

nopeutti tyyppivirheiden korjaamista merkittävästi. Semanttisten, eli loogisten virheiden, jotka eivät estä ohjelman suoritusta korjaamisessa ei havaittu eroa. Staattiset tyyppijärjestelmät nopeuttivat tehtävien suorittamista, joten pääteltiin, että staattiset tyyppijärjestelmät vaikuttavat positiivisesti ohjelmistojen ylläpidettävyyteen. [22]

5.3 Kehitysnopeus

Staattisen tyypityksen eduiksi katsotaan parempi koodieditorien tarjoama tuki ja tyyppivirheiden havaitseminen varhaisessa vaiheessa, jolloin jälkikäteen korjaamiseen ei tarvitse käyttää aikaa. Toisaalta dynaamisen tyypityksen omaavien kielten ajatellaan usein olevan nopeampia prototyyppien kehityksessä niiden joustavuuden ansiosta.

A. Stuchlik ja S. Hanenbergin tutkimuksessa tutkittiin tyyppimuunnosten vaikutusta kehitysnopeuteen staattisissa ja dynaamisissa tyyppijärjestelmissä. Tutkimuksessa 21 koehenkilölle annettiin ohjelmointitehtäviä suoritettavaksi Javalla sekä Groovyllä. Groovyn staattisesti tyypitetyt ominaisuudet eivät olleet käytössä, eli sitä käytettiin dynaamisesti tyypitettynä versiona Javasta. Koehenkilöt olivat kaikki ohjelmoineet Javalla mutta eivät Groovyllä. Tehtävät olivat pieniä ohjelmointitehtäviä, jotka vaativat tietyn määrän tyyppimuunnoksia. Jokaiselle koehenkilölle tehtäviä annettiin suoritettavaksi viisi, ja staattisesti tyypitetyissä ympäristöissä näihin kuluneen ajan mediaani oli alle 100 minuuttia. Dynaaminen tyypitys vähensi kaikkiin tehtäviin tarvittua aikaa 8:sta 37 minuuttiin, joten katsottiin, että dynaaminen tyyppijärjestelmä nopeutti tehtävien suorittamista keskimäärin merkittävästi. Koska dynaamisissa tyyppijärjestelmissä säästettiin niin paljon aikaa, pääteltiin, että tyyppimuunnokset vaativat kehittäjiltä älyllistä työtä pelkkien tyyppimuunnosten kirjoittamisen lisäksi. Tutkimuksessa kuitenkin väitettiin, että suuremmissa ohjelmointitehtävissä ja ohjelmistoissa eroa ei havaita, ja tyyppimuunnosten vaikutus jää muiden tekijöiden varjoon. [11]

L. Fischerin ja S. Hanenbergin tutkimuksessa tukittiin koodin täydentämisen ja tyyppijärjestelmien vaikutusta rajapintojen käytettävyyteen Microsoft Visual Studiossa. Satunnaistetussa kokeessa 24 koehenkilöä suoritti ohjelmointitehtäviä JavaScriptillä ja TypeScriptillä, sekä koodintäydennyksen kanssa että ilman. Viisi koehenkilöä poistettiin otannasta, koska he joko keskeyttivät jonkun tehtävän tai eivät saaneet sitä suoritettua takarajaksi asetetussa 70 minuutissa. Tutkimuksessa tyypitys paransi merkittävästi kehitysnopeutta, kun taas koodintäydennys nopeutti tehtävien suorittamista vain vähäisesti joissain tapauksissa. [14]

S. Hanenbergin tutkimus keskittyy staattisen tyyppijärjestelmän vaikutukseen ohjelmistokehityksessä. Tutkimusta varten kehitettiin uusi, Purity-niminen ohjelmointikieli sekä

staattisesti että dynaamisesti tyypitettyinä versioina, ja koehenkilöt opetettiin käyttämään sitä. Tutkimukseen osallistui 49 koehenkilöä ja heitä pyydettiin kehittämään ensin yksinkertainen selaaja, joka poistaa erikoismerkit sanasta ja sen jälkeen yksinkertaistettu Java-jäsennin. Tutkimus siis koostui kahdesta osasta, pienemmästä selaajatehtävästä ja molemmat tehtävät sisältävästä kokonaisuudesta. Havaittiin, että dynaamista tyypitystä käyttäneet koehenkilöt suorittivat pienemmän tehtävän nopeammin, kun taas molemmat osat sisältävässä kokonaistehtävässä ei havaittu eroa. Tämän pohjalta esitettiin, että staattisen tyyppijärjestelmän hyödyt saattoivat nousta esiin tehtävän myöhemmässä vaiheessa, vaikka dynaaminen tyyppijärjestelmä auttoikin suorittamaan pienen tehtävän nopeammin. Staattisella tyyppijärjestelmällä ei kuitenkaan havaittu positiivista vaikutusta kehitysaikaan. [15]

6. TULOKSET

Kaksi kolmesta tutkimuksesta, joissa käsiteltiin ohjelmointivirheiden esiintyvyyttä, tukivat TypeScriptin ja staattisten tyypintarkastajien käyttöä virheiden esiintyvyyden näkökulmasta [13], [21]. Toisessa näistä tutkimuksista ohjelmointivirheitä sisältävälle JavaScript-koodille lisättiin tyypit staattisilla tyypintarkistimilla, jolloin 15 % virheistä havaittiin [21]. Tämä tukee oletusta, että staattiset tyypintarkastajat havaitsevat virheitä varhaisessa vaiheessa. Toisaalta suuri osa JavaScriptin virheistä liittyi DOM:in käsittelyyn, joten TypeScriptin tarjoaman tyypintarkastelun hyödyt eivät välttämättä nouse esiin niin hyvin DOM-rajapintaa hyödyntävissä web-sovelluksissa [20]. Yhdessä tutkimuksessa myös havaittiin suurempi määrä ohjelmointivirheitä TypeScript- kuin JavaScript-sovelluksissa [10]. Tulokset ovat ristiriitaisia, mutta TypeScript vaikuttaisi kokonaisuudessaan parantavan ohjelmistoja ohjelmointivirheiden esiintyvyyden näkökulmasta.

TypeScript JavaScriptiin verrattuna myös vaikutti parantavan ohjelmistojen ylläpidettävyyttä, kun tarkasteltiin koodihajujen määrää ja koodin ymmärrettävyyttä. TypeScriptin any-tyypin käyttö myös huononsi ohjelmistojen laatua molemmista näkökulmista. [10] Staattinen tyypitys nopeutti ylläpitotehtävien suorittamista [22]. TypeScript ja staattiset tyypintarkastajat vaikuttavat vaikuttavan positiivisesti ohjelmistojen ylläpidettävyyteen. Tulee kuitenkin huomioida, että ylläpidettävyyden osalta tarkasteltiin vain kahta tutkimusta.

Dynaaminen ja heikko tyypitys nopeuttivat pienten, tyyppimuunnoksia vaativien ohjelmointitehtävien suorittamista [11]. Yhdessä tutkimuksista dynaaminen tyypitys nopeutti tehtävän suorittamista tehtävän pienessä osassa, mutta kokonaistehtävässä ei havaittu eroa staattiseen tyypitykseen [15]. TypeScript puolestaan nopeutti isommissa, korkeintaan 70 minuuttia kestävissä ohjelmointitehtävissä tehtävien suorittamista merkittävästi [14]. JavaScript vaikuttaa nopeuttavan ohjelmistokehitystä pienissä tehtävissä ja TypeScript puolestaan suuremmissa tehtävissä.

Vertailukriteerien erottelu vaikeutti kielten vertailua. Kriteerien välillä on myös todennäköisesti korrelaatiota. Esimerkiksi useat ohjelmiston laatutekijät vaikuttavat todennäköisesti ohjelmistojen ylläpidettävyyteen, mutta ylläpidettävyyden mainitsevia laatuvertailuja löytyi rajatusti.

7. YHTEENVETO

Tässä tutkielmassa toteutettiin kirjallisuuskatsaus, jonka tavoitteena oli tutkia TypeScriptin vaikutusta ohjelmistoihin ohjelmointivirheiden esiintyvyyden, ohjelmistojen ylläpidettävyyden ja kehitysnopeuden näkökulmista. Tutkimusten perusteella voidaan päätellä, että TypeScriptin käyttö ohjelmistokehityksessä tuo hyötyjä erityisesti ohjelmointivirheiden vähenemisenä ja ohjelmiston ylläpidettävyyden parantumisena. TypeScriptin käyttö vaikuttaisi olevan perusteltua lähes kaikissa ohjelmistoprojekteissa, sillä kokonaisuudessaan dynaamiset tyyppijärjestelmät suoriutuivat paremmin ainoastaan kehitysnopeuden osalta pienten ohjelmointitehtävien suorittamisessa, ainakin kun tehtävät vaativat tyyppimuunnoksia. TypeScript puolestaan nopeutti isompien tehtävien suorittamista.

Oletus siitä, että prototyyppien kehittäminen dynaamisella tyypityksellä on nopeampaa vaikuttaa oikealta. Koska TypeScript sallii vähittäisen tyypityksen ansiosta sekä vahvan ja staattisen että heikon ja dynaamisen tyypityksen, voidaan projektin alkuvaiheessa kehittää prototyyppi nopeasti heikommilla tyypintarkastuksilla ja muuntaa koodi myöhemmin ylläpidettävämmäksi. TypeScriptin käyttö ohjelmistoprojekteissa JavaScriptin sijaan vaikuttaa perustellulta etenkin suuremmissa ohjelmistoprojekteissa. JavaScriptin käyttö voi olla perustellumpaa pienissä projekteissa tai tehtävissä heikon ja dynaamisen tyypityksen joustavuuden ansiosta, mutta tyypintarkistimien muut hyödyt tukevat TypeScriptin käyttöä niissäkin. TypeScriptiä käyttäessä olisi myös hyvä pyrkiä minimoimaan dynaamisen tyypityksen sallivan any-tyypin käyttö.

LÄHTEET

[1] K. B. Bruce, *Foundations of object-oriented languages: types and semantics*. Cambridge, Mass: MIT Press, 2002.

[2] B. Eckel, 'Strong Typing vs. Strong Testing', in *The Best Software Writing I*, Apress, 2005, pp. 67–77. doi: 10.1007/978-1-4302-0038-3_11.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, 'Dynamic typing in a statically typed language', *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, pp. 237–268, Apr. 1991, doi: 10.1145/103135.103138.

[4] J. Siek and W. Taha, 'Gradual Typing for Objects', in *ECOOP 2007 – Object-Oriented Programming*, vol. 4609, E. Ernst, Ed., in Lecture Notes in Computer Science, vol. 4609. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 2–27. doi: 10.1007/978-3-540-73589-2_2.

[5] M. M. Vitousek, C. Swords, and J. G. Siek, 'Big types in little runtime: open-world soundness and collaborative blame for gradual type systems', *ACM SIGPLAN Not.*, vol. 52, no. 1, pp. 762–774, May 2017, doi: 10.1145/3093333.3009849.

[6] 'Typing: Strong vs. Weak, Static vs. Dynamic'. Accessed: Apr. 07, 2024. [Online]. Available: https://www.artima.com/weblogs/viewpost.jsp?thread=7590

[7] B. Cherny, *Programming TypeScript*. O'Reilly Media, Inc., 2019.

[8] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Englewood Cliffs, N.J: Prentice-Hall, 1978.

[9] 'ISO/IEC 25010:2011: Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models.'

[10] J. Bogner and M. Merkel, 'To type or not to type?: a systematic comparison of the software quality of JavaScript and typescript applications on GitHub', in *Proceedings of the 19th International Conference on Mining Software Repositories*, Pittsburgh Pennsylvania: ACM, May 2022, pp. 658–669. doi: 10.1145/3524842.3528454.

[11] A. Stuchlik and S. Hanenberg, 'Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time', *ACM SIGPLAN Not.*, vol. 47, no. 2, pp. 97–106, Mar. 2012, doi: 10.1145/2168696.2047861.

[12] L. Lavazza, A. Z. Abualkishik, G. Liu, and S. Morasca, 'An empirical evaluation of the "Cognitive Complexity" measure as a predictor of code understandability', *J. Syst. Softw.*, vol. 197, p. 111561, Mar. 2023, doi: 10.1016/j.jss.2022.111561.

[13] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, 'A large scale study of programming languages and code quality in github', in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong China: ACM, Nov. 2014, pp. 155–165. doi: 10.1145/2635868.2635922.

[14] L. Fischer and S. Hanenberg, 'An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS visual studio', *ACM SIGPLAN Not.*, vol. 51, no. 2, pp. 154–167, May 2016, doi: 10.1145/2936313.2816720.

[15] S. Hanenberg, ‘An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time’, *ACM SIGPLAN Not.*, vol. 45, no. 10, pp. 22–35, Oct. 2010, doi: 10.1145/1932682.1869462.

[16] ‘History of JavaScript’, GeeksforGeeks. Accessed: Apr. 09, 2024. [Online]. Available: https://www.geeksforgeeks.org/history-of-javascript/

[17] A. Wirfs-Brock and B. Eich, ‘JavaScript: the first 20 years’, *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, pp. 1–189, Jun. 2020, doi: 10.1145/3386327.

[18] P. Wilton and J. McPeak, *Beginning JavaScript*, 4. ed. in Wrox Programmer to Programmer. Indianapolis, Ind: Wiley, 2010.

[19] J. Turner [MS, ‘Announcing TypeScript 1.0’, TypeScript. Accessed: Apr. 09, 2024. [Online]. Available: https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/

[20] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, ‘A Study of Causes and Consequences of Client-Side JavaScript Bugs’, *IEEE Trans. Softw. Eng.*, vol. 43, no. 2, pp. 128–144, Feb. 2017, doi: 10.1109/TSE.2016.2586066.

[21] Z. Gao, C. Bird, and E. T. Barr, ‘To Type or Not to Type: Quantifying Detectable Bugs in JavaScript’, in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires: IEEE, May 2017, pp. 758–769. doi: 10.1109/ICSE.2017.75.

[22] S. Kleinschmager, S. Hanenberg, R. Robbes, and A. Stefik, ‘Do static type systems improve the maintainability of software systems? An empirical study’, in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, Passau, Germany: IEEE, Jun. 2012, pp. 153–162. doi: 10.1109/ICPC.2012.6240483.