Evaluatie van de bijdrage van refactoring voor onderhoudbaarheid

Bart Du Bois
Lab On REengineering
Universiteit Antwerpen
Middelheimlaan 1, B-2020 Antwerpen
bart.dubois@ua.ac.be

Promotor: Prof. S. Demeyer
Co-promotor: Prof. J. Verelst

19 oktober 2003

1 Probleemstelling

> Progress comes when what is actually true can be separated from what is only believed to be true – Victor R. Basili [1]

Noodzaak Binnen het domein van reengineering is er consensus dat refactoring – gedragsbehoudende codeherstructurering – de enige fundamentele oplossing vormt voor het aanpakken van legacy systemen. De alternatieven, wrapping en herschrijven van nul, richten zich niet tot de kern van het probleem. Als herstructureringstechniek wordt refactoring daarom reeds meer dan een decennium onderzocht [2, 3, 4, 5]. Een tastbaar resultaat hiervan is de beschrijving van fundamentele operaties als basisblokken voor het herstructureren van software [6]. De huidige generatie software tools ondersteunen deze, hetwelk de software ontwikkelaar aanzienlijk helpt in het gecontroleerd onderhouden van software systemen. Voor een optimale benutting van deze operaties als engineering-techniek is validatie strikt noodzakelijk, zoals erkend in [7, 8, 9]. Deze noodzaak vormt de kern van dit onderzoek.

Beoogd proces *Perfectief onderhoud* omvat het verbeteren van een software systeem in termen van functionele (functies, taken, gedrag) en niet-functionele (attributen van functies, taken, gedrag) vereisten. Vooral in legacy systemen vormt lage perfectieve onderhoudbaarheid de hoofdaanzet tot het investeren in herstructurering van de broncode.

Het *opmeten* van perfectieve onderhoudbaarheid is mogelijk door het effectief *uitvoeren van representatieve perfectieve onderhoudstaken,* waarbij de uitvoeringssnelheid en correctheid gemeten worden. Hiernaast gebruiken software ontwikkelaars alternatieve indicatoren voor deze kwaliteitsmaat [10]. Dit leidde tot het vastleggen van zogenaamde code smells en anti-patterns, die codestructuren beschrijven die slecht onderhoudbaar zijn. Om deze probleemgebieden in de broncode te identificeren steunt men op de berekening van *interne programma attributen*, dewelke af te leiden zijn uit het software product zelf [10, 11, 12].

Goede herstructurering vormt deze probleemgebieden om tot implementaties die beter onderhoudbaar zijn, en m.a.w. de overeenkomende programma attributen verbeteren. Het belang van deze attributen binnen het refactoringproces toonden we aan in een overzicht van de huidige onderzoeksresultaten en toekomstige onderzoeksvragen over refactoring [13]. Voor adequate herstructurering moeten refactoring-tools daardoor rekening houden met deze indicatoren.

Situering in gerelateerd onderzoek Het wetenschappelijk testen van de bijdrage van technieken bleef tot begin jaren ’90 binnen het Software Engineering domein uiterst beperkt. Op 10 jaar tijd is dit domein echter zeer snel gegroeid, zoals te merken aan het grote aantal conferenties en journals die zich expliciet concentreren op het toetsen van rigoureuze wetenschappelijke methoden voor software ontwikkeling en onderhoud (ISESE, EASE, ICSE, WESS, Empirical SE: an International Journal, TSE...). Gerelateerde validaties van technieken binnen Software Engineering zijn [14, 15, 16, 17]. Als bijdrage aan de toepassing van refactoring sluit dit werk aan bij eerder onderzoek over codeherstructureringen [2, 18, 3, 6, 4, 19, 4, 5] en het gebruik van programma attributen voor het opsporen en verhelpen van probleemgebieden in de broncode [10, 11, 20, 21].

2 Doelstellingen

Het doel van het project is het wetenschappelijk testen van de bijdrage van *refactoring* als techniek voor de verbetering van *perfectief onderhoud*. De gebruikte onderzoeksmethodologie is opgedeeld in drie delen:

Analytisch onderzoek Voor het opmeten van de inwerking van refactoring op *interne programma attributen* (metrieken) bouwden we een formeel model op voor programma structuur. Het beschrijven van refactorings en metrieken in termen van dit hoger abstractie-niveau laat toe om de relatie tussen beiden te analyseren, onafhankelijk van instanties van dit model, t.t.z. de specifieke broncode. De grondbeginselen en eerste resultaten voor dit onderzoek bespraken we in [22]. De automatisering van deze analyse beschreven we in [23], hetwelk toeliet deze inwerking te traceren op basis van logische feiten, regels en queries. Voor deze geautomatiseerde analyse zijn de fundamenten m.a.w. succesvol gelegd. De volgende stap is het verder formaliseren van meer refactorings en metrieken, als invoer voor de automatische analyse (sectie 3.2).

Empirisch onderzoek Voor het opmeten van de inwerking van refactoring op *effectieve uitvoeringssnelheid en -correctheid van representatieve perfectieve onderhoudstaken* bouwden we een experimenteel ontwerp op, in samenwerking met experts inzake experimenteel onderzoek in Software Engineering. Hiervoor verbleef ik bij het SIMULA research laboratory in Oslo, Noorwegen, hetwelk de fundamenten voor het uitvoeren van realistische experimenten opleverde (sectie 3.3). In deze experimenten onderzoeken we *refactoringstrategieën* en *architecturale stijlen* als onafhankelijke variabelen, *representatieve onderhoudstaken* als experimentele taken en *uitvoeringssnelheid en -correctheid* als afhankelijke variabelen.

Verificatie op basis van onderhoudskostschattingen De resultaten in termen van interne programma attributen en uitvoeringssnelheid en -correctheid van perfectieve onderhoudstaken verifiëren we op basis van twee technieken voor schattingen in onderhoudskost. Deze technieken worden beiden in de praktijk gebruikt, en zijn complementair. *Ramingsmodellen* relateren interne programma attributen met externe kwaliteitsmaten. *Expertschattingen* steunen op ervaring om externe kwaliteitsmaten op te meten op basis van de broncode. De inwerking van refactoring op deze schattingen verifieert de resultaten van het analytische en empirische onderzoek (sectie 3.4).

De investeringen in de realisatie van refactoring-tools leidden de laatste jaren tot aanzienlijke ondersteuning voor het uitvoeren van refactorings in ontwikkelingsomgevingen (IDE's TogetherSoft ControlCenter, Eclipse, Netbeans, CodeGuide...). Als validatie van de bijdrage van deze techniek voor onderhoudbaarheid heeft dit project een uiterst innovatief karakter. Zonder de resultaten van dit onderzoek worden refactorings ter verbetering van onderhoudbaarheid toegepast op basis van intuïtie, hetwelk ontegensprekelijk feilbaar is. Voor het uitbouwen van refactoring als mature engineering-techniek is dit werk daarom onmisbaar.

Het projectonderwerp behoort tot het kernonderzoek van de onderzoeksgroep Lab On REengineering, zoals concreet aan te tonen in gepubliceerd werk over refactorings [18, 19, 4, 24, 13, 25, 22, 26] en metrieken [20, 21, 27].

3 Projectbeschrijving

> Software Engineering is the application of a systematic, disciplined, *quantifiable* approach to the development, operation and *maintenance* of software.
> – IEEE Standard 610.12

Het project omkadert een onderzoeksmethodologie voor het *kwantificeren* van de bijdrage van refactoring voor perfectieve onderhoudbaarheid. Deze inwerking onderzoeken we vanuit twee perspectieven, waarvan we de resultaten met een derde methode verifiëren.

Het projectvoorstel is als volgt gestructureerd. In (3.2) bespreken we het *analytisch* onderzoek, dat de inwerking in termen van *interne programma attributen* (metrieken voor complexiteit, koppeling, cohesie...) opmeet. In (3.3) bespreken we het *empirisch* onderzoek, dat de inwerking in termen van *uitvoeringssnelheid en -correctheid* van perfectieve onderhoudstaken opmeet. De verificatie van de resultaten van beide methoden bespreken we in (3.4), dat de inwerking in termen van onderhoudskostschattingen via (3.4.1) *ramingsmodellen* en (3.4.2) *expertschattingen* opmeet.

Het gebruik van verschillende onderzoeksmethoden om hetzelfde fenomeen te bestuderen laat toe de resultatenbetrouwbaarheid van elk van de gebruikte onderzoeksmethoden te analyseren. Inconsistenties tussen de resultaten verzwakken de betrouwbaarheid van beide resultaten en wijzen op externe variabelen, terwijl overeenkomst het vertrouwen in de resultaten verhoogt.

3.1 Onderhoudbaarheid als softwarekarakteristiek

In de literatuur wordt software kwaliteit onderverdeeld in karakteristieken en subkarakteristieken volgens het zogenaamde Factor-Criteria-Metrics model [28]. Verschillende organisaties en auteurs stelden dergelijke kwaliteitsraamwerken voor. Zo delen ze *onderhoudbaarheid* op in volgende subkarakteristieken:

- ISO 9126: Analysability, changeability, stability en testability [29].
- McCall: Consistency, simplicity, conciseness [28].
- IEEE: Correctability, testability en expandability [30].

Deze constructen zijn echter niet rechtstreeks uit het software product meetbaar. Wel kunnen we interne programma attributen opmeten. Bovendien bestaan er programmeertaal onafhankelijke programma attributen, binnen een specifiek paradigma. Daarom vormen deze attributen fundamentele eigenschappen van softwareproducten binnen dat paradigma, en niet louter artificële programmeertaal-details.

Als indicatoren voor fundamentele eigenschappen van softwareproducten kan een extreme waarde voor een programma attribuut twee betekenissen hebben. Ofwel is er sprake van *goede* modellering, en geeft deze extreme waarde een foutieve indicatie van onderhoudbaarheid. Ofwel is er sprake van *slechte* modellering, en houdt de verbetering van deze probleemsituatie inherent een verlaging van de waarde van dit programma attribuut in.

De correlatie van interne programma attributen met kwaliteitsattributen is onderwerp van bestaand validatie-onderzoek, dat op een zeer laag tempo correlaties bevestigt of ontkracht [31, 32, 33].

Onafhankelijk van deze correlatie is het essentieel de inwerking van refactorings op deze programma attributen op te meten, teneinde herstructureringsoperaties te identificeren die de

probleemsituatie verbeteren, en daardoor inherent de waarde van het programma attribuut verlagen.

3.2 Analytisch onderzoek inzake verandering van interne programma attributen

In dit deel van het onderzoek zoeken we naar een open methodologie voor het nagaan van de wijze waarop refactorings interne programma attributen veranderen. Aangezien we deze wijzigingen onafhankelijk van de broncode willen beschrijven, is het nodig een abstractie-niveau hoger te gaan.

3.2.1 Conceptuele opbouw

Huidige tools voor het uitvoeren van refactorings of het berekenen van metrieken werken op het *instantie-niveau*. Hiermee bedoelen we dat ze werken op de broncode, die uiteraard voor elk programma anders is. Door een abstractie te maken tot op het *model-niveau*, verkrijgen we een generische beschrijving van de software. Een dergelijk model beschrijft de software als bestaande uit software objecten (packages, klassen, methoden...) en relaties tussen deze objecten (overerving, methode-oproep, compositie...).

Gesteund op een dergelijk model voor programmastructuur kunnen we redeneren over refactorings als transformaties op instanties van dit model, en metrieken als wiskundige berekeningen op instanties van dit model.

- Door te redeneren over *refactorings* op dit model-niveau zijn de resultaten onafhankelijk van de specifieke broncode. Deze redenering steunt op de model-elementen uit het hogere abstractie-niveau. Een voorbeeld maakt dit duidelijker. De Extract Method refactoring introduceert een nieuwe methode met gepaste parameters, en verhuist de betreffende statements naar deze nieuwe methode. Deze nieuwe methode wordt dan opgeroepen vanuit de originele methode. Een dergelijke redenering is uitsluitend gebaseerd op de software objecten en relaties uit het model-niveau, en zal dus voor elke instantie, m.a.w. voor elke specifieke broncode gelden.

- Analoog kunnen we over *metrieken* redeneren op dit model-niveau. Hierbij steunen we opnieuw op de model-elementen uit het hogere abstractie-niveau. Zo kunnen we de koppelingsmetriek Coupling Between Objects voor klasse X operationeel definiëren als de cardinaliteit van de verzameling unieke klassen (software objecten) die vanuit klasse X gerefereerd worden (relatie tussen software objecten). Deze beschrijving is onafhankelijk van de instanties van het programma-model (specifieke broncode), en is uitsluitend opgebouwd in termen van software objecten en relaties tussen deze objecten.

- Door de *inwerking* van deze transformaties (refactorings) op de wiskundige berekeningen op dit model (metrieken) te *traceren*, kunnen we een broncode-onafhankelijke beschrijving geven van de wijze waarop refactorings interne metrieken veranderen.

3.2.2 Vordering

De grondbeginselen voor het analytisch onderzoek zijn reeds gelegd in [22]. Ter illustratie toont Tabel 1 de resultaten van het traceren van de impact van refactorings Extract Method, Encapsulate Field en Pull Up Method op interne metrieken Number Of Methods (NOM), Number Of

Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC) en Lack Of Cohesion among Methods (LCOM).

Refactoring	**NOM**	**NOC**	**CBO**	**RFC**	**LCOM**
EncapsulateField	+	0	0	+	+
PullUpMethod subclass	-	0	-	-	-
PullUpMethod superclass	+	0	+	+	+
ExtractMethod	+	0	0	+	+

Tabel 1: Opdeling van de invloed van refactorings op interne metrieken in een potentiale positieve, negatieve of nil-impact (resp. +,- of 0).

Aangezien deze methodologie geen beperking stelt aan het type refactorings dan wel interne metrieken, beschikken we zo over een open methodologie. Bovendien steunt deze methodologie op een formele basis, waardoor het traceren van deze impact te *automatiseren* is. Deze automatisering werd reeds succesvol toegepast m.b.v. een logische redenerings-engine (Prolog). Hiervoor beschreven we de de impact van refactorings op model-elementen en de bijdrage van model-elementen op de wiskundige berekening van een metriek als *logische feiten*. Het combineren en aaneenschakelen van verschillende impact-feiten beschreven we als *logische regels*. Het uiteindelijke traceren van de impact van een refactoring op een metriek beschreven we als een *logische query* [23].

De integratie van deze informatie in refactoring-tools zoals Together ControlCenter, Eclipse, codeGuide, IntelliJ IDEA, etc. geeft de programmeur de mogelijkheid om vóór het uitvoeren van de refactoring in te schatten hoe specifieke interne programma attributen zullen beïnvoed worden. Daardoor kan zij/hij beter inschatten of de toepassing van deze herstructureringsoperatie een verstandige stap in het pad naar de oplossing van de probleemsituatie is.

3.3 Empirisch onderzoek inzake verandering in onderhoudskost

In het empirische luik van dit project willen we onderzoeken hoe de uitvoeringsduur en uitvoeringscorrectheid van perfectieve onderhoudstaken varieëert als functie van het toepassen van refactorings op software systemen. Allereerst beschrijven we de concepten *refactoring, software systeem* en *onderhoudstaak*, waarmee we een conceptueel raamwerk opbouwen. Gesteund op voorgaand onderzoek over en ervaring met refactoring kunnen we gefundeerde hypothesen op te bouwen die we kunnen testen m.b.v. *manipulatie-experimenten*.

3.3.1 Typologieën binnen het conceptueel raamwerk

Ter voorbereiding van het experimenteel onderzoek delen we het universum van elk van de concepten refactoring, software systeem en perfectieve onderhoudstaak op volgens gepaste typologieën. Zo een typologie gebruiken we voor het opstellen van het experimenteel ontwerp om representatieve manipulaties van de verschillende factoren op te zetten.

Refactorings Een refactoring is een gedragsbehoudende codetransformatie die variabelen, statements, attributen, methoden en klassen herindeelt over de klassehiërarchie. Daardoor kunnen refactorings ingedeeld worden volgens hun specifiek doel: {Composing Methods, Moving Features Between Objects, Organizing Data, Simplifying Conditional Expressions, Making Method Calls Simpler, Dealing with Generalization en Big Refactorings} [6].

Vanuit het praktijkperspectief kunnen we deze fijnschalige indeling groeperen in zogenaamde *refactoringstrategieën*. Een refactoringstrategie is een methode om een software systeem te herstructureren, en wordt geassocieerd met een beslissingspunt in de kostenbatenanalyse van het refactoringproces. Deze beslissingspunten zijn o.a *schaal* (ontwerp, implementatie) en *kwaliteitsobjectief* (ruw in te delen in performantie en onderhoudbaarheid). Terwijl we in het analytische luik ingaan op het laatste beslissingspunt (kwaliteitsobjectief) onderzoeken we in dit luik het beslissingspunt schaal. Hierdoor groeperen we de refactorings in de refactoringstrategieën *ontwerp-gericht herstructureren* en *implementatiegericht herstructureren*.

Software Systemen Net als alle andere constructies worden software systemen gekarakteriseerd door hun structurele opbouw. Deze analogie leidde tot de opstelling van *Architecturale Stijlen* [34]. Een architecturale stijl wordt beschreven op basis van zijn topologie, data- en controle-interactie tussen de componenten en de voor- en delen van het gebruik van deze stijl. Ze definiëren klassen van ontwerpen met geassocieerde eigenschappen. Het indelen van software systemen in families op basis van hun architecturale stijl maakt het mogelijk representatieve voorbeelden van deze stijl te gebruiken in experimenteel onderzoek. De meest verspreide stijlen zijn *Layered Systems, Interpreters* en *Batch Sequential* [34].

Perfectieve Onderhoudstaken Deze kunnen we indelen op basis van hun interactie met de architectuur van het software systeem. De drie gradaties van interactie zijn a) taken met betrekking op *één component* binnen de architectuur; b) taken met betrekking op *meerdere componenten*, maar die de architectuur intact houden; en c) taken met betrekking op de *interactie van deze componenten*, waardoor ze de architectuur beïnvloeden [35].

Deze gradaties maken het mogelijk onderhoudstaken op verschillende architecturale stijlen met mekaar te vergelijken, wat essentieel is voor dit experimenteel onderzoek.

Op basis van deze typologieën beschrijven we de experimenten ter studie van de relaties tussen de concepten.

3.3.2 Hypothese-opbouw

Bij het bestuderen van de relaties binnen het conceptueel raamwerk zijn onze afhankelijke variabelen de *uitvoeringsduur en -correctheid van perfectieve onderhoudstaken*. De onafhankelijke variabelen zijn *refactoringstrategie* en *architecturale stijl*. Zo komen we tot volgende onderzoeksvragen (OV) die we kunnen instantiëren met de meest relevante varianten van beide afhankelijke variabelen.

OV_1 Welke invloed hebben refactoringstrategieën inzake de *uitvoeringsduur* van perfectieve onderhoudstaken op software systemen met specifieke architecturale stijlen?

OV_2 Welke invloed hebben refactoringstrategieën inzake de *uitvoeringscorrectheid* van perfectieve onderhoudstaken op software systemen met specifieke architecturale stijlen?

Zowel uitvoeringsduur als uitvoeringscorrectheid van perfectieve onderhoudstaken worden gemeten aangezien de duur van een incorrecte uitvoering onmogelijk vergeleken kan worden met de duur van een correcte uitvoering. Hierbij kiezen we voor de uitvoeringscorrectheid een ordinale schaal met 5 waarden, analoog aan [36] in $\{$*Opdracht Niet Begrepen, Verkeerd Antwoord, Juiste Denkpiste, Bijna Correct, Correct*$\}$.

Voor het testen van deze hypothesen gebruikt men *manipulatie-experimenten*, waarbij de factoren op een gecontroleerde manier gemanipuleerd worden en de invloed van deze manipulatie op de afhankelijke variabele wordt gemeten [37, 38].

3.3.3 Opbouwen van manipulatie-experimenten

Gebaseerd op de typologieën binnen het conceptueel raamwerk stelden we een *experimenteel ontwerp* op met hulp van experts inzake software engineering experimenten tijdens een onderzoeksverblijf aan het SIMULA research laboratory, Oslo, Noorwegen.

In dit experimenteel ontwerp worden representatieve voorbeelden van architecturale stijlen *behandeld* m.b.v. representatieve uitvoeringen van refactoringstrategieën, *vooraleer er onderhoudstaken op uitgevoerd worden*. De behandeling die geen refactoring uitvoert noemen we de *placebo*-behandeling. Het ontwerp is opgetekend in Tabel 2. Hierna voeren groepen van deelnemers per behandeld software systeem (versie) 3 perfectieve onderhoudstaken uit. De uitvoeringsduur en -correctheid van deze onderhoudstaken vormen meetbare variabelen voor de perfectieve onderhoudbaarheid van deze versies. Hierdoor is het mogelijk een beeld te vormen van de invloed van refactoringstrategieën op de perfectieve onderhoudbaarheid van software systemen.

Architecturale Stijl \ Behandeling	**Placebo**	**Behandeling$_1$**	**Behandeling$_2$**
AS $_1$	$Groep_1$	$Groep_2$	$Groep_3$
AS $_2$	$Groep_3$	$Groep_1$	$Groep_2$
AS $_3$	$Groep_2$	$Groep_3$	$Groep_1$

Tabel 2: Gebalanceerd experiment ontwerp tussen 3 architecturale stijlen en 3 behandelingen, waaronder een placebo-behandeling (geen refactoringstrategie toegepast).

De voordelen van dit experimenteel ontwerp zijn o.a. dat het weinig deelnemers vereist aangezien meteen alle combinaties tussen de bestudeerde factoren nagaat (factorial design), en dat elke deelnemer dient als zijn eigen controle (within-participants design). Aangezien er weinig deelnemers (bv. 8 per groep) vereist zijn voor statistische analyse van de resultaten van deze opstelling, is het haalbaar om naast licenciaat-studenten ook professionele software ontwikkelaars te laten deelnemen. Zo verkrijgen we een zicht op de invloed van dit verschil op de resultaten. Voor de deelname van professionele software ontwikkelaars aan de experimenten zijn reeds goede contacten gelegd binnen verschillende Vlaamse bedrijven.

3.3.4 Uitvoeren van een piloot-studie

Een *pilootstudie* vormt een eerste (gedeeltelijke) uitvoering, waarbij we de experimentele set-up testen en eventuele herzieningen plegen. Deze pilootstudie levert ons tevens de eerste datapunten, waaruit we gegevens kunnen afleiden die essentieel zijn voor de validiteit van de experimenten.

Zo verschaffen we ons een beeld van de standaardafwijking van de resultaten, waardoor we de ideale steekproef-grootte kunnen bepalen. Bovendien bekomen we zo een prestatie-evaluatie van de deelnemers. *Prestatie* behoort tot de lijst van eigenschappen waarmee we de deelnemers profileren, om via blocking op basis van deze eigenschappen een random verspreiding van deelnemers met specifieke eigenschappen te verkrijgen. Het verzamelen van gegevens over de relevante eigenschappen (ervaring met technologie en technieken, probleemoplossend vermogen...) van deelnemers gebeurt via *questionnaires*, die potentiële deelnemers invullen [39]. De

opstelling en analyse van dergelijke questionnaires hebben we reeds praktisch getest op professionele software ontwikkelaars. Het gebruik van questionnaires zowel vóór als na het experiment biedt een aanzienlijke bron van informatie bij de verwerking van de datapunten.

3.3.5 Concrete instantiatie experiment

Het besproken experimenteel ontwerp vormt een *sjabloon* dat via concrete instanties inzake refactoringstrategieën, architecturale stijlen en onderhoudstaken ingevuld kan worden. Gebaseerd op de indeling voor refactorings (volgens hun schaal in refactoringstrategieën), onderhoudstaken (volgens hun impact op de architectuur) en software systemen (volgens hun architecturale stijl) selecteren we representatieve voorbeelden.

Representatieve voorbeelden van software systemen Voor de selectie van software systemen steunen we op volgende criteria:

- Zeer uitgesproken architecturale stijl
- Open Source, als noodzaak voor uitvoeren onderhoudstaken door experiment-deelnemers
- Realistisch systeem, zowel in product (omvang $\pm$ 30 Kilo Lines Of code) als in proces (sterke ontwikkelaars-contributie)
- Implementatietaal Java, waarin alle bekende refactorings toepasbaar zijn.
- Leeftijd van het systeem, gerekend in versie-nummer

Deze selectie-procedure leidde tot representatieve software systemen als instanties voor Architecturale Stijlen (Tabel 3), een selectie van refactorings in twee instanties van refactoringstrategieën (Tabel 4) en onderhoudstaken als instanties van taken met verschillende impact op de architectuur, als ingedeeld in [35] (Tabel 5).

Architectural Style	Voorbeeldsysteem
Layered System	`Jakarta Tomcat`
Interpreter	`Mozilla Rhino`
Batch Sequential	`HSQLdb`

Tabel 3: Instantiatie van het hoger orde concept architecturale stijl.

Representatieve voorbeelden van refactoringstrategieën Een beslissingspunt in de kosten-baten-analyse van refactoring is o.m. de schaal van refactoring. Hierbij doelen we op het detailniveau dat binnen een bepaalde schaal bereikt kan worden. Zo zullen refactorings op *ontwerp-schaal* zich concentreren op compositie van klassen in packages, methoden en attributen in klassen en op overerving tussen verschillende klassen. Deze worden respectievelijk gewijzigd in de refactoringcategorieën Move Features Between objects en Deal with Generalization. Refactorings op *implementatie-schaal* concentreren zich dan weer eerder op de concrete compositie van en keuze uit data-typen, het gebruik van tijdelijke variabelen en het samenstellen van of opdelen in keuzes in de control-flow. De refactoringcategoriën Organize Data en Simplify Conditional Expressions richten zich respectievelijk op deze veranderingen.

Refactoringstrategie	**Refactoringcategorieën van Fowler**
Ontwerp-gericht refactoring	{Move Features Between Objects, Deal with Generalization}
Implementatie-gericht refactoring	{Organize Data, Simplify Conditional Expressions}

Tabel 4: Instantiatie van het hoger orde concept refactoringstrategie.

Representatieve voorbeelden van perfectieve onderhoudstaken Als behandeling van de software systemen zal de toepassing van deze refactoringstrategieën op geen enkele wijze beïnvloed worden door de specifieke invullingen voor onderhoudstaken. Daarom wordt er in deze fase dan ook niet gezocht naar onderhoudstaken voor het specifieke software systeem, maar enkel voor de specifieke architecturale stijl. Na voltooiing van de behandelingen (refactoring-proces) worden, gebaseerd op de concrete semantiek van het software systeem, instanties van deze onderhoudstaken gekozen. Deze onderhoudstaken, uit te voeren door de experiment-deelnemers, zijn voor alle versies (behandelde software systemen) gelijk. Een dergelijke opbouw vermijdt inspeling van de toepassing van refactorings op perfectieve onderhoudstaken.

Impact op architectuur	**Layered System**	**Interpreter**	**Batch Sequential**
1 eenheid	1 laag	1 van {Interpreter, Programma, Interpreter Toestand, Programma Toestand}	1 filter
$>$ *1 eenheid*	$>$ 1 laag	$>$ 1 van {Interpreter, Programma, Interpreter Toestand, Programma Toestand}	$>$ 1 filter
architectuur schadend	lagenmodel schaden	opdeling schaden	sequentieel model schaden

Tabel 5: Instantiatie van het hoger orde concept onderhoudstaak volgens hun impact op de architectuur.

Deelnemers worden willekeurig verspreid over de deelnemersgroepen, gebaseerd op basis van hun specifieke eigenschappen als beoordeeld door pre-experiment questionnaires en hun pre-experiment performantie-evaluatie (zowel in uitvoeringssnelheid als uitvoeringscorrectheid van analoge onderhoudstaken). Zo is de aanwezigheid van deelnemers met een specifiek deelnemersprofiel even groot voor elke groep. Onder andere voor het *leer-effect* is dit zeer belangrijk.

Uitvoering van het experiment Gebaseerd op het experimenteel ontwerp opgetekend in Tabel 2 worden zo drie groepen gevormd, en zullen de experimenten in een aantal fasen verlopen. Aangezien we in deze voorbeeld-instantiatie van het experimenteel ontwerp kiezen voor 3 behandelingen (*placebo*, *ontwerp-gericht refactoring* en *implementatie-gericht refactoring*) op 3 software systemen (`Jakarta Tomcat` als Layered System, `Mozilla Rhino` als interpreter, `HSQLdb` als Batch Sequential) krijgen we drie versies voor elk van de drie software systeem. Het ontwerp zorgt ervoor dat een deelnemersgroep nooit twee maal hetzelfde software systeem onderhoud, en een rotering van de groepen bij elke fase-overgang zorgt ervoor dat elke deelnemersgroep nooit twee maal onderhoud uitvoert op een versie met dezelfde behandeling.

Dit experiment kent daardoor volgende fasen:

- *Fase 1*: Hierbij is de opdracht drie onderhoudstaken op een versie van `Jakarta Tomcat`

uit te voeren. De verdeling van de versies aan de groepen gaat als volgt: {(1, placebo), (2, OR), (3, IR)}, waarbij OR en IR respectievelijk voor *Ontwerp-gericht Refactoring* en *Implementatie-gericht Refactoring* staan.

- *Fase 2*: Hierbij voeren de deelnemers drie onderhoudstaken uit op een versie van `Mozilla Rhino`. Versie-verdeling: {(1,OR),(2,IR),(3,placebo)}.
- *Fase 3*: Hierbij worden de versies van `HSQLdb` aan onderhoud onderworpen. Versie-verdeling: {(1,IR),(2,placebo),(3,OR)}.
- *Fase 4*: Extra fase ter vermijding van het *ceiling-effect* - aan het eind van een experiment veranderen de prestaties van een experiment-deelnemer. Een analoge opdracht op een extra software systeem dient voor het opvangen van meer performante deelnemers.

De meting voor elke uitvoering van een onderhoudstaak door elke deelnemer zorgt voor een totaal van 216 datapunten voor uitvoeringssnelheid en 216 datapunten voor uitvoeringscorrectheid. Dit komt overeen met 24 datapunten voor zowel de snelheid als de correctheid van de uitvoering van een specifieke onderhoudstaak voor een specifiek software systeem. Deze datapunten zullen met gepaste statistische technieken worden verwerkt.

3.4 Gebruik van onderhoudskostschattingen

Ter verificatie van de resultaten van het analytische en empirische onderzoek inzake de inwerking van refactoring op onderhoudbaarheid gebruiken we in dit deel van het onderzoek *onderhoudskostschattingen*. In de praktijk steunt men op deze kostenramingen ter ondersteuning van beslissingen in het onderhoudsproces. Uit literatuuronderzoek blijkt dat een combinatie van het gebruik van ramingsmodellen en expertschattingen de meest correcte schattingen oplevert [40].

We bespreken het gebruik van elke techniek apart, volgens motivatie, doelstellingen en aanpak.

3.4.1 Ramingsmodellen inzake onderhoudskost

Motivatie In het analytisch onderzoek gaan we de impact na van refactorings op interne programma attributen. Op basis van ramingsmodellen die deze attributen als variabelen voor een kostenmodel gebruiken kunnen we de analytische resultaten verifiëren. Bovendien steunen verschillende ramingsmodellen op architecturale informatie, dat als factor in het empirisch onderzoek werd geïncludeerd. Dit laat een verificatie van de empirische resultaten toe.

Doelstellingen Op basis van duidelijke criteria selecteren we ramingsmodellen. M.b.v. ramingen op basis van deze modellen verifiëren we de resultaten van het analytische en empirische onderzoek.

Aanpak Ramingsmodellen worden reeds meer dan 20 jaar gebruikt in Software Engineering [41]. Een uitgebreid overzicht van bestaande ramingsmodellen wordt beschreven in [42].

Allereerst maken we een selectie van de meest geschikte ramingsmodellen, op basis van volgende criterialijst:

C1 Vergelijkbaarheid in resultaten (bv. inzicht in mogelijke verbeteringen. Zo geeft [43] een techniek om de optimale onderhoudbaarheid te bepalen)

C2 Gebruik van interne programma attributen (bv. [44] waarin een zgn. maintainability index wordt beschreven)

C3 Gebruik van architecturale informatie (zoals aangeraden in [45])

Het rangschikken inzake onderhoudskost van alternatieve versies van een software systeem (op specifieke wijze geherstructureerd via refactorings) vereist vergelijkbaarheid (C1) van de schattingen.

Wanneer ramingsmodellen interne programma attributen gebruiken (C2), kunnen we de resultaten van het analytische onderzoek nauwkeurig verifiëren. Aangezien deze resultaten de inwerking van refactorings op deze attributen vastleggen, en zulke ramingsmodellen de bijdrage van elk attribuut aan de onderhoudskost kwantificeert, kunnen we nagaan of de rangschikking in geschatte onderhoudskost overeenkomt met de resultaten van het analytisch onderzoek.

Door gebruik te maken van ramingsmodellen op basis van architecturale informatie (C3) kunnen we een verificatie van de experimentele resultaten bekomen. Deze experimenten manipuleren namelijk ook de factor architecturale stijl in het onderzoek naar de inwerking van de toepassing van refactorings op onderhoudbaarheid. Indien ramingsmodellen deze informatie inrekenen, kunnen we de experimentele resultaten verifiëren.

3.4.2 Expertschattingen inzake onderhoudskost

Motivatie De manipulatie-experimenten (besproken in deel 3.3) bestuderen hoe de onderhoudskost van software systemen wordt beïnvloed door de specifieke architecturale stijl, en de daarop toegepaste refactoringstrategieën. In deze experimenten meten we zowel de uitvoeringstijd als de uitvoeringscorrectheid van perfectieve onderhoudstaken.

Deze uitvoeringstijd wordt in de praktijk bij het inplannen van de onderhoudsactiviteiten ingeschat op basis van kostenramingen. In het merendeel van de bedrijven gebruikt men hiervoor expertschattingen [46]. Dit zijn kosten- en inspannings-ramingen gemaakt door ervaren project planners op basis van intuïtie. De correctheid van deze schattingen is zeer moeilijk te benaderen met formele ramingsmodellen, en vormt daarom een uiterst betrouwbare verificatie van de experimentele resultaten inzake verschil in onderhoudbaarheid.

De argumentatie voor deze schattingen kunnen verifiëren of er externe factoren zijn die voor de resultaten verantwoordelijk kunnen zijn. Een dergelijke verificatie van de empirische resultaten kan m.a.w. aan het licht brengen of een kortere uitvoeringstijd van onderhoudstaken wel degelijk te wijten is aan de specifieke behandeling.

Doelstellingen Op basis van questionnaires worden experts inzake project planning geselecteerd. Deze experts worden verzocht schattingen te maken van de kost van de perfectieve onderhoudstaken voor de verschillende versies, gebruikt in de manipulatie-experimenten. Het vergaren van deze schattingen, alsook de argumentatie voor deze schattingen vormt een verificatie van de empirische resultaten, alsook de besluiten uit deze resultaten.

Aanpak Voor dit deel van het onderzoek is het belangrijk de contacten met de industriële sector aan te halen. Zo zijn er reeds contacten gelegd met verschillende bedrijven in Vlaanderen

met waarbij software-ontwikkeling tot de kernactiviteiten behoord (Alcatel, EDS, Kava, TryX, Nayima, etc.).

Analoog aan het empirisch onderzoek (deel 3.3) worden questionnaires opgesteld om de profielen van experts te achterhalen. Het doel hiervan is een steekproef van experts te verzamelen met sterk verschillende profielen, teneinde meerdere subjectieve beoordelingen af te vlakken naar een enkele objectieve beoordeling.

De rangschikking van de verschillende versies van de software systemen inzake onderhoudskost op basis van deze objectieve beoordeling vergelijken we met de rangschikking verkregen uit de experimentele resultaten. Door voor elke beoordeling een argumentatie op te vragen, kunnen we onze veronderstellingen over de oorzaken van specifieke experimentele resultaten verifiëren. Zo kunnen we achterhalen of er eventuele alternatieve verklaringen mogelijk zijn, die de conclusies uit de empirische resultaten kunnen verzwakken. Het ontbreken van alternatieve verklaringen versterkt het vertrouwen in de behaalde resultaten.

4 Planning

4.1 Voorbereidend onderzoek

Samenvattend realiseerde ik voor dit onderzoek reeds a) een publicatie van een paper die de nadruk legt op hiaten in het huidige onderzoek binnen het domein van refactoring [13]; b) de grondbeginselen voor en eerste resultaten van het analytisch onderzoek [22, 23]. Dit werk resulteerde uit een samenwerking met Prof. Tom Mens van de Université de Mons; en c) de nodige voorbereidingen voor het empirisch onderzoek. Hiervoor stelden we samen met hoog aangeschreven onderzoekers uit het SIMULA research laboratory in Noorwegen een experimenteel ontwerp op voor het aanpakken van deze specifieke problematiek. Binnen de eigen universiteit hebben we een samenwerking opgezet met Prof. Jan Verelst van de faculteit TEW, die gespecialiseerd is in empirisch onderzoek. Zijn co-promotorschap vormt een grote meerwaarde voor de uitvoering van dit project.

4.2 Eerste deel IWT onderzoek

Het is haalbaar het analytische en empirische onderzoek parallel uit te voeren, gezien de sterke voorbereiding.

4.2.1 Analytisch onderzoek

Aangezien de basis voor het analytisch onderzoek reeds gelegd is in het voorbereidend onderzoek [22, 23], schatten we de planning voor dit deel van het onderzoek in op 12 maanden. Deze activiteit zal volgende deeltaken omvatten:

1. Verder formaliseren van refactorings en interne programma attributen in termen van het formele programma structuurmodel.

2. Implementeren van de impact-analyse op basis van logische feiten, regels en queries zoals besproken in [23].

3. Uitvoeren van het geautomatiseerde impact-analyse proces, resulterend in de opstelling van impact tabellen analoog aan Tabel 1.

Op basis van de resultaten van dit impact-tracerings proces wordt de mijlpaal opgesteld in de vorm van papers op conferenties over software-onderhoud, waaronder als voornaamste de International Conference on Software Maintenance (ICSM) en de European Conference on Software Maintenance and Reengineering (CSMR). Gezien de huidige vraag naar validatie van refactoring in Call for Papers van Europese workshops zoals REFACE en QAOOSE zullen de resultaten ook op deze evenementen besproken worden.

4.2.2 Empirisch onderzoek

Voor het uitvoeren van de manipulatie-experimenten plannen we ruim twee jaar. Gezien de sterke voorbereiding en de opgezette samenwerkingen met experts inzake empirisch onderzoek, zowel internationaal (SIMULA research lab) als nationaal (co-promotor Prof. Jan Verelst) hebben we echter voldoende vertrouwen in de voorgestelde aanpak om de eerste interpretaties van de resultaten tegen de aanvraag voor het tweede deel van het IWT project klaar te hebben. Gebaseerd op

de ervaring uit deze eerste reeks experimenten kunnen we de aanpak bijsturen. Zo kunnen we nauwkeuriger inspelen op specifieke bevindingen. Hieronder beschrijf ik de deeltaken van deze activiteit:

1. Opstellen questionnaires, deelnemers vergaren, pre-questionnaires, pilootstudie, post-questionnaires
2. Resultaatverwerking en eventuele aanpassing experimentele set-up
3. Testen hypothesen d.m.v. manipulatie-experimenten (volgens eventuele bijsturing van pilootstudie)
4. Resultaatverwerking

Het spreekt voor zich dat in de loop van het project middelen (o.a. questionnaires) uit vorige iteraties kunnen herbruikt worden.

Op basis van de resultaten van de experimenten wordt de mijlpaal opgesteld in de vorm van papers op empirische conferenties en symposia zoals de International Symposium on Empirical Software Engineering (ISESE), de International Conference on Empirical Assessment in Software Engineering (EASE) en de International Conference on Software Engineering (ICSE). Onze ambitie ligt op het publiceren van onze resultaten in journals zoals *Empirical Software Engineering: An International Journal* en de *Transactions on Software Engineering*.

4.3 Tweede deel IWT onderzoek

Het is essentieel de nodige maatregelen te treffen om de betrouwbaarheid van dit werk zo hoog mogelijk op te voeren. Daarom plannen we op dit moment om quasi het gehele tweede deel van het onderzoek aan een verificatie van de eerder verkregen resultaten te wijden.

Het is echter onmogelijk om op voorhand alle specifieke bevindingen uit de experimenten te voorspellen. Een mogelijke aanpassing van deze planning is daarom het verlengen van de benodigde tijd voor het empirische onderzoek.

Deze verificatie kan parallel gebruik maken van twee verschillende technieken voor het verifiëren van de resultaten uit het analytisch en het empirisch onderzoek. De uitvoering van deze activiteiten is daarom sterk afhankelijk van de uitvoering van de activiteiten uit het eerste deel van het onderzoek. Daarom is het onmogelijk hiervoor reeds een gedetailleerde planning te voorzien.

5 Toepassingsmogelijkheden

Dit werk valideert het gebruik van refactorings voor het verbeteren van onderhoudbaarheid. Als wetenschappelijke test levert dit werk dan ook geen nieuwe technieken op voor het aanpakken van software problemen. Wat dit werk *wel* oplevert is gedetailleerd inzicht in het optimale gebruik van een bestaande techniek binnen het kader van onderhoudbaarheid.

De output van dit onderzoek zal niet rechtstreeks kunnen leiden tot de automatisering van het refactoringproces. Wel is dit werk absoluut noodzakelijk op het pad naar dergelijke automatisering, en zal het samen met de detectie van code-fragmenten met lage onderhoudbaarheid leiden tot een verdere ondersteuning van het onderhoudsproces.

De resultaten van dit onderzoek zullen evenwel rechtstreeks in de praktijk te brengen zijn bij het gebruik van huidige refactoringtools. Aangezien dit werk de bijdrage van refactoring voor de verbetering van onderhoudbaarheid test, vormt een logisch volgende stap de exploitatie van deze kennis gedurende het refactoringproces.

Een dergelijk refactoringproces, met kennis over de manier waarop herstructureringen de onderhoudbaarheid van broncode beïnvloeden, vormt een krachtig wapen in het verlagen van de onderhoudskost. Aangezien deze kost het overgrote deel van het software budget uitmaakt, vormt dit argument wellicht de belangrijkste reden om te investeren in dit onderzoek aan de Universiteit Antwerpen.

Referenties

[1] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Trans. Software Engineering*, 25(4):456–473, July/August 1999.

[2] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[3] Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[4] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proc. Int. Symp. Principles of Software Evolution*, pages 157–169. IEEE Computer Society Press, 2000.

[5] Jan Philipps and Bernhard Rumpe. Roots of refactoring. In Kenneth Baclavski and Haim Kilov, editors, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15, 2001*. Northeastern University, 2001.

[6] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[7] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Software Engineering*, 25(4):493–509, July/August 1999.

[8] Michelle Lee, A. Jefferson Offutt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proc. 34th Int'l Conf. Technology of Object-Oriented Languages and Systems*, August 2000.

[9] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. Int'l Conf. Software Maintenance*, pages 736–743. IEEE Computer Society Press, 2001.

[10] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In Qiaoyun Li, Bertrand Meyer, Gilda Pour, and Ruchard Riehle, editors, *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 39)*, volume 1, pages 173–182. IEEE Computer Society, 2001.

[11] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Proc. International Conference on Software Maintenance*, pages 154–162, october 2000.

[12] Michele Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, Switzerland, 2003.

[13] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Language Descriptions, Tools and Applications (LDTA)*, 2002.

[14] Lionel C. Briand, Christian Bunse, and John W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Engineering*, 27(6):513–530, 2001.

[15] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. Controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. Software Engineering*, 27(12):1134–1144, 2001.

[16] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Engineering*, 26(8):797–814, 2000.

[17] Victor R. Basili, Filippo Lanubile, and Forrest Shull. Investigating reading techniques for object-oriented framework learning. *IEEE Trans. Software Engineering*, 26(11):1101–1118, 2000.

[18] S. Demeyer, S. Ducasse, R. Nebbe, O. Nierstrasz, and T. Richner. Using restructuring transformations to reengineer object-oriented systems. Technical report, Software Composition Group, University of Berne, Switzerland,

1997. A Position Paper on the FAMOOS project.

[19] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Int. Conf. OOPSLA 2000*. ACM Press, 2000.

[20] Serge Demeyer and Stéphane Ducasse. Metrics: Do they really help? In *Proc. Languages et Modèles à Objets*, pages 69–82. Hermes Science Publications, 1999.

[21] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proc. Working Conf. Reverse Engineering (WCRE '99)*. IEEE Computer Society Press, October 1999.

[22] Bart Du Bois and Tom Mens. Describing the impact of refactoring on internal program quality. International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), ICSM-workshop, 2003.

[23] Bart Du Bois. Model-level logical reasoning about the impact of refactoring on metrics. Technical report, University of Antwerp, September 2003.

[24] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.

[25] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proceedings First International Conference ICGT 2002, Barcelona, Spain.

[26] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of UML 2003 – The Unified Modeling Language*. Springer-Verlag, 2003.

[27] Bart Du Bois and Serge Demeyer. Accommodating changing requirements with EJB. In *Proc. 9th Int'l Conf. on Object Oriented Information Systems, OOIS 2003*, September 2003.

[28] J. A. McCall, P.K. Richards, and G.F. Walters. Factors in software quality. US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977.

[29] ISO9126. Software product evaluation - quality characteristics and guidelines for their use. Technical report, ISO/IEC Standard ISO-9126, 1991.

[30] IEEE. Standards for a software quality metrics methodology. Technical report, IEEE, 1997.

[31] N. Schneidewind. Methodology for validation software metrics. *IEEE Trans. Software Engineering*, 18(5):410–422, May 1992.

[32] Victor R. Basili and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Engineering*, 22(10):751–761, October 1996.

[33] M. Genero and M. Piattini. Empirical validation of measures for class diagram structural complexity through controlled experiments. In *Proc. 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001)*, June 2001.

[34] Mark Klein and Rick Kazman. Attribute-based architectural styles. Technical Report CMU/SEI99-TR-022, Software Engineering Institute, Carnegie Mellon University, 1999.

[35] P. Clements and L. Northrop. Software architecture: an executive overview. Technical report, Software Engineering Institute, 1996.

[36] Marek Vokáč, Walter Tichy, Dag I.K. Sjoberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patterns - a replication in a real programming environment. Technical report, Simula Research Laboratory, 2003.

[37] Jim K. Lindsey. *Revealing Statistical Principles*. Arnold Publishers, 1999.

[38] Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[39] Mark Balnaves and Peter Caputi. *Introduction to Quantitative Research Methods*. SAGE Publications, 2001.

[40] M. Jorgensen, G. Kirkeboen, D. Sjoberg, B. Anda, and L. Bratthall. Human judgement in effort estimation of software projects. Beg, Borrow, or Steal Workshop, International Conference on Software Engineering, 2000.

[41] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[42] Jussi Koskinen, Henna Lahtonen, and Tero Tilus. Software maintenance cost estimation and modernization support. Technical report, Information Technology Research Institute, June 2003.

[43] Jan Bosch and PerOlof Bengtsson. Assessing optimal software architecture maintainability. In *Proc. 5th Int'l Conf. Software Maintenance and Reengineering*, pages 168–175, March 2001.

[44] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, pages 44–49, August 1994.

[45] Richard D. Stutzke. Software estimation technology: a survey. *CrossTalk - the Journal of Defence Software Engineering*, May 1996.

[46] F. J. Heemstra. Software cost estimation. *Information and Software Technology*, 34(10):627–639, 1992.