Over de understandability van subtype polymorfisme in objectgeoriënteerde systemen

Nico Schoenmaker
6026052

30 augustus 2010

Master Software Engineering

Afstudeerdocent: Jurgen Vinju

Centrum voor Wiskunde en Informatica

Universiteit van Amsterdam

Samenvatting

Door het gebruik van polymorfisme in objectgeoriënteerde systemen ontstaat delokalisatie, waarbij conceptueel gerelateerde code verspreid is over een systeem. Deze delokalisatie kan een probleem vormen voor de understandability. Polymorfisme kan worden vervangen door conditionele logica, maar de complexiteit die dit toevoegt kan ook een negatieve invloed hebben op de understandability. Uit een gecontroleerd experiment blijkt dat deze complexiteit de understandability verlaagt. Gebruik van conditionele logica is dus een slecht alternatief voor de delokalisatie.

1 Inleiding

Een belangrijk aspect van software maintenance is het begrijpen van de software. Hier besteed een programmeur tot wel 50% van zijn tijd aan [13, 3]. *Software comprehension* is het proces van het verkrijgen van een begrip van hoe een software systeem werkt [36].

Het mentaal uitvoeren van broncode is een bekende strategie om een programma te begrijpen [14]. Wanneer in broncode `GOTO` statements worden gebruikt, wordt het moeilijker om het gedrag van een programma te voorspellen [12]. De understandability van broncode is in hoeverre deze broncode invloed heeft op software comprehension.

Op een soortgelijke manier als `GOTO` statements, beïnvloedt ook *subtype polymorfisme* het verschil tussen broncode en executie [23]. Subtype polymorfisme is een vorm van polymorfisme waarbij een datatype (het subtype) is gerelateerd aan een ander datatype (het supertype) met een notie van vervangbaarheid. Dit betekent dat statements die uitgaan van het supertype ook instanties van het subtype accepteren. In dit document wordt met polymorfisme, subtype polymorfisme bedoeld.

Ondanks dat polymorfisme problemen voor de understandability kan opleveren, wordt dit bij enkele refactorings in de doelsituatie gebruikt. *Refactoren* is het herstructureren van objectgeoriënteerde programma's zonder de werking ervan te veranderen.

Martin Fowler introduceert in zijn boek de refactoring "*Replace conditional with Polymorphism*", welke conditionele logica vervangt door polymorfisme [15]. Fowler introduceert vaak paren van refactorings, waarbij refactoring I de inverse is van refactoring II. Bij "*Replace conditional with Polymorfism*" wordt echter geen inverse gegeven, alsof deze overbodig is. Bij de motivatie van de refactoring geeft Fowler alleen voordelen:

> *The biggest gain occurs when the same set of conditionals appears in many places in the program. If you want to add a new type you have to find and update all the conditionals.*

Het gebruik van polymorfisme levert voordelen op bij het bewerken van het programma, nieuwe types kunnen makkelijk worden toegevoegd. In tegenstelling tot het gebruik van conditionele logica kunnen er nieuwe types worden toegevoegd zonder toegang tot de broncode van een programma te hebben.

Polymorfisme zorgt echter voor *delocalized plans*. Een delocalized plan ontstaat wanneer stukken code, die conceptueel gerelateerd zijn, verspreid zijn over een systeem [30]. Ook zorgt het gebruik van polymorfisme ervoor dat er niet met statische analyse kan worden bepaald welke methoden worden aangeroepen. Dit heeft dus negatieve gevolgen voor de understandability van het systeem.

De daadwerkelijke gevolgen van polymorfisme op de understandability van een systeem zijn echter nog onduidelijk, uit onderzoeken hiernaar komen namelijk tegenstrijdige resultaten. Zo kostte het maken van een aanpassing in [10] minder tijd voor een systeem met subtyping, terwijl er in [6] voor hetzelfde systeem, met dezelfde onderzoeksmethode, omgekeerde resultaten kwamen. Een volgend onderzoek kwam tot de conclusie dat subtyping niet het probleem is, maar dat delokalisatie het probleem kan zijn [24].

1.1 Onderzoeksvraag

Het doel van dit onderzoek is om het gebruik van een belangrijk objectgeoriënteerd principe, polymorfisme, te beoordelen op het gebied van understandability. De onderzoeksvraag is:

> Is het belangrijk om de delokalisatie die ontstaat door polymorfisme mee te nemen bij het beoordelen van de understandability van source code?

De deelvragen zijn:

1. Wat is de invloed van delokalisatie, die ontstaat door polymorfisme, op de understandability van een objectgeoriënteerd systeem?

2. Wat is de invloed van de expertise van een programmeur tijdens het begrijpen van systemen waarin veel polymorfisme wordt gebruikt?

Om dit te onderzoeken wordt de understandability van een polymorfe implementatie vergeleken met de understandability van een conditionele implementatie, waarbij de bestede tijd en behaalde correctheid worden gemeten.

1.2 Structuur

De verdere structuur van het paper is als volgt: In hoofdstuk twee wordt een overzicht gegeven van de huidige inzichten betreffende program understanding. Hoofdstuk drie introduceert de onderzoeksopzet. Hoofdstuk vier presenteert de resultaten van het onderzoek, welke in hoofdstuk vijf worden geïnterpreteerd. Tot slot staat in hoofdstuk zes de conclusie.

2 Achtergrond

Om beter te begrijpen wat de understandability van polymorfisme is, dient eerst worden begrepen hoe mensen software begrijpen.

Hieronder wordt eerst een link gelegd met het begrijpen van teksten, de broncode van een programma is immers ook een tekst. Vervolgens wordt beschreven hoe programma's worden begrepen. Daarna wordt het effect van het ontwerp van een programma en de invloed van het doel van het uitvoeren van een taak beschreven.

2.1 Teksten begrijpen

Een *situation model*, ook wel *mentaal model* genoemd, is het beeld van een persoon van een tekst. Een *cognitief model* beschrijft de cognitieve processen welke worden gebruikt om het situation model op te bouwen. De processen van een cognitief model ontwikkelen, op basis van de kennis van de programmeur, de broncode en de documentatie, het situation model.

Het situation model wordt tijdens het lezen bijgewerkt, om zo nieuw binnengekomen informatie mee te nemen. Hoe groter de verandering in het situation model, hoe meer tijd de lezer nodig heeft om de nieuwe situatie te begrijpen.

Relaties leggen Een tekst bevat op zichzelf niet alle informatie welke nodig is om deze te begrijpen, pas zodra een tekst wordt gecombineerd met informatie uit het langetermijngeheugen kan deze worden begrepen. Een voorbeeld:

"Gerhard Schroeder is in front of some journalists. Looking forward to new ideas is nothing special for the Ex-German chancellor. It is like in the good old days in 1971 when the leader of the Jusos was behind the polls and talked about changes."

Deze zin kan alleen worden begrepen als begrepen is dat met "Gerhard Schroeder", "Ex-German chancellor" en "the leader of the Jusos in 1971" dezelfde persoon wordt bedoeld. Het situation model wordt opgebouwd op basis van Gerhard Schroeder, waarbij alle andere informatie wordt gerelateerd aan hem. Hierbij wordt gebruik gemaakt van het langetermijngeheugen, waar bijvoorbeeld is opgezocht dat met de "Ex-German chancellor" Gerhard Schroeder wordt bedoeld. In het situation model zijn deze relaties gelegd.

Bij het leggen van de relaties maakt het niet uit wat de bron van de informatie is. Zo geven respondenten welke een film hebben gekeken soortgelijke antwoorden als studenten welke een tekst hebben gelezen met dezelfde inhoud [2]. Voor het begrijpen van programma's betekent dit bijvoorbeeld dat de inhoud van diagrammen wordt gerelateerd aan de broncode, om samen één situation model te vormen.

Invloed van domeinkennis Domeinkennis fungeert als kapstok om informatie over de tekst aan op te hangen. Zo bleek dat jonge kinderen met veel voetbalkennis zich meer van een tekst over voetbal herinnerden dan oudere kinderen met minder voetbalkennis [27]. De invloed van domeinkennis is bij het opbouwen van het situation model dus groter dan die van het leeftijdsverschil.

Ontwikkelen van het situation model Hoe wordt op basis van de tekst het situation model opgebouwd? Volgens van Dijk & Kintsch kent tekst

drie representaties [32]. Allereerst is er de *surface form*, dit is de tekst welke wordt gelezen. Ten tweede wordt de *text base* opgebouwd, dit is een taalkundige representatie van de tekst. De text base bevat alleen de elementen en relaties welke van de tekst zelf kunnen worden afgeleid. Op basis van de text base wordt het *situation model* gevormd.

Beperkingen Het kortetermijngeheugen kan maar een bepaalde hoeveelheid informatie aan [19]. *Cognitive load* refereert naar de huidige "druk" op het kortetermijngeheugen. Cognitive load bestaat uit een deel wat afhankelijk is van de complexiteit van de te begrijpen informatie, en een deel wat wordt veroorzaakt door de manier van presenteren. Door een tekst anders op te schrijven kan de cognitive load dus veranderen.

Baumann en Krems observeerden dat respondenten met een hogere cognitive load een incompleet situation model opbouwden [4]. Een hoge cognitive load kan de opbouw van een situation model dus beïnvloeden.

2.2 Programma's begrijpen

In 1976 onderzochten Shneiderman en Mayer het gedrag van programmeurs, waarbij een programma werd getoond en respondenten werden gevraagd om, zonder het programma nogmaals te bekijken, het programma op te schrijven [28]. Hieruit bleek dat de programma's van ervaren programmeurs alleen syntactische verschillen hadden, maar semantisch hetzelfde waren. Programmeurs onthouden dus een abstractie van de exacte statements, namelijk wat het doel van de statements is. Dit wijst op de aanwezigheid van cognitieve processen, welke een mentaal model opbouwen van het programma.

Er zijn verschillende opvattingen over de volgorde van het begrip. Volgens de bottom up modellen begint het begrip bij de broncode, en wordt gewerkt naar een begrip van het hele programma. Bij top down begint een programmeur vanuit het doel van het programma te redeneren. Hieronder worden verschillende cognitieve modellen beschreven, onderverdeeld in bottom up en top down.

2.2.1 Bottom up

Het cognitieve model van Shneiderman en Mayer maakt onderscheid tussen syntactische en semantische kennis [29]. Hierbij is syntactische kennis taalspecifiek, terwijl semantische kennis algemeen is. Voorbeelden van semantische kennis zijn weten wat een assignment statement doet, of weten hoe de elementen van een array op te tellen. Volgens dit model gebruikt een programmeur tijdens het begrijpen van een programma zijn syntactische kennis om in semantische termen het programma te beschrijven.

Het cognitieve model van Pennington relateert zich aan het model van Shneiderman en Mayer in de zin dat deze uitgaat van een semantische representatie welke bottom up wordt opgebouwd. Ze onderscheidt twee soorten kennis, *text structure* en *plan knowledge*. *Text structure* bevat kennis van de control flow en conditionele logica. Op basis van de text structure wordt de *plan knowledge* opgebouwd, deze bevat de data flow en het doel van het programma (program function).

Uit haar experiment [22] bleek dat in eerste instantie de text structure wordt ontwikkeld. Na een modificatie taak werd de plan knowledge opgebouwd. Interessant is hierbij dat de scores voor de text structure, met name van de control flow structure, slechter werden dan voor de modificatie taak.

De resultaten kunnen worden gerelateerd aan hoe teksten worden begrepen, hierbij wordt eerst de text base opgebouwd, waarna het situation model wordt gevormd. Tijdens het begrijpen van een programma wordt in de text base al een idee gevormd van de control flow. Pas na langere aanraking met het programma bouwen programmeurs het situation model op, waardoor ze vragen over data flow en het doel kunnen beantwoorden. Pennington denkt dat het opbouwen van het situation model een gevolg is van de langere tijd en de andere motivatie.

Een andere interessante observatie is dat voor de modificatie taak de meeste fouten worden gemaakt op het gebied van conditionele logica[1]. Na de modificatie daalt dit aantal fouten licht en worden er meer fouten gemaakt in de control flow.

Hierbij dienen enkele kanttekeningen worden gezet. Programmeurs konden tijdens het beantwoorden van de vragen het programma niet naslaan. Er zou een ander beeld kunnen ontstaan wanneer dit wel het geval is. Daarnaast testte Pennington een

[1] De resulterende acties van een programma n.a.v. een beslissing in een if statement, "state" in het onderzoek van Pennington.

procedureel programma, terwijl dit onderzoek zich op object georiënteerde programma's richt.

Een andere kanttekening is dat de programma's welke werden bestudeerd klein waren, tussen de 15 en 200 regels code. Dit betekent dat de resultaten niet kunnen worden gegeneraliseerd naar grotere programma's. Pennington stelt bijvoorbeeld dat eerst de (volledige) text base wordt opgebouwd, voordat wordt begonnen aan een situation model. Dit is onrealistisch voor grotere programma's.

2.2.2 Top down

Volgens de theorie van top down comprehension vormt een programmeur een hypothese van de functie van het programma. Op basis van de broncode of de documentatie worden er nieuwe hypotheses opgesteld, waardoor een hiërarchie van hypotheses ontstaat. Deze hypotheses worden depth first verfijnd en gecontroleerd.

Het bevestigen of afwijzen van een hypothese wordt gedaan door middel van *beacons*. Een beacon is een belangrijk punt in de broncode welke een hypothese bevestigt of afwijst. Een voorbeeld van een beacon is de aanroep naar een `swap` functie, dit zou de hypothese "er wordt een lijst gesorteerd" bevestigen. Volgens Wiedenbeck onthouden ervaren programmeurs regels met beacons beter dan regels zonder beacons [35].

Koenemann et al. hebben programmeurs onderhoudstaken laten uitvoeren op een programma van 635 regels [18]. Hieruit bleek dat programmeurs gemiddeld 33% van de code bekeken, dit betekent dat er niet alléén bottom up wordt begrepen. Programmeurs schakelden pas over op bottom-up comprehension zodra een hypothese werd verworpen of het relevante stuk code was gevonden.

Mayrhauser en Vans hebben programmeurs geobserveerd tijdens onderhoud aan grote programma's, waarbij ze hebben waargenomen dat programmeurs zowel top down als bottom up werken [34, 33]. De grootte van het programma is een bepalende factor, bij een groter programma werken programmeurs meer met top down strategieën. Mayrhauser en Vans bevestigen het gebruik van beacons door programmeurs, ze observeren dat beacons punten zijn waar een programmeur van strategie kan wisselen (bijvoorbeeld van top down naar bottom up).

2.2.3 Objectgeoriënteerde programma's begrijpen

De experimenten welke hiervoor zijn beschreven zijn uitgevoerd op procedurele programma's. In het objectgeoriënteerde (OO) paradigma representeert iedere klasse een concept. Burkhardt et al. nemen aan dat deze concepten onderdeel zijn van het situation model, aangezien deze concepten worden gemodelleerd naar de werkelijkheid [5]. Uit hun onderzoek blijkt dat experts na het bestuderen van een OO systeem een situation model hebben opgebouwd. Dit verschilt met de procedurele stijl, waarin een programmeur begint met de opbouw van het program model, en pas na een modificatie het situation model ontwikkeld [22].

Het effect van OO systemen op junior programmeurs is nog niet geheel duidelijk. De resultaten van Burkhardt et al. geven aan dat deze, net zoals binnen procedureel programmeren, pas na een modificatie taak een situation model opbouwen. Ramalingam en Wiedenbeck rapporteren dat junior programmeurs bij werken aan een OO systeem een beter situation model opbouwen dan bij werken aan een procedureel systeem [26].

2.3 Invloed van het ontwerp

Het ontwerp van een systeem is van invloed op hoe makkelijk dit systeem is om te begrijpen. Een verandering in het ontwerp kan de tijd die het kost om taken uit te voeren beïnvloeden [16].

Een van de moeilijkheden voor een programmeur is het begrijpen van if-then-else statements [17, pagina 21]. Met de refactoring "*Replace conditional with Polymorphism*" kunnen deze statements soms worden omgezet om gebruik te maken van polymorfisme [15]. Door deze refactoring toe te passen wordt de *inheritance depth* verhoogd en ontstaat er delokalisatie.

Inheritance depth Inheritance depth is de totale diepte van een klasse hiërarchie. Een class is op niveau *n* als deze *n - 1* supertypen heeft. Het niveau van de diepste class wordt de inheritance depth genoemd.

Daly et al. hebben onderzocht wat het effect van een diepere inheritance tree is, hieruit bleek dat inheritance zorgde voor snellere, of even snelle, oplossingen tijdens het uitvoeren van onderhoudsta-

ken [9]. Uit een replicatie van dit experiment bleek echter dat de oplossingen langzamer waren bij inheritance, maar wel compacter [6].

Prechelt et al. zetten een experiment op, waarbij het programma groter was, respondenten documentatie tot hun beschikking hadden en ze meer taken dienden uit te voeren [24]. Ondanks dat de verschillen niet in alle gevallen statistisch significant waren, was de trend hierbij dat een diepere inheritance depth tot langzamere en minder correcte resultaten leidde. Vanwege de tegenstrijdige resultaten ten opzichte van de andere onderzoeken, stellen ze dat de onderliggende aanname van deze experimenten, inheritance depth is een bepalende factor, fout is. In plaats daarvan speculeren ze op andere relevante factoren, zoals de gegeven taak (zie paragraaf 2.4), het aantal methoden dat dient te worden begrepen en de hoeveelheid delokalisatie.

Delokalisatie Bij gebruik van polymorfisme staat de logica verspreid over verschillende klassen, allen subtypen van hetzelfde supertype. Deze verspreiding van de logica wordt delokalisatie genoemd. Delokalisatie kan tot gevolg hebben dat een programmeur alle subtypes moet bekijken voordat hij begrijpt wat de aanroep van een methode doet. Mogelijk helpt de notie van *behavioral subtyping*[2] hierbij, maar aangezien dit niet aanwezig is in de huidige objectgeoriënteerde talen is dit buiten beschouwing gelaten.

Arisholm en Sjøberg hebben de *delegated control style*, waarin de verantwoordelijkheden verdeeld zijn over verschillende klassen, vergeleken met de *centralized control style*, waarin een klasse de control flow controleert en andere klassen zich gedragen als data objecten [1]. De delegated control style (DC) werd beschreven als de beter onderhoudbare oplossing, omdat er in de centralized control style (CC) een klasse bestond met een hoge complexiteit. De respondenten waren verdeeld in undergraduate, graduate, junior, intermediate en senior.

Uit het onderzoek bleek dat er bij de CC vaker correcte oplossingen waren geproduceerd, hoewel in de groep van senior programmers de correctheid bijna hetzelfde was. De undergraduates, graduates en junioren waren bij het maken van de CC oplossing ook sneller, terwijl de intermediates en seniors sneller waren met de DC oplossing. De conclusie welke werd getrokken is dat een ontwerp rekening dient te houden met de cognitieve vaardigheden van de programmeurs welke dit ontwerp zullen onderhouden. Hierbij is er een tegenstelling tussen junioren en experts, waar junioren het beste werken met een simpel ontwerp en experts beter werken met een elegant (complex) ontwerp.

Het doel van dit onderzoek was om te zien of er een wisselwerking tussen junioren en experts was wanneer de stijl van controle anders is, niet om te controleren wat de invloed van delokalisatie is. De twee ontwerpen zaten aan de twee uiterste einden van het spectrum van de verdeling van controle, waar in het DC ontwerp de verantwoordelijkheid verspreid was over twaalf klassen en het CC ontwerp één klasse had met veel verantwoordelijkheid en zes data klassen. Toch geeft het onderzoek een goede indicatie dat delokalisatie een factor is bij het onderhouden van een programma.

2.4 Invloed van doel

Het doel van de uit te voeren taak lijkt effect te hebben op de opbouw van de text base (program model) en het situation model. Bij het lezen van teksten wordt er onderscheid gemaakt tussen *read to recall* en *read to do*. Hierbij lijkt read to recall de opbouw van de text base te stimuleren, terwijl read to do de opbouw van het situation model stimuleert. Zo blijkt uit een onderzoek van Mills et al. dat studenten met read to recall instructies de tekst beter konden herinneren, terwijl studenten met read to do instructies beter in staat waren om de taak uit te voeren [20].

In het domein van programmeren lijkt deze invloed ook te bestaan, zo werd in het onderzoek van Pennington het situation model pas opgebouwd na een modificatie taak [22]. Tegelijk met de opbouw van het situation model daalde het program model. Het bestudeerde programma was hierbij korter dan de programma's welke in de praktijk dienen te worden begrepen (220 regels code).

In een onderzoek van Burkhardt et al. [5] kregen programmeurs (junior of expert) een taak toegewezen: documentatie maken (read to recall) of een modificatie uitvoeren (read to do). De programmeurs kregen daarna tijd om het programma te bestuderen, waarna ze de taak uitvoerden. Het ni-

[2] Een semantische relatie waarin gegarandeerd wordt dat subtypes aan dezelfde pre- en postcondities voldoen als het supertype.

veau van het program model en het situation model is gemeten na het bestuderen en na het uitvoeren van de taak.

Het documenteren bleek niet hetzelfde effect als een read to recall taak te hebben, de documentatie groep ontwikkelde tijdens het documenteren namelijk zowel het program model als het situation model.

In de modificatie groep ontwikkelden de experts na het bestuderen zowel een program model als een situation model. De junioren ontwikkelden een sterker program model, maar een slechter situation model. Het uitvoeren van de modificatie verkleinde het verschil tussen junioren en experts, de junioren ontwikkelden hier het situation model terwijl de experts hun situation model niet verder ontwikkelden. Dit bevestigt dat modificatie een read to do actie is, waardoor het situation model wordt ontwikkeld. Tegelijkertijd kan dit betekenen dat er een punt is waar het situation model voldoende is ontwikkeld, en niet verder stijgt.

Een read to do taak zorgt voor een betere opbouw van het situation model. Aangezien veel van de kennis in het begrijpen van objectgeoriënteerde programma's zich op dat niveau bevind, kan het het begrip van zo'n systeem helpen om een read to do taak uit te voeren.

3 Experiment

In het experiment zijn twintig respondenten in twee groepen verdeeld. Een groep heeft typische onderhoudstaken uitgevoerd in een systeem waarin veel polymorfisme wordt gebruikt, een polymorf systeem. De andere groep heeft deze taken uitgevoerd in een conditioneel systeem, welke is gemaakt op basis van het polymorfe systeem. De taken zijn uitgevoerd in een, voor dit experiment gemaakte, online code browser.

In dit hoofdstuk wordt de opzet van dit experiment in meer detail besproken. Er wordt begonnen met de hypothesen, waarna de respondenten en het verloop van het experiment worden besproken. Vervolgens worden de twee versies van het systeem en de uitgevoerde taken daarin beschreven. Daarna volgen de resultaten van het testen van het onderzoek. Tot slot zijn de afhankelijke- en onafhankelijke variabelen besproken.

3.1 Hypothesen

Hieronder volgen per deelvraag de opgestelde hypothesen.

Wat is de invloed van delokalisatie, welke ontstaat door polymorfisme, op de understandability van een objectgeoriënteerd systeem? Zoals vergelijkbare studies [1, 7] is understandability opgedeeld in twee factoren, tijd en correctheid. Hierdoor zijn voor deze deelvraag twee nulhypothesen opgesteld:

- $H1_0$: De delokalisatie heeft geen invloed op de *tijd* besteed aan typische onderhoudstaken.
- $H2_0$: De delokalisatie heeft geen invloed op de *correctheid* van de oplossingen gegeven tijdens het werken aan die taken.

De alternatieve hypothesen zijn hierbij:

- $H1$: De delokalisatie heeft een negatieve invloed op de *tijd* besteed aan typische onderhoudstaken.
  Door de if-then-else statements te vervangen door polymorfisme wordt cyclomatische complexiteit vervangen door delokalisatie. Hierdoor dienen programmeurs meer bestanden op te vragen, waardoor de taken meer tijd kosten om uit te voeren.
- $H2$: De delokalisatie heeft een positieve invloed op de *correctheid* van de gegeven oplossingen.
  In de conditionele situatie staat alle informatie op een plaats, dit heeft een hoge complexiteit waardoor er fouten worden gemaakt. Door de lagere complexiteit in de polymorfe situatie komen er hier meer correcte oplossingen.

Wat is de invloed van de expertise van een programmeur tijdens het begrijpen van systemen waarin veel polymorfisme wordt gebruikt?

- $H3$: Experts kunnen beter omgaan met de delokalisatie, waardoor ze sneller zijn in het uitvoeren van typische onderhoudstaken in een systeem waarin veel polymorfisme wordt gebruikt.
  Bij het afzetten van een delegated control style

tegenover een centralized control style bleken experts beter om te gaan met met de delegated control style, waar de verantwoordelijkheid verspreid is over meerdere klassen [1]. Aangezien de delegated control style overeenkomsten heeft met de polymorfe situatie zullen experts waarschijnlijk beter omgaan met de polymorfe situatie.

3.2 Uitvoering

In totaal hebben twintig respondenten het experiment voltooid, deze waren gelijk verdeeld over de conditionele- en de polymorfe implementatie. Hierbij kregen de respondenten een korte introductie van de online code browser en vulden ze een korte vragenlijst in, waarna de taken werden uitgevoerd.

Door middel van de vragenlijst is het niveau van de respondent beoordeeld. De gemiddelde werkervaring was in de conditionele implementatie vijf jaar, tegenover vier in de polymorfe implementatie. Met behulp van een 5-punts likertschaal is de java ervaring gevraagd, het gemiddelde was in beide implementaties 3,3.

Tijdens de eerste taak is er een klasse diagram (Zie figuur 1) en een readme gegeven is als documentatie. Door deze vroeg in het proces beschikbaar te stellen is het begrip van het systeem ondersteund [8] en is het experiment realistisch [11].

Vrijwillig Een belangrijke overweging was het feit dat alle respondenten vrijwillig deelnamen aan het onderzoek. Om de barrière om mee te doen zo laag mogelijk te houden, is er voor gekozen om het mogelijk te maken om via internet deel te nemen aan het onderzoek. Dit betekent dat een browser genoeg is om de broncode en documentatie te bekijken en de taken uit te voeren. Een respondent hoeft dus geen reis te maken of tools te installeren, maar kan meteen beginnen. Ook is er geen aparte taak nodig om respondenten te introduceren met de procedure van het downloaden of uploaden van de broncode, wat de duur van het experiment beperkt.

Het verlagen van deze barrière heeft echter een lagere controle tot gevolg. De online respondenten zouden immers kunnen worden afgeleid, of pauzeren tijdens het experiment. Dit maakt het meten van de tijd lastig, in paragraaf 4.1 staat hoe hier rekening mee is gehouden bij het beoordelen van de resultaten.

Naast de online binnengekomen resultaten zijn er twee sessies georganiseerd waarbij het onderzoek 'persoonlijk' werd afgenomen. Hier zijn de respondenten zo min mogelijk beïnvloed, zodat de resultaten vergelijkbaar zijn met de online sessies. Vragen over de taken, de tool of de broncode werden hierbij niet beantwoord. In geen van de sessies waren de respondenten op de hoogte van het doel van het onderzoek.

3.3 Systeem

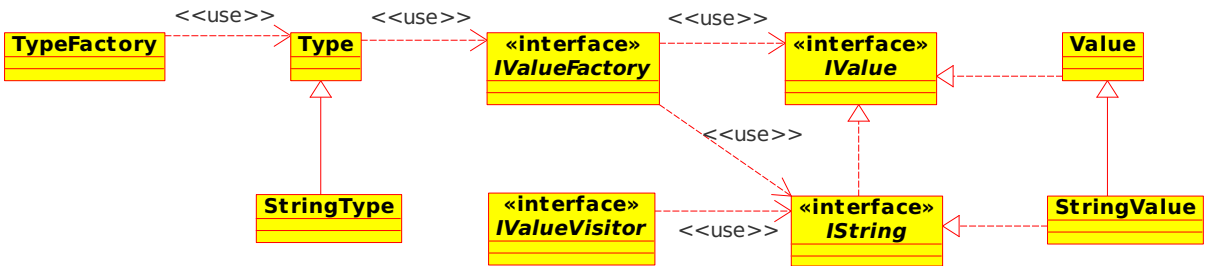
Het te begrijpen systeem is een library van het Eclipse IDE Meta-tooling Platform (IMP), namelijk Pdb.values. Deze library wordt gebruikt om feiten over broncode te representeren. Hiervoor is gekozen omdat:

- Het geschreven is in Java, waar potentiële respondenten bekend mee kunnen zijn.
- Het systeem zich houdt aan de richtlijnen voor objectgeoriënteerd programmeren, er is dus veel delokalisatie aanwezig. Hierdoor kan het, zonder aangepast te worden, gebruikt worden als de polymorfe implementatie.
- Kiezen voor een open source systeem helpt bij het reproduceren van het onderzoek.
- Het 175 klassen en 28119 regels code bevat. Een groter systeem zou te moeilijk zijn om binnen relatief korte tijd te begrijpen, terwijl het tegelijkertijd groot genoeg is om een uitdaging te vormen.
- Er feedback kon worden gevraagd aan een auteur van het systeem.

Als indicatie van de complexiteit van de library staat in figuur 1 een klasse diagram.

3.4 Maken van conditionele versie

Hieronder wordt beschreven hoe, op basis van de polymorfe implementatie, de conditionele implementatie is gemaakt. Hierbij is erop gelet dat de aanpassing het gedrag van het systeem niet aanpast. Daarnaast hebben de twee ontwerpen dezelfde code stijl, naamgeving en hoeveelheid commentaar.



Figuur 1: Klasse diagram van pdb.values

Een risico bij het aanpassen is dat er een onrealistische onderhoudssituatie ontstaat [11]. Zo werden in andere experimenten [24, 16] de attributen en methoden van het supertype naar de subtypes gekopieëerd (refactorings *push down method* en *push down field*), de inheritance link weggehaald en het supertype verwijderd. Op deze manier was het systeem van [24] bij een inheritance depth van vijf 1200 regels lang, terwijl de versie zonder inheritance 2470 regels telde. Dit komt omdat op deze manier de code van het supertype geduplicceerd is over alle subtypes, wat geen realistische onderhoudssituatie oplevert.

Om dit te voorkomen is, in plaats van *push down*, voor *pull up* gekozen. Het bleek echter moeilijk om de (twintig) subtypen compleet te verplaatsen naar het supertype, vooral het aantal constructors leverde een probleem op. De kans dat hierdoor een bug zou worden gemaakt was te groot. Ook zou het weghalen van de subtypes betekenen dat er twee versies van de documentatie moesten worden gemaakt, terwijl het verschil tussen de twee groepen zou klein mogelijk dient te zijn.

Hierdoor, en op basis van de input van de auteur van Pdb.values, is besloten om niet alle methoden te combineren, maar alleen twee centrale methoden, namelijk `isSubtypeOf()` en `accept()`. De hiërarchie is dus in stand gehouden, maar de subtypen overschrijven minder methoden. Aangezien een deel van de code in de subtypes overlapte, zijn de twee resulterende methode korter dan de som van de methoden in de subtypes. Voor de `isSubtypeOf()` methode resulteerde dit in een methode van 65 regels code. Het systeem is in de conditionele situatie gereduceerd van 28119 naar 28014 regels code.

#	Omschrijving
A1	Investigating the functionality of (a part of) the system
A2	Adding to or changing the systems functionality
A3	Investigating the internal structure of an artefact
A4	Investigating dependencies between artefacts
A5	Investigating runtime interactions between artefacts
A6	Investigating how much an artefact is used
A7	Investigating patterns in the system's execution
A8	Investigating the quality of the system's design
A9	Understanding the domain of the system

Tabel 1: Principal comprehension activities van Pacione et al.

3.5 Taken

Er zijn twee typen taken, de controle taken en de experimentele taken. De controle taken (T1 en T2) dienen om te bepalen of de groepen in evenwicht zijn, hierbij is erop gelet dat de respondent geen code tegenkomt welke verschilt tussen de twee versies van het systeem. Daarna volgen er taken waarbij de respondent de verschillen wel tegenkomt, de experimentele taken (T3 tot en met T7).

Bij het opstellen van deze taken is rekening gehouden met verschillende criteria. Zo is het belangrijk dat de taken representatief zijn voor onderhoudswerkzaamheden en dat de taken representatief zijn voor het systeem wat wordt onderhouden. Tot slot is ervoor gekozen om open vragen te

#	Omschrijving	Activiteiten 1	2	3	4	5	6	7	8	9
T1	We start at the `runWithString` method of the `SimpleRun` class. You can find it in org.eclipse.imp.pdb.facts.io. While you're busy debugging you notice that `SimpleRun` is calling `StandardTextReader` at line 37. Imagine that there is a breakpoint on line 126 of `StandardTextReader`. What would the class of 'result' be at this point, and what would the value of the relevant variables of that class be? *Discard lines 127-139 of StandardTextReader for now.*	x		x		x		x		x
T2	Now continue with the `runWithList` method of the `SimpleRun` class. You still have the breakpoint on line 126 of `StandardTextReader`. How often would it hit the breakpoint? What would the class of 'result' be each time, and what would the value of the relevant variables of that class be? *You can discard lines 127-139 of StandardTextReader.*	x		x		x		x		x
Totaal		x		x		x		x		x

Tabel 2: De controle taken

stellen. Hieronder wordt ingegaan op deze punten.

Onderhoudstaken Om de ontwikkeling van het situation model te stimuleren en dicht bij de praktijk te blijven is op basis van paragraaf 2.4 besloten om minstens één modificatie taak uit te laten voeren.

Daarnaast hebben Pacione et al. op basis van literatuuronderzoek een lijst gemaakt van *principal comprehension activities*. Ze stellen dat een set van onderhoudstaken samen al deze activiteiten dient te bevatten [21]. De activiteiten staan opgesomd in tabel 1.

In tabellen 2 en 3 staat een overzicht van de verschillende taken, samen met welke taken welke activiteiten afvangen. Zo vangt bijvoorbeeld taak T1 activiteit A1, *"investigating functionality of (a part of) the system"*, af. Er is gecontroleerd dat de set van experimentele taken alle activiteiten omvat. De twee controle taken omvatten niet alle activiteiten, hiervoor is gekozen om zo weinig mogelijk tijd van de respondenten in beslag te nemen.

Onderhoud in PDB Het is belangrijk dat de taken representatief zijn voor het systeem waaraan onderhoud wordt gepleegd. Om hiervoor te zorgen zijn de taken opgesteld op basis van commit berichten in het versiebeheersysteem. Vervolgens is er feedback gevraagd aan een ontwikkelaar van het systeem, waarna de taken zijn bijgesteld.

Open vragen Er is gekozen om de taken als open vragen te stellen, hierdoor zijn de taken representatiever voor echte onderhoudstaken. Zo kunnen respondenten niet terugvallen op gokken, of een voor een de foute antwoorden afstrepen [7].

3.6 Online code browser

Er is voor gekozen om de tool in de browser aan te bieden. Hierdoor is de controle hierop sterk, in alle sessies wordt immers dezelfde tool gebruikt.

Een ander voordeel is dat de resultaten onderling beter vergeleken kunnen worden, er is immers dezelfde tooling gebruikt. Ook kan er op de webserver worden gelogged welke bestanden wanneer zijn opgevraagd. Daarnaast worden de vragen en

#	Omschrijving	Activiteiten								
		1	2	3	4	5	6	7	8	9
T3	Now focus on lines 127-130 of `StandardTextReader`. When executing the `runWithList` method of `SimpleRun`, are these lines executed? For each time they are executed: Which two types are compared against each other and what is the result of `isSubTypeOf()`?	x		x		x		x		x
T4	What is the output when you feed the resulting value from the `runWithList` method of `SimpleRun` to `StandardTextWriter`?	x		x		x		x		x
T5	`Comparable` is a method of `Type`. Is a `VoidType` comparable to a `MapType`? In other words: What is the result of `voidType.comparable(mapType)` Please include the class(es) and the line number(s) that gave you the answer.	x		x		x		x	x	x
T6	First an `AliasType` is created, containing a `VoidType`. What is the response if this instance of `AliasType` is the first parameter to the `isSubtypeOf` function of a `StringType`? Please include the class(es) and the line number(s) that gave you the answer.	x	x	x		x				x
T7	How would you add a new `Type`, `OrderedSet`, including it's new `IValue`? Which interface(s) would you add, which classes would you add, what would these classes extend? Which methods would you add and which methods would you edit?		x	x	x		x			x
Totaal		x	x	x	x	x	x	x	x	x

Tabel 3: De experimentele taken

de broncode overzichtelijk in hetzelfde scherm getoond. Tenslotte komen de resultaten hierbij in één database terecht, wat het beoordelen van de resultaten makkelijker maakt.

Een nadeel van deze keuze is dat deze code browsers door middel van statische analyse worden gemaakt, wat het bijvoorbeeld onmogelijk maakt om een debugger aan te bieden. Daarnaast kan het gedrag van individuele webbrowsers verschillen, waardoor de ervaring van respondenten kan verschillen.

Het grootste probleem met deze keuze is echter dat, ondanks dat er bewegingen zijn in de richting van online IDE's[3], er op het moment van schrijven geen online code browser bestond welke voldeed aan de eisen. Hierdoor is besloten om er zelf een te maken.

Aangezien de tool nieuw is, was de tool bij alle respondenten onbekend. Hierdoor zijn er wel leereffecten (Zie ook paragraaf 3.7), maar deze zijn gelijk. De respondenten konden de broncode niet downloaden voor gebruik in een andere tool.

3.6.1 Eisen aan een code browser

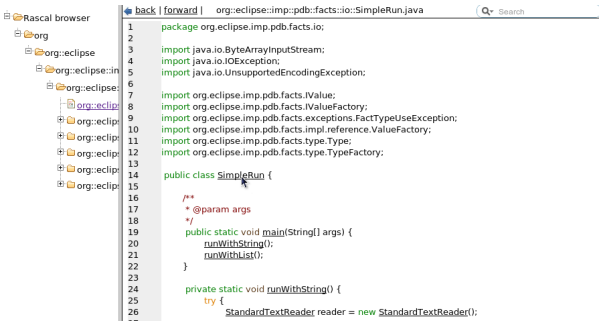
Een belangrijke eis aan een code browser is om niet onder te doen voor de tooling welke een ontwikkelaar normaal tot zijn beschikking heeft. Hierbij is uitgegaan van de functionaliteit welke Eclipse biedt in bij het bekijken en begrijpen van broncode.

Daarnaast is er uitgegaan van het framework van Storey voor het beoordelen van *software exploration tools* [31]. Storey geeft bij het opstellen van het framework aan dat huidige IDE's niet alle elementen van het framework bevatten. Het doel van het framework is dan ook om de toekomst van IDE's te bepalen, terwijl het doel in dit onderzoek is om niet onder te doen voor bestaande IDE's. Hierdoor is gekozen om te kijken in hoeverre Eclipse voldoet

[3] http://eclipse.dzone.com/news/who-needs-online-ide

E1	Indicate syntactic and semantic relations between software objects	x
E2	Reduce the effect of delocalized plans	x
E3	Provide abstraction mechanisms	
E4	Support goal-directed, hypothesis-driven comprehension	
E5	Provide overviews of the system architecture at various levels of abstraction	
E6	Support the construction of multiple mental models	
E7	Cross-reference mental models	
E8	Provide directional navigation	x
E9	Support arbitrary navigation	x
E10	Indicate the maintainer's current focus	
E11	Show the path that led to the current focus	x
E12	Indicate options for further exploration	x
E13	Reduce additional effort for user-interface adjustment	
E14	Provide effective presentation styles	

Tabel 4: Ondersteuning van Eclipse voor cognitieve elementen welke helpen tijdens software comprehension.



Figuur 2: Screenshot van de code browser

aan het framework, waarna de code browser ook deze elementen dient te bevatten. Zie tabel 4 voor een overzicht van deze elementen.

Links tussen objecten De elementen `E1` en `E8` betreffen de relaties binnen de broncode en de navigatie ertussen. Door een gebruiker in staat te stellen om hiertussen snel te navigeren wordt meteen het effect van delokalisatie verminderd (`E2`). Door ook aan te geven wanneer een methode wordt overschreven of aangeroepen wordt voldaan aan `E12`.

Doorzoekbaarheid Wanneer een ontwikkelaar naar een plek in de code wilt gaan, welke niet direct met de links bereikbaar is, dient deze te kunnen zoeken naar die plek, hierdoor wordt `E9` afgevangen. Naast de zoekfunctie bevat Eclipse hiervoor ook een package browser.

Historie Door een historie op te bouwen, en de gebruiker in staat te stellen om naar vroeger opgevraagde plekken in de code te navigeren, wordt voldaan aan `E11`.

Documentatie In Eclipse zijn er geen links tussen de documentatie en de broncode, de documentatie is los opvraagbaar. Hierdoor mist Eclipse `E6`, `E7` en `E10`. Dit gedrag is overgenomen.

3.6.2 Maken van een code browser

Op basis van de hiervoor beschreven eisen is de code browser gemaakt, zie figuur 2 voor een screenshot.

3.7 Testen van het onderzoek

Om het onderzoek te testen is het afgenomen op drie medestudenten en twee programmeurs welke aan het systeem hebben gewerkt. Op basis van hun feedback is het onderzoek aangepast.

Algemene feedback In eerste instantie werd bij meerdere programmeertalen alleen de optie gegeven of een taal wel of niet bekend was. Dit werd als beperkt ervaren, dus is dit aangepast naar een 5-punts likertschaal.

Daarnaast werd er in het begin een readme aangeboden, maar deze kon een gebruiker niet nogmaals opvragen. Dit is mogelijk gemaakt.

Taken

- Na het lezen van de taken was niet altijd duidelijk wat er werd verwacht, dus deze zijn specifieker geformuleerd.
- Tijdens de eerste test bleek dat de eerste (controle) taak overweldigend was voor iemand die het systeem voor het eerst ziet, deze is daarom opgesplitst in twee kleinere taken.
- Het was onduidelijk hoeveel taken er nog moesten worden uitgevoerd. Op basis van deze feedback is het aantal resterende taken weergegeven.

Code browser De zoekfunctie stond bovenaan de pagina. Wanneer er een lang bestand werd opgevraagd, en er naar beneden werd gescrolled, was deze niet meer zichtbaar. Om dit te verhelpen is een balk bovenin toegevoegd welke altijd zichtbaar is. Hierin is ook de bestandsnaam weergegeven, zodat respondenten deze altijd kunnen zien.

Een ander probleem met de zoekfunctie was dat er resultaten in terugkwamen welke niet relevant waren, dit is aangepast zodat deze worden gefilterd.

Tot slot waren alle packages in de package browser standaard open geklapt, waardoor het meer tijd kostte om een bepaald bestand te vinden. Op basis van deze feedback is besloten om deze slechts uit te klappen tot een bepaalde diepte, waarna een respondent er zelf voor kan kiezen om packages in- of uit te klappen.

3.8 Analyse van variabelen

De eerste onafhankelijke variabele van dit experiment is het krijgen van de conditionele of polymorfe versie.

De tweede onafhankelijke variabele is het zijn van expert of junior. Dit onderscheid is gemaakt op basis van de werk- en java ervaring.

De eerste afhankelijke variabele is de bestede tijd aan een taak. Hiervoor wordt de tijd bijgehouden bij het beantwoorden van een vraag. Aangezien teruggaan naar eerdere taken niet toegestaan is, kan de tijd per taak makkelijk worden gereconstrueerd.

De tweede afhankelijke variabele is de correctheid van de oplossingen. Deze wordt gemeten door een oplossingsmodel toe te passen op de oplossingen. In het oplossingsmodel is aangegeven welke elementen een oplossing dient te bevatten en hoeveel punten deze waard zijn.

Om de hypothesen te valideren zal gebruik gemaakt gaan worden van een one-tailed Student's t-test. Deze veronderstelt een normaalverdeling voor de variabelen. Voor de behaalde correctheid was dit het geval, maar voor de bestede tijd niet. Om voor de bestede tijd toch een normaaldistributie te bereiken is, zoals in bijvoorbeeld [1], een logaritmische transformatie toegepast.

Voor zowel de tijd als de correctheid wordt een betrouwbaarheidsinterval van 95% aangehouden, het resultaat is dus statistisch significant bij p-waarden onder de 0,05.

Voor het berekenen van de statistieken is R gebruikt[4].

4 Resultaten

Hieronder volgt eerst een beschrijving van de procedure die is gevolgd om tot vergelijkbare resultaten te komen. Daarna volgt per hypothese een paragraaf met de betreffende resultaten. Tot slot volgt een analyse van de fouten welke respondenten hebben gemaakt.

De resultaten beperken zich tot de experimentele taken (taak T3 tot en met T7, exclusief de controle taken) tenzij anders vermeld.

[4] http://www.r-project.org/

4.1 Filteren van de resultaten

Het online afnemen van het experiment heeft tot gevolg dat de focus van de deelnemers niet altijd op het experiment ligt. Zo kunnen ze worden afgeleid door andere programma's (bijvoorbeeld chat of e-mail) of mensen. Bij het bestuderen van de resultaten bleek dat twee respondenten met onderbrekingen aan het experiment hebben meegedaan, dit maakt het meten van de tijd onmogelijk. Aangezien de tijd en de correctheid van een oplossing niet los van elkaar staan, is besloten om deze respondenten niet mee te nemen.

De respondenten zijn gevonden door te zoeken naar reactie tijden van boven een uur, of met afwijking van het gemiddelde groter dan drie maal de standaard deviatie van die persoon. De gevonden respondenten zijn met de hand nagelopen en bleken geen valse positieven te bevatten, terwijl wel alle gevallen die met de hand gevonden waren hierin zaten. Dit is dezelfde procedure als Pennington heeft gebruikt [22], aangepast voor de langere duur van dit experiment.

Dit sluit niet uit dat bij andere respondenten sprake is geweest van onderbreking, maar er wordt aangenomen dat, mede omdat er geen grote onderbrekingen meer aanwezig zijn, deze factor even groot is in beide groepen.

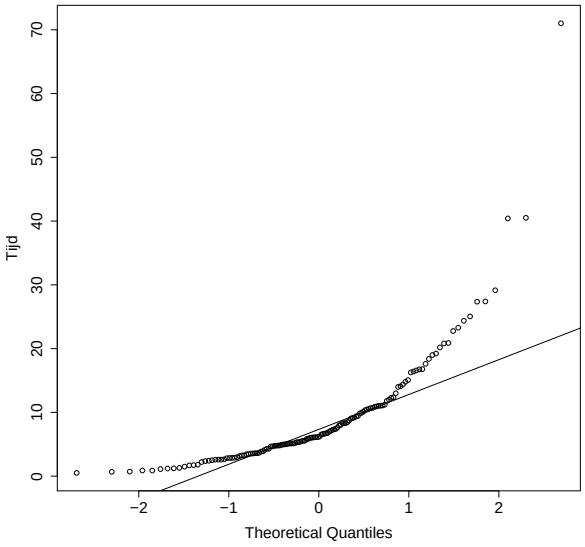
Een ander mogelijk gevolg van het online afnemen is dat de motivatie van enkele respondenten onvoldoende was. Drie respondenten behaalden een correctheid van nul, deze zijn niet meegenomen in de resultaten.

Tot slot bleek het experiment voor twee respondenten te moeilijk, deze slaagden er niet in om de eerste taak te voltooien.

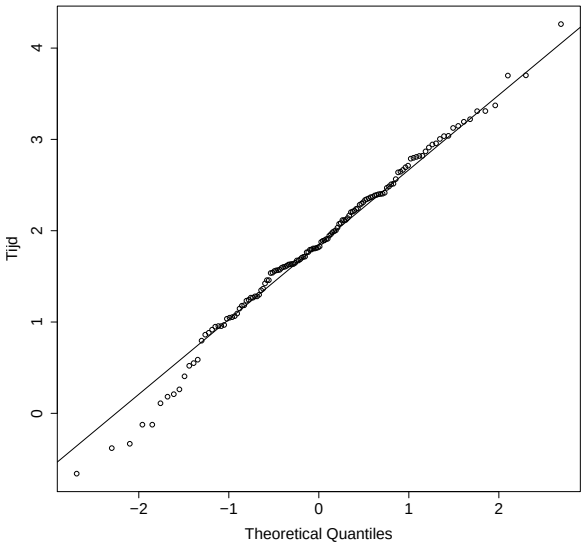
4.2 Tijd

Allereerst wordt de nulhypothese $H1_0$ getest: De delokalisatie heeft geen invloed op de *tijd* besteed aan typische onderhoudstaken.

Figuur 3 laat het effect van de logaritmische transformatie zien. Figuur 4 laat de totaal bestede tijd zien voor de experimentele taken. Uit tabel 5 blijkt dat de polymorfe groep gemiddeld 1,6% minder tijd nodig had. Aangezien dit maar een klein verschil is kan deze nulhypothese niet worden afgewezen.



(a) Voor transformatie



(b) Na transformatie

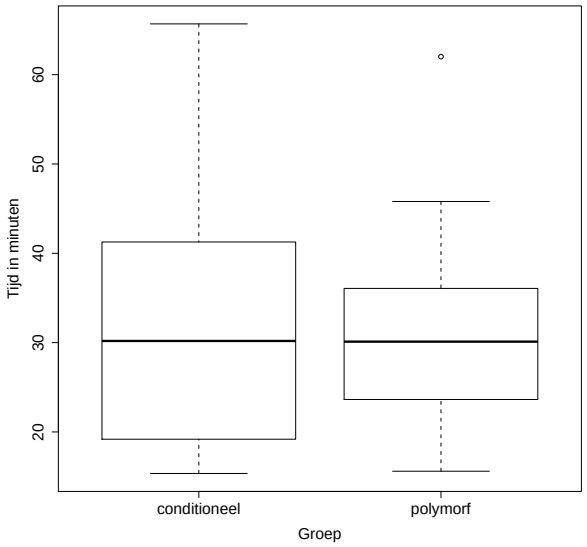
Figuur 3: Een Q-Q plot van de normaalverdeling van de bestede tijd voor en na de logaritmische transformatie.

	Tijd		**Correctheid**	
	Conditioneel	Polymorf	Conditioneel	Polymorf
gemiddelde	32,67	32,16	11,3	13,1
verandering (%)		-1,6%		+15,9%
min	15,35	15,6	3	6
max	65,68	62,02	19	18
mediaan	30,18	30,11	12,5	13
st.d.	15,99	13,74	6,08	3,67
one-tailed Student's t-test				
df	17,61			14,84
t	0,05			0,81
p-waarde	0,522			0,217

Tabel 5: Tijd (in minuten) en correctheid voor conditionele en polymorfe versie.

	Taken				
	T3	T4	T5	T6	T7
Gemiddelde tijd (minuten)					
Conditioneel	8,35	4	4	6,19	10
Polymorf	8,32	4,9	3,7	7,3	7,87
Verandering (%)	-0,04%	+22,5%	-7,5%	+17,93%	-21,3%
Gemiddelde correctheid					
Conditioneel	2,3	1,9	3,2	2,1	1,8
Polymorf	2,1	2,3	3,7	3,4	1,6
Verandering (%)	-8,7%	+21,05%	+15,63%	+61,9%	-11,11%
Gemiddeld aantal fouten					
Conditioneel	0,9	0,6	0,5	1,1	0,4
Polymorf	0,8	0,8	0,2	0,5	0,4
Verandering (%)	-11,11%	+33%	-40%	-54,54%	0%

Tabel 6: Gemiddelde bestede tijd, behaalde correctheid en gemaakte aantal fouten per taak.



Figuur 4: Boxplot van de bestede tijd in minuten

De tijdsverschillen voor individuele taken zijn weergegeven in tabel 6.

4.3 Correctheid

De volgende nulhypothese, $H2_0$, zegt dat de delokalisatie geen invloed heeft op de *correctheid* van de oplossingen gegeven tijdens het werken aan typische onderhoudstaken.

In figuur 5 staan de scores voor de correctheid per taak. In tabel 5 zijn de scores van T3 tot en met T7 opgeteld, hieruit blijkt dat de polymorfe groep gemiddeld 15,9% correcter was. Uit de maximaal haalbare score van 20 haalde deze groep gemiddeld 13,1 punten, tegenover 11,3 punten voor de conditionele groep.

Ondanks dit verschil is de p-waarde met 0,217 niet statistisch significant ($> 0{,}05$), waardoor de nulhypothese niet kan worden afgewezen.

In de discussie zal dieper worden ingegaan op het verschil in correctheid van individuele taken, deze zijn te vinden in tabel 6.

4.4 Expertise

De derde hypothese is dat experts beter om kunnen gaan met de delokalisatie, waardoor ze sneller zijn in het uitvoeren van typische onderhoudstaken in een systeem waarin veel polymorfisme wordt gebruikt.

Hierbij is op basis van werk- en java ervaring de meest ervaren helft van de respondenten verzameld, dit zijn er 10. Door dit lage aantal zijn de groepen tijdens het uitvoeren van de controle taken niet geheel in balans. Volgens tabel 7 waren de experts tijdens de controle taken in de polymorfe groep 5,3% sneller. Tijdens de experimentele taken waren de experts in de polymorfe groep slechts 3,9% sneller, de trend wijst dus in de omgekeerde richting. De hypothese kan op basis van deze resultaten niet worden geaccepteerd.

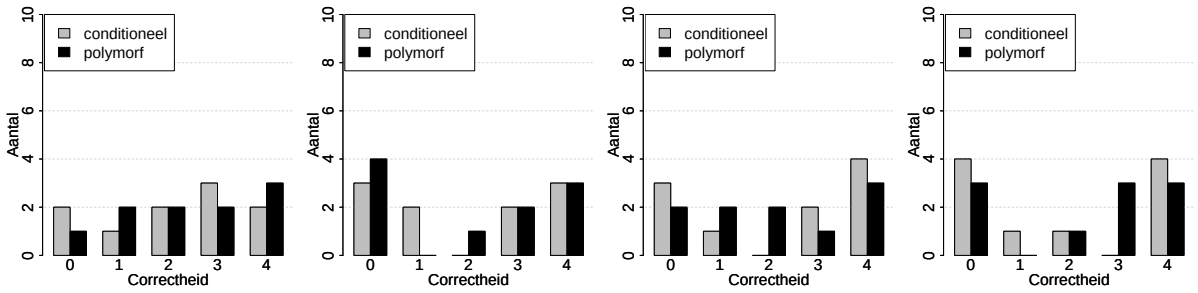
4.5 Oorzaken van fouten

Aangezien er geen statistische significantie is bereikt, is besloten om een diepere analyse te doen. Iedere keer dat een respondent een taak heeft uitgevoerd, is op basis van het antwoord gekeken wat de oorzaken waren van een eventuele fout[5]. Bij gebrek aan een bestaande typering uit de literatuur is er een opgesteld. Aangezien deze typering niet vooraf is opgesteld, is op basis hiervan geen statistiek bedreven. Door deze typering kan meer informatie worden gehaald uit de resultaten, welke een leidraad kan vormen in de discussie.

Hieronder worden de acht typen oorzaken beschreven, samen met hoe vaak deze oorzaak is toebedeeld.

1. Incompleet (24). De redenatie klopt, maar er missen details waardoor het antwoord niet de volledige punten krijgt qua correctheid.
2. Booleaanse fout (8). De uitkomst van een booleaanse expressie wordt omgedraaid.
3. Verkeerde methode (7). Er wordt uitgegaan dat een andere methode wordt aangeroepen dan eigenlijk wordt aangeroepen. Een voorbeeld is aannemen dat een methode in een supertype wordt aangeroepen, terwijl een subtype de methode had overschreven.
4. Recursie (11). Een methode roept zichzelf aan, maar dit wordt niet opgemerkt. Een andere mogelijkheid is dat er een vergissing wordt gemaakt in de volgorde van de recursieve aanroepen.

[5]Het is mogelijk dat een respondent ondanks een redenatie fout toch een correct antwoord heeft ingevuld.



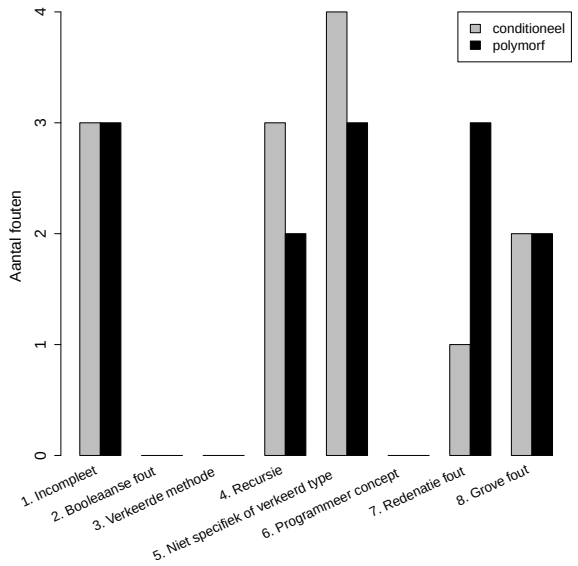
(a) Taak 1 (b) Taak 2 (c) Taak 3 (d) Taak 4

(e) Taak 5 (f) Taak 6 (g) Taak 7

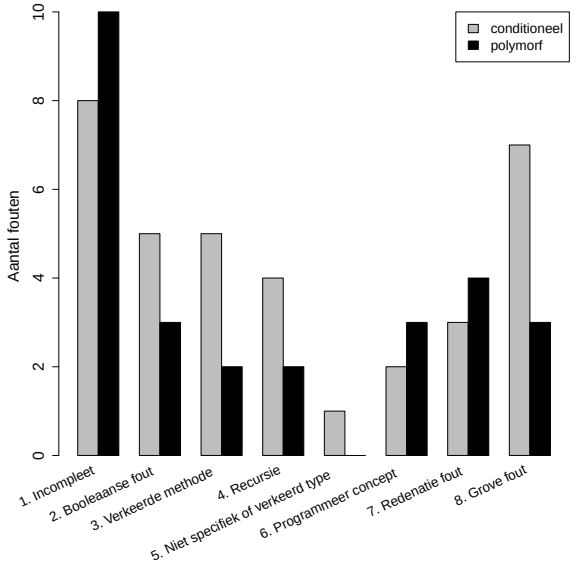
Figuur 5: Aantal respondenten met een bepaalde correctheid per taak.

	Controle taken		**Experimentele taken**	
	Conditioneel	Polymorf	Conditioneel	Polymorf
gemiddelde	28,97	27,43	31,99	30,74
verandering (%)		-5,3%		-3,9%
min	16,58	10	19,18	18,45
max	37,22	61,3	47,62	45,8
mediaan	34,37	17,98	34,5	30,75
st.d.	10,08	21,73	12,16	10,57
one-tailed Student's t-test				
df				7,86
t				0,11
p-waarde				0,46

Tabel 7: Bestede tijd (in minuten) door experts tijdens controle- en experimentele taken.



Figuur 6: Oorzaken van gemaakte fouten in de conditionele en polymorfe groep tijdens de controle taken



Figuur 7: Oorzaken van fouten in de conditionele en polymorfe situatie na de controle taken

5. Niet specifiek of verkeerd type (8). Het runtime type van een variabele is van een ander type dan de respondent denkt. Wanneer dit van toepassing is wordt verkeerde methode en recursie niet als oorzaak gezien.

6. Programmeer concept (5). De respondent maakt een fout doordat een bepaald programmeer concept niet wordt begrepen.

7. Redenatie fout (11). Een respondent begint met een logische redenatie, maar is ergens op een verkeerd idee gebracht waardoor het antwoord niet klopt. Wanneer dit de oorzaak is sluit dit incompleet antwoord als oorzaak uit.

8. Grove fout (14). Het antwoord van de respondent klopt niet en er kan geen specifieke oorzaak aan worden gekoppeld. Wanneer dit de oorzaak is sluit dit de andere oorzaken uit.

In figuur 6 staan de oorzaken van fouten tijdens de controle taken. In figuur 7 staan de oorzaken van fouten welke in het experiment zijn gemaakt, deze fouten zijn in tabel 6 uitgesplitst per taak.

5 Discussie

De discussie is opgesplitst in verschillen van tijd, correctheid, expertise en taken.

5.1 Invloed op tijd

Er zit weinig verschil in de bestede tijd tussen de conditionele en polymorfe versie, waardoor de nulhypothese $H1_0$ niet kan worden afgewezen.

De alternatieve hypothese nam aan dat de delokalisatie een negatieve invloed had op de *tijd* besteed aan typische onderhoudstaken, omdat het wisselen tussen meerdere bestanden voor overhead zorgt. Het klopt dat de polymorfe groep meer bestanden heeft opgevraagd, gemiddeld 56 tegenover 49 voor de conditionele groep[6]. Het bekijken van een bestand kost tijd:

- Het kost tijd om een bestand weer te geven.
- De programmeur dient zich te oriënteren op dit nieuwe bestand. Hierbij stelt hij zichzelf vragen zoals: “Waar ben ik in het bestand?” en “Waar staat de informatie die ik zoek?”

[6]Bij het gebruiken van de back en forward knoppen wordt

In de conditionele versie hoeven programmeurs minder bestanden op te vragen, dit scheelt dus tijd. Toch is deze groep niet sneller.

Dit resultaat spreekt de suggestie van Prechelt et al. tegen, namelijk dat de hoeveelheid delokalisatie en de hoeveelheid methoden een belangrijke factor zijn. Mogelijk komt dit door het verschil in het maken van de conditionele versie. Waar in andere experimenten de code van het supertype werd gedupliceerd naar de subtypes, is hier de code van de subtypes naar het supertype verplaatst. Dit zorgt voor een hogere complexiteit in het supertype. Een ander verschil is dat het te begrijpen systeem groter is.

Over het hele experiment wogen de twee factoren, complexiteit en delokalisatie, tegen elkaar op: de bestede tijd verschilde niet significant. Een reden voor dit evenwicht kan zijn dat er slechts twee methoden zijn aangepast, waardoor de systemen veel op elkaar leken. Toch is in tabel 6 te zien dat er in individuele taken wél verschil zit in de bestede tijd, daarom volgt in paragraaf 5.3 een analyse per taak.

5.2 Invloed op correctheid

De correctheid was in de polymorfe groep 15,9% hoger. Dit is niet statistisch significant, waardoor de nulhypothese $H2_0$ niet kan worden afgewezen. De trend is hierbij wel in de richting van de alternatieve hypothese, namelijk dat er correctere resultaten worden bereikt wanneer taken worden uitgevoerd op een systeem waarin veel polymorfisme wordt gebruikt.

Uit figuur 7 blijkt dat het verschil in correctheid niet werd veroorzaakt door incomplete antwoorden, deze kwamen ongeveer even vaak voor. Aangezien alle fouten betreffende recursie in de derde taak zijn gemaakt worden deze in paragraaf 5.3 behandeld. De overige verschillen in correctheid waren te vinden in de categorieën booleaanse fouten, verkeerde methode en grove fouten.

Booleaanse fouten In de conditionele versie werd er vijf maal een fout gemaakt in conditionele logica, tegenover drie in de polymorfe versie. Dit is een verwacht resultaat, de conditionele versie bevat immers meer conditionele statements waar fouten in kunnen worden gemaakt.

Verkeerde methode In de conditionele versie zijn vijf fouten gemaakt vanwege het kijken naar een verkeerde methode, tegenover twee in de polymorfe versie. Aangezien er in de conditionele versie juist *minder* methoden worden overschreven dan in de polymorfe versie, is dit een verrassend resultaat. Er zijn immers minder mogelijkheden om deze fout te maken.

Een nadere inspectie wijst uit dat in alle gevallen de respondenten dachten dat een methode in het supertype werd aangeroepen, terwijl deze werd overschreven in een subtype. Door een pull up method refactoring uit te voeren op een belangrijke methode van een systeem worden dit soort fouten dus vaker gemaakt.

Grove fouten Tijdens het experiment zijn er tien grove fouten gemaakt, waarvan zeven tijdens werken aan het conditionele systeem en drie tijdens werken aan het polymorfe systeem. Het betreft vooral respondenten die niet in staat waren de taak uit te voeren en daardoor gokten naar het goede antwoord of aangaven deze taak niet uit te kunnen voeren. Vier respondenten (drie conditioneel, één polymorf) gaven geen uitleg bij een fout antwoord, waardoor er geen specifiekere fout aan kon worden verbonden.

5.3 Verschillen in individuele taken

Hieronder wordt dieper ingegaan op individuele taken en de verschillen tussen de groepen.

Taak 3

Taak drie is de eerste taak waarbij de respondenten de verschillen tussen de twee systemen zijn tegenkomen, hierbij zijn dus de initiële effecten geobserveerd. De taak is een vervolg op de vorige taak, waar gevraagd werd om de executie van een scenario door te lopen tot een bepaald punt. De respondent diende er in deze taak achter te komen of er aan het if statement na dit punt werd voldaan.

De taak is in ongeveer evenveel tijd uitgevoerd en gemiddeld gezien waren de antwoorden even correct. Zoals blijkt uit figuur 5c waren de onderlinge

de browser cache gebruikt, dus worden deze acties niet gelogged.

verschillen in correctheid echter groter in de conditionele groep. Er is een tweedeling ontstaan in deze groep, waarbij zes respondenten tot een goed antwoord kwamen en vier niet. Dit verschil kan komen door de lengte van de if-then-else structuur in de `isSubtypeOf()` methode. Sommige respondenten waren niet in staat hierover te redeneren.

De lengte van de if-then-else structuur zorgde ervoor dat er meer, en ook ernstigere fouten werden gemaakt betreffende recursie. In de conditionele versie zijn vijf recursieve fouten gemaakt, ten opzichte van twee in de polymorfe versie. In vier gevallen (twee in beide versies) ging het om de volgorde of het aantal van de aanroepen, maar in drie gevallen (allen in de conditionele versie) is aanwezige recursie niet opgemerkt.

Taak 4

In taak vier dienden de respondenten aan te geven wat de output was van een bepaalde aanroep, op basis van een variabele die in taak drie een waarde had gekregen. Hier was de polymorfe groep 22,5% langzamer, maar wel 21% correcter.

Bij de resultaten van deze taak is voor beide onderzoeksgroepen een tweedeling te zien in correcte en minder correcte programmeurs, dit is omdat er weinig nuance mogelijk was bij de beoordeling van deze taak: het antwoord was goed of niet.

Een opvallend punt aan de correctheid bij deze vraag was dat de respondenten die in de vorige taak een slechte score hadden, niet in staat waren zich te verbeteren. Enkele respondenten hadden het idee dat deze taak een strikvraag was, waarna ze hun antwoord op de vorige taak herhaalden.

Taak 5

Bij taak vijf was de opdracht om te bepalen hoe twee subtypes van `Type` zich met elkaar verhouden. Het was voor beide groepen een korte opdracht, gemiddeld waren respondenten vier minuten bezig. De correctheid was hoog, uit een maximale score van 4 werden gemiddeld 3,2 punten behaald in het conditionele systeem en 3,7 punten in het polymorfe systeem.

In het polymorfe systeem verwachtten respondenten delokalisatie, er werd dan ook begonnen met zoeken in de subtypes. Door dit verschil in werkwijze maakten ze minder vaak een verkeerde methode fout. Het verschil in de werkwijze kan worden verklaard doordat `Type` in het conditionele systeem meer verantwoordelijkheden heeft. Hierdoor wordt de zoektocht hierop geconcentreerd, en in sommige gevallen niet verder gekeken.

Doordat de methode in het subtype hetzelfde gedrag vertoonde als de methode in `Type`, was er weinig verschil in correctheid.

Taak 6

In taak zes was de opdracht hetzelfde als in taak vijf, namelijk om te bepalen hoe twee types zich met elkaar verhouden. In dit geval gaat het echter om een complexere situatie, waarbij een `AliasType` is gebruikt die verwijst naar een ander type. Deze taak had een diepere calltree dan taak vijf.

Het uitvoeren van de taak kostte in het polymorfe systeem ruim zeven minuten, dit is 18% langzamer. Wanneer wordt gekeken naar de correctheid (figuur 5f) komt er een ander beeld naar boven, in het polymorfe systeem werd een 61,9% hogere correctheid behaald (one tailed Student's t-test, df = 17, t = 2,4, p = 0,015).

De werkwijze van de respondenten was vergelijkbaar met taak 5, waarbij het verschil tussen de twee groepen groter is geworden. Een voorbeeld is het kijken naar het `AliasType`, een belangrijke klasse in deze taak. In het polymorfe systeem hebben alle respondenten deze klasse bekeken, maar in het conditionele systeem heeft slechts 70% van de respondenten het `AliasType` opgevraagd. Wanneer de gemaakte fouten per type worden bekeken valt op dat in de conditionele situatie drie maal een grove fout is gemaakt, dit is logisch als essentiële informatie niet is bekeken.

Kortom, door meer complexiteit toe te voegen aan een taak kijken programmeurs die aan een conditioneel opgezet systeem werken minder ver om zich heen, en missen hierdoor belangrijke kennis.

Taak 7

In taak zeven werd gevraagd hoe de respondent een `OrderedSetType` zou toevoegen. Een logische uitbreiding zou de bestaande implementatie van het `SetType` herbruiken. Aangezien het `SetType` in geen van de taken is genoemd, test deze taak dus in hoeverre er een situation model is opgebouwd.

In beide groepen bevatte 70% van de oplossingen het `SetType`. In het conditionele systeem werd een 11% hogere correctheid behaald, maar dit verschil is te klein om te zeggen dat werken aan het conditionele systeem ervoor zorgt dat deze taak beter is uitgevoerd.

Dit is opvallend, want het conditionele systeem bevat een if-then-else statement waarin alle subtypes, dus ook het `SetType`, voorkomen. Toch is hier weinig voordeel uit gehaald. Mogelijk komt dit door de complexiteit van de `isSubtypeOf()` methode, die voor een hoge cognitive load heeft gezorgd. Hierdoor lag de aandacht op het uitvoeren van de huidige taak, en vielen andere details (zoals het bestaan van een `SetType`) niet op.

In het polymorfe systeem konden respondenten de subtypes van `Type` te bekijken om te kijken of er al een soortgelijk type was. Deze werkwijze bleek bijna net zo efficiënt.

In termen van tijdsbesteding was de groep die aan het conditionele systeem heeft gewerkt iets langzamer, met tien tegenover acht minuten. Langer werken aan een systeem leidt tot een beter begrip hiervan, een trend die ook in andere taken zichtbaar is.

5.4 Expertise

In [1] bleken experts beter om te gaan een gedelokaliseerde control flow, deze zorgde ervoor dat ze het experiment 31% sneller voltooiden. In dit experiment was hypothese $H3$ dan ook dat experts beter om kunnen gaan met de delokalisatie, waardoor ze sneller zijn in het uitvoeren van typische onderhoudstaken in een systeem waarin veel polymorfisme wordt gebruikt.

Na het selecteren van de experts bleven er tien respondenten over. Hierdoor gaan individuele verschillen een grotere rol spelen, zo waren de experts in de polymorfe groep 5% sneller tijdens de controle taken. Tijdens de experimentele taken waren de experts slechts 4% sneller. Aangezien het verschil ongeveer hetzelfde bleef, kan de hypothese niet worden geaccepteerd. Dit is opvallend, want de trend die in een ander onderzoek te zien is komt hier dus niet terug. De verschillen kunnen te maken hebben met de respondenten of het systeem.

Respondenten Een reden voor het verschil kan zijn dat er, na de opdeling, weinig respondenten overblijven.

Een andere reden voor het verschil kan liggen in de externe kenmerken van de respondenten. Het aantal jaren werkervaring is vergelijkbaar, gemiddeld 6,3 tegenover 6,1 jaar ervaring. Het opleidingsniveau ligt in [1] lager: de experts hebben gemiddeld 1,7 jaar computer science gestudeerd, tegenover minimaal een bachelor diploma in dit onderzoek.

Systeem Het systeem kan ook de oorzaak zijn voor het verschil. Zo was er een groot verschil in het domein van de systemen: koffiezetapparaten tegenover feiten over broncode representeren. Een ander systeem kan leiden tot andere resultaten [25].

Een ander verschil was de grootte van het systeem. Waar dit systeem 175 klassen bevatte, was dit in [1] twaalf bij het DC ontwerp en zeven in het CC ontwerp.

5.5 Validiteit

De bedreigingen voor de validiteit zijn op te splitsen in drie punten, namelijk de constructvaliditeit, de interne validiteit en de externe validiteit.

5.5.1 Constructvaliditeit

De constructvaliditeit gaat over de vraag of de resultaten van een onderzoek een indicatie zijn voor het begrip waarover een uitspraak wordt gedaan.

Understandability is een belangrijk construct in dit onderzoek. Understandability is moeilijk te meten, bij een onderzoek waar al het contact via de browser verloopt kunnen de gedachten van een persoon niet worden gelezen. Als indicatoren voor understandability zijn de bestede tijd en de behaalde correctheid gebruikt. Aangenomen dat deze variabelen correct zijn gemeten (zie hiervoor de interne validiteit), zijn dit twee belangrijke indicatoren voor understandability. De bestede tijd en de behaalde correctheid zijn ook in andere studies gebruikt, zowel in het veld van software engineering [1, 7] als in de cognitieve psychologie [31].

Echter, doordat de duur van het experiment beperkt is, meet dit alleen de understandability op korte termijn. Het is mogelijk dat aanpassingen op een van de systemen resulteren in code van hogere kwaliteit, waardoor de understandability op langere

termijn wordt beïnvloed. Dit is een mogelijk onderwerp voor toekomstig onderzoek.

5.5.2 Interne validiteit

De interne validiteit betreft in hoeverre de veranderingen in de afhankelijke variabelen kunnen worden toegeschreven aan veranderingen in de onafhankelijke variabelen. Deze is opgesplitst in interne validiteit betreffende respondenten, taken en beoordeling.

Respondenten Er zijn verschillende bedreigingen voor de interne validiteit betreffende de respondenten. Zo zou het kunnen zijn dat de externe kenmerken van de beide groepen niet in balans waren. Hiermee is rekening gehouden, zie hiervoor ook paragraaf 3.2.

Daarnaast kan het zijn dat de respondenten onvoldoende waren gemotiveerd, of met onderbrekingen aan het experiment hebben meegedaan. Hiermee is rekening gehouden bij het beoordelen van de resultaten, zie ook paragraaf 4.1.

Ook zou er een bias kunnen ontstaan als respondenten op de hoogte waren van het doel van het experiment. Ondanks dat het doel niet is gecommuniceerd, kan niet worden uitgesloten dat enkele respondenten het hebben geraden.

Taken De taken zouden een bias kunnen creëren voor een van de twee implementaties, om deze dreiging te minimaliseren is een bestaand framework [21] gebruikt voor het opstellen van de taken. Daarnaast zijn de taken opgesteld op basis van commit berichten in het versiebeheersysteem en met feedback van een auteur van het systeem, waardoor de taken ook taken zijn die zijn uitgevoerd op het systeem.

Beoordeling Ten eerste is het mogelijk dat de gemeten tijd incorrect is. Het kan zijn dat iemand iets te drinken heeft gehaald, heeft gewacht met het opsturen van het antwoord of in een eerdere taak tijd heeft besteed om achter informatie te komen die in een volgende taak nodig was. Deze afwijkingen kunnen een bias hebben geïntroduceerd in de resultaten.

Ten tweede zou het kunnen dat de correctheid incorrect is toegekend. Om dit tegen te gaan is, zoals in [7], een oplossingsmodel opgesteld, waardoor het beoordelen van een antwoord een kwestie werd van kijken of een antwoord bepaalde elementen bevatte. Dit oplossingsmodel is aangepast op basis van de antwoorden die tijdens het testen van het onderzoek zijn gegeven.

Ten derde is het zo dat de typering van fouten pas is opgesteld nadat de resultaten zijn bekeken. Door de lage interne validiteit hiervan zijn deze enkel gebruikt als leidraad in de discussie. Voor iedere fout die is gemaakt is er wel een fout toegekend, de set met types was dus compleet.

5.5.3 Externe validiteit

De externe validiteit is in hoeverre de resultaten uit dit experiment kunnen worden gegeneraliseerd. Deze is opgedeeld in bedreigingen met betrekking op de respondenten, de tool, de taken en het systeem.

Respondenten Het eerste punt is de steekproefomvang. Een hogere steekproefomvang zorgt ervoor dat er met grotere precisie uitspraken kunnen worden gedaan over de populatie. Het bleek niet mogelijk om meer dan twintig programmeurs te motiveren om aan het experiment deel te nemen, hierdoor kan niet worden gegeneraliseerd naar “de programmeur”.

Het tweede punt is dat de steekproef niet aselect is. Zo hadden alle respondenten de Nederlandse nationaliteit. Daarnaast is het moeilijk om professionele programmeurs te motiveren om een uur van hun tijd op te geven voor een experiment. Toch is het gemiddelde ervaringsniveau, 5 jaar, hoger dan sommige studies. Ook ligt de de hoogst behaalde opleiding hoger dan in andere experimenten, vijf respondenten hadden een master diploma, veertien respondenten hadden een bachelor diploma en waren bezig met hun master, en één een respondent was bezig met zijn bachelor. De resultaten van dit experiment kunnen dus in principe worden gegeneraliseerd naar Nederlandse hoogopgeleide programmeurs.

Het derde punt is de bekendheid met het systeem. Het zou kunnen dat respondenten met meer ervaring in het systeem andere resultaten opleveren. Er kan dus alleen worden gegeneraliseerd naar understandability op de korte termijn.

Tool In een gecontroleerd experiment is er sprake van een tradeoff tussen realisme (externe validiteit) en controle (interne validiteit). De keuze voor het gebruiken van dezelfde tool voor alle respondenten verhoogt de interne validiteit, alle respondenten hebben dezelfde functionaliteit en leereffecten.

Een nadeel voor de interne validiteit is echter dat de tool gemaakt is met kennis van de experimentele opzet, het is niet uit te sluiten dat hier een bias is ontstaan. Ook is dit een bedreiging voor de externe validiteit, want in de praktijk gebruiken programmeurs tools waar ze al bekend mee zijn.

Om de validiteit te verhogen is rekening gehouden met een framework voor *software exploration tools* [31], dit is beschreven in paragraaf 3.6.1. Daarnaast is, om de leereffecten te verminderen, een korte tutorial toegevoegd om uit te leggen hoe de tool werkt.

Taken De taken zijn opgesteld om in ongeveer een uur uitvoerbaar te zijn, waardoor deze kleiner zijn dan in een praktijksituatie. Ondanks dat feedback na het experiment aangaf dat de complexiteit van de taken hoog was, kan niet worden uitgesloten dat met langere taken een ander beeld was ontstaan.

Behalve de lengte zijn de taken wel realistisch voor een onderhoudssituatie, zie hiervoor ook de interne validiteit.

Systeem Het zou kunnen dat het bekijken van een tweede systeem andere resultaten oplevert [7]. Toch is ervoor gekozen om één systeem te bekijken, op deze manier wordt er zo weinig mogelijk gevraagd van de respondenten.

Daarnaast is het mogelijk dat de systemen niet representatief zijn. Het polymorfe systeem is representatief, want deze is gebaseerd op objectgeoriënteerde principes.

Bij het conditionele systeem zijn echter twee vraagtekens te zetten. Ten eerste is het conditionele systeem gemaakt op basis van het polymorfe systeem, waarbij nét genoeg is aangepast om een effect te genereren. Hierdoor is het systeem mogelijk niet representatief voor conditionele systemen.

Ten tweede is het systeem aangepast met kennis van de experimentele opzet, er is dus niet uit te sluiten dat er een bias is ontstaan. Het maken van de conditionele versie is wel zo nauwkeurig mogelijk gedaan, zo was er lichte code duplicatie in enkele subtypes die kon worden opgelost.

6 Conclusie

In een gecontroleerd experiment is de understandability van polymorfisme vergeleken met de understandability van conditionele logica. Er zijn in twee systemen typische onderhoudstaken uitgevoerd, waarbij de bestede tijd en behaalde correctheid is gemeten. Op basis van een polymorf systeem is een conditioneel systeem gemaakt, waarbij een klasse hiërarchie deels geïmplementeerd is door middel van een if-then-else statement.

Uit de resultaten blijkt dat er geen effect is op de bestede tijd. Het kostte evenveel tijd om de complexiteit te begrijpen als om om te gaan met de delokalisatie in het polymorfe systeem.

De correctheid was 16% hoger in het polymorfe systeem. Ondanks dat dit verschil niet statistisch significant is, geeft het een goede indicatie dat, zeker tijdens complexere taken, er een hogere correctheid wordt behaald bij het werken aan polymorfe systemen.

De resultaten van individuele taken zijn ook geanalyseerd, dit geeft een indicatie van de verschillen tussen de groepen. Bij het uitvoeren van eerste taken binnen een conditioneel systeem ontstaan grotere prestatie verschillen tussen programmeurs dan in een polymorf systeem. In latere taken kan bij programmeurs in een conditioneel systeem een tunnelvisie ontstaan, die duidelijker naar voren komt bij het uitvoeren van complexe taken. Hierbij wordt meer aandacht gericht op het if-then-else statement en minder op de rest van het systeem. Tot slot zijn programmeurs in een conditioneel systeem iets beter in een taak die een onbekend subtype betreft, omdat in het if-then-else statement veel van de subtypen worden genoemd.

Dit onderzoek draagt het volgende bij:

- Een omgeving om online experimenten af te nemen.
- Een set van onderhoudstaken met gevalideerd antwoord model voor een open source systeem.
- De indicatie dat de factor complexiteit zwaarder weegt dan delokalisatie, samen met een kwalitatieve analyse naar de oorzaken hiervan.

6.1 Toekomstig onderzoek

Conditionele logica is niet de enige manier waarop polymorfisme, en de delokalisatie die dit met zich meebrengt, te vervangen. Toekomstig onderzoek zou zich kunnen richten op andere manieren om polymorfisme te vervangen. Ook kan de factor delokalisatie mogelijk worden verminderd door verbeteringen in IDE's door te voeren.

Zoals genoemd in paragraaf 5.5.3 is het mogelijk dat er een ander beeld ontstaat wanneer op een ander systeem wordt onderzocht, hiervoor zou dit onderzoek moeten worden gerepliceerd in een systeem uit een ander domein.

6.2 Dankwoord

Allereerst wil ik de leden van het SEN1 team bedanken voor de prettige werksfeer. In het specifiek wil ik graag mijn afstudeerbegeleider Jurgen Vinju bedanken voor zijn scherpe en nuttige feedback. Ook bedank ik mede studenten David Walschots, Jelle Geelhoed en Waruzjan Shahbazian voor hun help aan dit onderzoek. Daarnaast bedank ik Paul Griffioen, wie me heeft geholpen om deze scriptie te verduidelijken. Ook wil ik alle testers en participanten van het onderzoek bedanken voor hun kostbare tijd. Tot slot wil ik graag mijn vriendin bedanken voor haar eeuwige geduld.

Referenties

[1] Erik Arisholm and Dag I. K. Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):521–534, 2004.

[2] Patricia Baggett. Structurally equivalent stories in movie and text and the effect of the medium on recall. *Journal of Verbal Learning and Verbal Behavior*, 18(3):333 – 356, 1979.

[3] Victor R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3 – 12, 1997. Achieving Quality in Software.

[4] Martin R. K. Baumann and Josef F. Krems. A comprehension based cognitive model of situation awareness. In Vincent G. Duffy, editor, *HCI (11)*, volume 5620 of *Lecture Notes in Computer Science*, pages 192–201. Springer, 2009.

[5] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[6] Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40(14):795–799, 1998.

[7] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[8] Cynthia L. Corritore and Susan Wiedenbeck. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54(1):1 – 23, 2001.

[9] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1996.

[10] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating the effect of inheritance on the maintainability of object oriented software. 1996.

[11] Ignatios S. Deligiannis, Martin Shepperd, Steve Webster, and Manos Roumeliotis. A review of experimental investigations into object-oriented technology. *Empirical Softw. Engg.*, 7(3):193–231, 2002.

[12] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[13] Bart Du Bois, Serge Demeyer, and Jan Verelst. Does the "refactor to understand" reverse engineering pattern improve program comprehension? In *CSMR '05: Proceedings of the Ninth European Conference on Software*

Maintenance and Reengineering, pages 334–343, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997.

[15] Martin Fowler et al. *Refactoring. Improving the design of existing code.* 1999.

[16] Gordon L. Freeman, Jr. *The task-dependent nature of maintenance of object-oriented programming: an empirical investigation.* PhD thesis, Nashville, TN, USA, 2003. Director-Schach, Stephen R.

[17] Amela Karahasanović, Annette Kristin Levine, and Richard Thomas. Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *J. Syst. Softw.*, 80(9):1541–1559, 2007.

[18] Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 125–130, New York, NY, USA, 1991. ACM.

[19] George Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information, 1956. One of the 100 most influential papers in cognitive science.

[20] Carol Bergfeld Mills, Virginia A. Diehlb, Deborah P. Birkmirec, and Lien-Chong Mou. Reading procedural texts: Effects of purpose for reading and predictions of reading comprehension models. 1995.

[21] Michael J. Pacione, Marc Roper, and Murray Wood. A novel software visualisation model to support software comprehension. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 70–79, Washington, DC, USA, 2004. IEEE Computer Society.

[22] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. 1987.

[23] Carl Ponder and Bill Bush. Polymorphism considered harmful. *SIGSOFT Softw. Eng. Notes*, 19(2):35–37, 1994.

[24] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. A controlled experiment on inheritance depth as a cost factor for code maintenance. *J. Syst. Softw.*, 65(2):115–126, 2003.

[25] Jochen Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 73–82, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Vennila Ramalingam and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 124–139, New York, NY, USA, 1997. ACM.

[27] Wolfgang Schneider and Joachim Körkel. The knowledge base and text recall: Evidence from a short-term longitudinal study. *Contemporary Educational Psychology*, 14(4):382 – 393, 1989.

[28] B. Shneiderman. Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences*, 5(2):123–143, June 1976.

[29] B Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3), XXX 1979.

[30] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.

[31] Margaret-Anne Darragh Storey. *A cognitive framework for describing and evaluating software exploration tools.* PhD thesis, Burnaby,

BC, Canada, Canada, 1998. Adviser-Fracchia, David and Adviser-Muller, Hausi.

[32] Teun A. van Dijk and W. Kintsch. *Strategies of discourse comprehension*. New York: Academic Press, 1983.

[33] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 39–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[34] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tools capabilities. In *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering,* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 230–239, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[35] Susan Weidenbeck. Processes in computer program comprehension. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 48–57, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[36] R. Wood. Assisted Software Comprehension. *Final report, June*, 2003.