KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering

Raising the level of Abstraction in Behavioral Modeling, Programming Patterns and Transformations

Geert Delanote

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

April 2014

Raising the level of Abstraction in Behavioral Modeling, Programming Patterns and Transformations

Geert DELANOTE

Examination committee:
Prof. dr. ir. Y. Willems, chair
Prof. dr. ir. E. Steegmans, supervisor
Prof. dr. ir. W. Joosen
Prof. dr. D. De Schreye
Prof. dr. Serge Demeyer
(Universiteit Antwerpen)
Prof. dr. ing. J. Boydens
(Kulab, Oostende)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

April 2014

© 2014 KU Leuven – Faculty of Engineering
Uitgegeven in eigen beheer, Geert Delanote, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

ISBN 978-94-6018-810-7

D/2014/7515/34

Don't hesitate to travel a difficult path
if that is the direction your heart is leading.

Dankwoord

Het dankwoord, het laatste stukje tekst dat moet ingevuld worden, het ogenblik waarop je terugblikt op de afgelegde weg en je vaststelt dat dit proefschrift nooit mogelijk was geweest zonder de fantastische steun van heel veel mensen. De weg was veel langer en vooral veel moeilijker dan initieel gedacht. De weg die ik volgde leek naarmate de tijd verstreek alsmaar verder af te dwalen van mijn vooropgestelde doel, ettelijke keren heb ik geworsteld met het dilemma "ermee ophouden of toch nog doorzetten". Alhoewel ... het was mijn rationele ik die mij aan het twijfelen bracht en mij veel te vaak aan opgeven deed denken. In mijn hart heb ik nooit een halve seconde getwijfeld, het proefschrift dat u nu in handen hebt is wat ik altijd écht wou. Zonder de fantastische steun van familie, vrienden, collega's en begeleiders had ik mijn doel nooit bereikt. Dankzij hen heb ik de juiste weg gevonden en ben ik heel trots dat ik dit project succesvol heb kunnen afronden. Ik wil graag iedereen die op een of andere manier zijn of haar steentje bijgedragen heeft bedanken. Een aantal mensen wil ik toch graag extra in de schijnwerper plaatsen.

Ik zou in de eerste plaats mijn promotor, professor Eric Steegmans, van harte willen bedanken voor zijn begeleiding en onze inspirerende gesprekken die dit doctoraat mee vorm gegeven hebben. Eigenlijk moet ik hem nog veel meer bedanken voor zijn geloof in mijn kunnen en zijn eindeloze geduld en begrip tijdens de talloze keren dat ik fysiek of mentaal in de lappenmand lag. Eric was ook de begeleider van mijn master-thesis die ik, na een woelige en moeilijke studieperiode, pas na veel vallen en opstaan afgerond kreeg. Groot was mijn verrassing toen ik na de presentatie van mijn master-thesis de vraag en kans kreeg om als onderzoeker te starten in zijn onderzoeksgroep, Softwareontwikkelingsmethodologie. Omdat ik mijn studies aangevat heb aan de KULAK, heb ik nooit mogen ervaren hoe het was om les te krijgen van Eric, winnaar van ontelbare gouden krijtjes, de onderscheiding voor beste prof. Als medewerker kreeg ik de kans om met hem samen te werken als assistent bij verschillende van zijn vakken. Ik heb hem dan ook leren kennen als een ontzettend

gepassioneerd docent. Door hem heb ik ontdekt dat doceren, mensen coachen en begeleiden iets is wat ik ook heel erg doe. Ik heb de vakken object-gerichte analyse en object-gericht programmeren altijd met heel veel enthousiasme ondersteund. Mocht ik de kans krijgen om te doceren, hoop ik dat ik ooit half zo goed word als hij is.

De overige leden van de begeleidingscommissie, professor Wouter Joosen en professor Danny De Schreye, alsook de overige juryleden professor Serge Demeyer en professor Jeroen Boydens, ben ik dankbaar om mijn werk aan hun kritische blik te onderwerpen. Hun opmerkingen hebben ongetwijfeld de kwaliteit van de tekst verbeterd.

Ik heb het voorrecht gehad om met heel fijne collega's in de onderzoeksgroep Softwareontwikkelingsmethodologie te mogen samenwerken.

Pieter Bekaert begeleidde mij eerst bij mijn master-thesis en was nadien degene die mij wegwijs maakte op het departement Computerwetenschappen. Zijn enthousiasme voor object-gerichte analyse en programmeren, zowel als assistent als als onderzoeker, heeft mij zeker positief beïnvloed. Hij was ook de drijvende kracht achter de eerste publicatie waar mijn naam als co-auteur op vermeld staat.

Van Frank Devos heb ik geleerd: "*Doctoreren es vele schriv'n*". Ik ben heimelijk nog altijd een beetje jaloers op zijn talent om dingen pragmatisch en krachtdadig aan te pakken. Ik hoor hem nog altijd zeggen (ondertussen ruim 10 jaar geleden): "Als ik vanaf nu elke dag 1 blad schrijf, dan is mijn thesis binnen 200 dagen klaar". 200 dagen later stonden we op de receptie te toosten ter ere van zijn promotie. Frank loodste mij ook binnen bij zijn toenmalige zaalvoetbalploeg waar ik daarna 13 jaar lang heel ontspannende en fijne momenten beleefd heb.

Jeroen Boydens kwam Frank aflossen als collega met West-Vlaamse roots. Nog altijd begrijp ik niet waar hij al zijn energie vandaan haalde (en haalt): hij combineerde namelijk zijn onderzoekswerk met lesgeven aan de KHBO en een gezin. Bovendien mocht je hem altijd advies of hulp vragen, ik denk niet dat hij eenmaal "nee" gezegd heeft. Elke donderdag trotseerde hij de files of vertragingen van de NMBS om naar Leuven te komen om te vergaderen en tussendoor, nadat we (soms vruchteloos) Sven De Labey wakker gemaakt hadden, een ViaVia-Spaghetti met de collega's te nuttigen.

Sven De Labey heb ik eerst mogen begeleiden bij zijn master-thesis. Al heel snel was ik overtuigd van zijn kwaliteiten en vertelde ik Eric dat Sven een aanwinst voor de onderzoeksgroep zou zijn. Aangezien Sven voor zichzelf ook al beslist had dat hij bij de onderzoeksgroep wou komen, was dat een heel

kort gesprek en werd hij een jaar later collega. In die periode zonk ik mentaal steeds verder weg in het moeras, en al heel snel werden de rollen van mentor en “ondersteunde” omgewisseld. Meer dan eens hebben tot ’s nachts heel laat via instant messaging eerst gepraat over waarom ik steeds moeizamer vooruit geraakte met mijn onderzoek waarna hij mij op zijn manier hielp. Sven heeft namelijk een uitzonderlijk talent om teksten overtuigend en aangenaam om lezen te schrijven. Zonder zijn hulp was de Pluto-publicatie nooit tot stand gekomen. Ook in de laatste fase van dit doctoraat heeft hij een heel belangrijke rol gespeeld.

Zonder te willen beweren dat dit hun enige verdienste was, wil ik ook nog Jan Dockx, Jamal Said, Stefan Van Baelen, Bart Swennen, Nele Smeets, Marko van Dooren en Koen Vanderkimpen bedanken voor de aangename werksfeer. Na een onderbreking van een aantal jaar, kreeg ik bij Sam Michiels een bureau om mijn doctoraat af te werken: ik wil hem zeker ook bedanken voor het aangename gezelschap.

In september 2000 ben ik begonnen als onderzoeker, nu bijna 14 jaar later kan ik het werk eindelijk succesvol afronden. De weg was (te) lang en vol onvoorziene hindernissen. Mijn weg liep blijkbaar via een diep en donker moeras waar ik zonder het te beseffen steeds dieper in vast geraakt ben. Initieel realiseerde ik mij enkel mijn steeds hardnekkigere fysieke ongemakken. Gelukkig kwam ik vrij snel bij Ronny Massa terecht die mij ontelbare keren via shiatsumassages zo goed en zo kwaad als hij kon oplapte. Naast het verlichten van de fysieke klachten, ondersteunde hij me ook mentaal, maar ik besef nu dat ik zijn inzichten toen niet altijd even goed begrepen heb. Eind 2007 liep mijn contract aan de KULeuven af. Tot september 2011 bleef mijn onderzoekswerk onaangeroerd. Ongewild. Ik had er gewoon de mentale kracht niet voor. Ik was helemaal geblokkeerd. En ik wilde het toch zó graag. Ik ging elke dag een beetje meer kapot. Mentaal. En fysiek. In september 2011 ben ik bij Synergeia binnengestapt en sindsdien heb ik mij met hun hulp stapje voor stapje kunnen bevrijden uit dat donkere gat waar ik in terecht gekomen was. Bedankt Pieter Lanoye om mij geleidelijk aan van mijn fysieke klachten af te helpen. Ik heb er vertrouwen in dat je mij in de toekomst terug fit krijgt. Ik ben er zeker van dat de yoga-sessies van Ronny Massa en fascia-behandelingen van Johan Van Welden ook een cruciale rol gespeeld hebben. Jos Van Pelt en Patricia Wichman, beiden ook verbonden aan Synergeia, startten tegelijk met het mentale oplapwerk. Jos, de sessie van 12 december vergeet ik nooit meer. Je dacht out-of-the-box, ik wist gevoelsmatig onmiddellijk dat je er boenk opzat met je visie. Na voorbereidende ondersteuning van Ronny Massa, kon ik met de

hulp van Leo Vulsteke in het voorjaar van 2012 een heel belangrijk keerpunt realiseren. Bedankt Leo. De volgende fase was om mijn doctoraatswerk terug op te pakken. Jona den Aantrekker was en is mijn kompas tijdens het laatste stuk van de weg. Jona, het belang van jouw bijdrage kan niet genoeg benadrukt worden. Jij helpt me keer op keer om de dingen beter in perspectief te zetten. Samen met Kaat Timmerman, heb jij er ook voor gezorgd dat ik mezelf eindelijk beter begrijp.

En als laatste maar niet minder belangrijk wil ik mijn vrienden en familie bedanken. Veel vrienden hebben mij de afgelopen jaren op verschillende manieren fijne en aangename momenten bezorgd. In het bijzonder wil ik heel graag Norman & Chantal, Tom & Mie en Michael & Griet bedanken.

Mijn schoonfamilie, Marleen, Catharina, Stijn en Marc, wil ik bedanken voor alle steun tijdens de afgelopen jaren. De blijk van interesse door me regelmatig te vragen hoe het nu was met mijn doctoraat was hartverwarmend.

Bert Hellinger schreef[1]: "*Er gebeurt iets prachtigs wanneer mensen naar hun ouders kijken en de bron van het leven herkennen. De liefde eist dat de ontvanger zowel het geschenk als de gever ervan respecteert. Wie het leven liefheeft en respecteert, heeft ook de gevers van het leven lief.*" Gilbert en Magda, ik ben blij dat jullie erbij zijn wanneer ik dit werk presenteer.

De vier mooiste dagen van mijn leven waren de vier geboortedagen van mijn lieve kinderen. Alexander, Lucas, Arend en Soetkin, jullie zijn mijn zon, maan en sterren tegelijk. Altijd kunnen jullie een glimlach op mijn gezicht toveren.

Natasja, je bent al 20 jaar mijn steun en toeverlaat. Niemand gelooft meer in mijn kunnen dan jij. Al was het vaak niet eenvoudig om de juiste stimulans te vinden om mij weer op pad te helpen, jij bleef onvermoeibaar zoeken. Afgelopen jaren was ik te vaak afwezig, moest jij de kar te vaak alleen trekken. Nu ik dit werk afgerond heb, beloof ik dat ik ook terug het voortouw zal nemen, zodat jij zorgeloos kunt meedrijven op mijn golf. Eindelijk kunnen we terug naar de toekomst kijken en onze dromen verder verwezenlijken. *Wodka!*

Tenslotte zou ik dit proefschrift ook willen opdragen aan mijn nonkel Christian. Je hebt ons al even verlaten, maar toch voel ik nog elke dag je aanwezigheid.

Veel leesplezier,
Geert Delanote
Leuven, April 2014

[1] De verborgen dynamiek van familiebanden – Bert Hellinger

When I was 17, I read a quote that went something like: "If you live each day as if it was your last, someday you'll most certainly be right." It made an impression on me, and since then, for the past 33 years, I have looked in the mirror every morning and asked myself: "If today were the last day of my life, would I want to do what I am about to do today?" And whenever the answer has been "No" for too many days in a row, I know I need to change something.

—Steve Jobs, Stanford Commencement Speech 2005

Aan mijn allerliefste Natasja,
Alexander, Lucas, Arend en Soetkin.

Abstract

Since the very beginning of software development there was an unstoppable demand for higher productivity, better quality and more complex software systems. If the problem to be solved by the software system has a high complexity, solving it will inevitably also be complex. This inherent complexity is often referred to as *essential complexity.* The way software is developed however also causes some complexity. Better software development processes and better software building techniques, for example (programming) languages, reduce this complexity. Lower expressiveness and less abstraction introduce unnecessary and avoidable challenges that is often referred to as *accidental complexity.*

The goal of this work is to contribute to the reduction of accidental complexity of building software systems. Improvements on three different places in the development process are proposed:

- *Programming Patterns*: properties and associations are typically accompanied with requirements restricting the values properties or objects an object can be associated with. Three different types of requirements are identified to facilitate the (re-)definition of requirements at different positions in the class hierarchy. *Value Requirements* restrict the values independent of the state of the object. *State Requirements* restrict allowed combinations of values of different properties and associations. *Transition Requirements* restrict transitions to new values in view of current values. A *separation of concerns* between the development of methods describing requirements and the methods describing the state changes is reached by encapsulating the description of the requirements in their own *inspectors.* A family of patterns describes how all methods describing the state and behavior of a property or association collaborate. Finally, a first step towards a language extension to avoid the technical code of the patterns is presented.

- *Behavioral Modeling*: conceptual models introduce accidental complexity when they contain technical aspects in order to describe real-world facts. Such complexity is introduced by enforcing ("locking in") decisions that should have been made in a later activity in the software development process. UML and OCL lack expressive constructs to reason about *event occurrences*, even more so when the *historical aspect* of such occurrences becomes important. This work presents a new operator, the #-operator, that allows analysts to treat events as first-class citizens. By assigning a property, that represents the execution time, to events, it becomes possible to model historical event information without the need to introduce irrelevant facts in the conceptual model. A conceptual model never describes the whole universe, but is always a description of a subset of real-world facts. The decision to model a given fact as an object or as an event depends on the selected subset of real-world facts. A *guiding principle* assists the analyst in his decision-making: if the lifetime of a fact is of importance then the fact should be modeled as an object. Otherwise, if the fact is instantaneous, the fact should be modeled as an event.

- *Transformations*: generally, multiple languages are used during the development of a software system. Each language is formally defined by a metamodel. These metamodels serve as the basis to define transformations between the different models. Different metamodels mostly have common structural constructs and associated functionality: a framework offering constructs to build *hierarchical composition structures* is developed to avoid the need repeat this work over and over again. A distinction is made between *parent/child* relations to specify interdependencies and *dependee/dependant* relations to specify unidirectional dependencies. Next to these constructs, the framework offers a metamodel-independent transformation approach. The knowledge of how to transform concrete metamodel elements is decoupled from the *managing* algorithm. Developers of a transformation provide *strategies* to transform concrete model elements, while the framework is responsible for tasks as execution order, managing cross-model consistency, model validity,. . .

Beknopte samenvatting

Sinds het prille begin van software ontwikkeling, was er een niet te stoppen vraag naar hogere productiviteit, betere kwaliteit en complexere software systemen. Als het probleem, dat door het software systeem opgelost moet worden, een hoge complexiteit heeft, dan zal het oplossen onvermijdelijk ook complex zijn. Deze inherente complexiteit wordt vaak aangeduid als *essentiële complexiteit*. De manier waarop software ontwikkeld wordt, veroorzaakt echter ook complexiteit. Betere software ontwikkelingsprocessen en betere technieken om software te bouwen, bijvoorbeeld programmeertalen, verminderen deze complexiteit. Lagere expressiviteit en minder abstractie introduceren onnodige en vermijdbare uitdagingen wat vaak aangeduid wordt als *accidentiële complexiteit*.

Het doel van dit werk is bij te dragen tot een reductie van accidentiële complexiteit bij het bouwen van software systemen. Verbeteringen op drie verschillende plaatsen in het ontwikkelingsproces worden voorgesteld:

- *Programmeerpatronen*: Kenmerken en relaties worden typisch vergezeld door vereisten die de waarden die kenmerken kunnen krijgen of de mogelijke objecten, waarmee een object geassocieerd kan worden, beperken. Drie verschillende soorten vereisten worden geïdentificeerd om de (her)definitie van de vereisten mogelijk te maken op verschillende plaatsen in de klasse hiërarchie. *Waarde Vereisten* beperken de waarden onafhankelijk van de toestand van het object. *Toestand Vereisten* beperken toegelaten combinaties van waarden van verschillende kenmerken en relaties. *Overgang Vereisten* beperken de overgangen naar nieuwe waarden ten opzichte van de huidige waarden. Een *scheiding van belangen* tussen de ontwikkeling van methodes, die de vereisten beschrijven, en de methodes, die de toestand veranderen, wordt bereikt door de beschrijving van de vereisten in hun eigen *inspectoren* te kapselen. Een familie van patronen beschrijft hoe alle methodes, die de toestand en het gedrag van

een kenmerk of een relatie beschrijven, samenwerken. Tenslotte wordt een eerste aanzet van een taaluitbreiding, om de technische code van de patronen te vermijden, voorgesteld.

- *Modelleren van gedrag*: conceptuele modellen introduceren accidentiële complexiteit als ze technische aspecten bevatten om feiten van het probleemdomein te beschrijven. Deze complexiteit wordt geïntroduceerd door beslissingen door te drukken die in een latere activiteit van het software ontwikkelingsproces zouden genomen moeten worden. UML en OCL missen expressieve constructies om te redeneren over *voorkomens van gebeurtenissen*, zelfs nog meer wanneer het *historische aspect* van zon voorkomens belangrijk wordt. Dit werk stelt een nieuwe operator voor, de #-operator, die analisten toelaat om gebeurtenissen te behandelen als volwaardige constructies. Door een eigenschap, die het tijdstip van voorkomen voorstelt, toe te kennen aan gebeurtenissen, wordt het mogelijk om informatie in verband met de geschiedenis van gebeurtenissen te modelleren zonder irrelevante feiten in het conceptuele model te moeten voeren. Een conceptueel model beschrijft nooit het volledige universum, maar is altijd een beschrijving van een deelverzameling van het universum. De beslissing om een gegeven feit te modelleren als een object of een gebeurtenis hangt af van de geselecteerde deelverzameling van het universum. Een *leidraad* helpt de analist in zijn besluitvorming: als de levensduur van een feit belangrijk is, dan moet dat feit als een object gemodelleerd worden. Als het feit echter geen levensduur heeft, dan moet dat feit als een gebeurtenis gemodelleerd worden.

- *Transformaties*: over het algemeen worden meerderen talen gebruikt tijdens de ontwikkeling van een software systeem. Elke taal wordt formeel gedefinieerd door een metamodel. Deze metamodellen dienen als basis om transformaties tussen verschillende modellen te definiëren. Verschillende metamodellen delen meestal vaak voorkomende structurele constructies met bijhorende functionaliteit: er wordt een raamwerk ontwikkeld, dat constructies aanbiedt om *hiërarchische compositie structuren* te bouwen en zo repetitief werk te vermijden. Een onderscheid wordt gemaakt tussen *ouder/kind* relaties om wederzijdse afhankelijkheid te specificeren en tussen *eigenaar/afhankelijke* relaties om unidirectionele afhankelijkheden te specificeren. Bovenop deze constructies biedt het raamwerk een metamodel-onafhankelijke aanpak voor transformaties aan. De kennis om concrete metamodel elementen te trans-

formeren wordt los gekoppeld van het *sturend* algoritme. Ontwikkelaars van een transformatie voorzien strategieën om concrete model elementen te transformeren, terwijl het raamwerk verantwoordelijk is voor taken zoals uitvoeringsvolgorde, beheren van de consistentie tussen modellen, validatie van het model, ...

Contents

1 Introduction **1**
- 1.1 Background . . . 2
  - 1.1.1 Software Engineering . . . 2
  - 1.1.2 Software Development Process . . . 4
  - 1.1.3 Software Development Techniques . . . 7
  - 1.1.4 No Silver Bullet . . . 10
- 1.2 Contributions . . . 11
- 1.3 Overview of the Text . . . 13

2 A Pattern-based Approach towards Expressive Specifications of Properties and Associations **15**
- 2.1 Preamble . . . 16
- 2.2 Introduction . . . 17
- 2.3 Motivation . . . 19
  - 2.3.1 Software Quality . . . 19
  - 2.3.2 Patterns . . . 20
  - 2.3.3 Language Support . . . 21
- 2.4 Principles and Notation . . . 22
  - 2.4.1 Design By Contract principle . . . 23
  - 2.4.2 Command-Query Separation principle . . . 24
  - 2.4.3 Liskov Substition Principle . . . 24
  - 2.4.4 Open-Closed Principle . . . 25
  - 2.4.5 No-Choice Principle . . . 26
  - 2.4.6 Notation . . . 26
- 2.5 Requirements . . . 27
  - 2.5.1 Value Requirements . . . 27
  - 2.5.2 State Requirements . . . 28
  - 2.5.3 Transition Requirements . . . 28

2.6 Properties 28
2.6.1 Representation 29
2.6.2 Value Requirements 30
2.6.3 State Requirements 32
2.6.4 Invariant 33
2.6.5 Transition Requirements 34
2.6.6 Construction 34
2.6.7 Inheritance 38
2.6.8 Language Construct 42
2.7 Associations 45
2.7.1 Unidirectional Associations 46
2.7.1.1 Representation 46
2.7.1.2 Value Requirements 47
2.7.1.3 State Requirements 47
2.7.1.4 Invariant 50
2.7.1.5 Transition Requirements 50
2.7.2 Bidirectional Associations 50
2.7.2.1 Value Requirements 51
2.7.2.2 State Requirements 52
2.7.2.3 Invariant 53
2.7.2.4 Transition Requirements 55
2.7.2.5 Language Construct 62
2.8 Another Approach? 63
2.9 Conclusion 66

3 Concepts for Abstracting away Object Reification at the level of Platform Independent Models (PIMs) 69
3.1 Preamble 70
3.2 Introduction 71
3.3 Principles for Conceptual Modeling 73
3.3.1 Principle of Uniqueness 73
3.3.2 Principle of No Redundancy 73
3.3.3 Principle of Completeness 74
3.3.4 Principle of Preciseness 75
3.3.5 Principle of Minimalism 75
3.3.6 Principle of No History 76
3.4 Platform Independent Models 76
3.4.1 Running Example 76
3.4.2 PIM Using Properties 77

3.4.3 PIM Using Reification 79
3.4.3.1 PIM With UML2.0 79
3.4.3.2 Method Class 80
3.4.4 Conclusion 82
3.5 The Base PIM 84
3.5.1 Semantics Of The # Operator 84
3.5.2 The Example Revisited 84
3.6 Transformation from PIM to PSM 86
3.6.1 PIM Using Properties 86
3.6.2 PIM Using Reification 90
3.7 Objects as Arguments 91
3.7.1 Life cycle of an object 92
3.7.2 Evolving state of an object 94
3.8 Object or event? 96
3.9 Conclusion 98

4 A Framework for Executing Cross-Model Transformations Based on Pluggable Metamodels 101

4.1 Preamble 102
4.2 Introduction 103
4.3 Motivation 104
4.4 Design Goals 105
4.5 The Pluto Framework – Overview 107
4.6 Designing Concrete Metamodels as Pluto Extensions 109
4.6.1 Reusable Composition Concepts 109
4.6.2 Reusable Dependency Management 111
4.7 Model Transformations 113
4.7.1 Decorating Model Elements with Transformation Strategies 114
4.7.2 Pluto's Generic Algorithm for Transforming Model Elements 115
4.7.3 Illustration 117
4.8 Related Work 119
4.9 Conclusion 120

5 Conclusion 121

5.1 Summary and Contributions 121
5.2 Directions for Future Work 124

Appendix A Property Pattern **129**

Appendix B Unidirectional Association Pattern **133**

Appendix C Bidirectional Association Pattern **139**

List of Figures

2.1 A bidirectional association with existential dependency 45
2.2 The two generic setters for associations with restricted multiplicity 56
2.3 Another approach to develop complex mutators. 65

3.1 PIM using Properties. 77
3.2 PIM using Reification. 80
3.3 The Method Class. 81
3.4 PIM using Method Class. 81
3.5 The Base PIM using the #-Operator. 85
3.6 Specification of the Different Transformation Steps. 88
3.6 Specification of the Different Transformation Steps. (Ctd) . . . 89
3.7 The PIM using Properties. 91
3.8 A query for retrieving all readers of a book. 93
3.9 EROOS Class: population and archive. 93
3.10 The partition notation for a query retrieving all readers. 94
3.11 The @-operator. 95
3.12 Definition of the @-operator. 96
3.13 The *same* fact "transfer" modeled given two different sets of real-world facts. 98

4.1 Overview of the Pluto framework for metamodeling 107
4.2 Parent/child and dependee/dependant relations in Pluto 110
4.3 Transformation Overview 113
4.4 Reusable Transformation Concepts 114
4.5 Strategies interact with the Pluto transformation algorithm . . 116
4.6 Illustration of Pluto's transformation algorithm 118

5.1 Shorthand to reduce the set of occurrences. 125

List of Listings

2.1 Representation of the property balance 30
2.2 Value Requirement of the property credit limit 31
2.3 State Requirement between balance and credit limit 33
2.4 Invariant from the property balance 35
2.5 Transition requirement of the property balance 36
2.6 Advanced mutator withdraw 36
2.7 Basic setter for the property balance 37
2.8 Construction of a bank account 38
2.9 ClassObject inner class for the class BankAccount 39
2.10 Redefinition of the Value Requirement of the property balance 40
2.11 'Dynamic binding' of a 'class method' 41
2.12 A State Requirement involving the balance and the upper limit 41
2.13 The class of bank accounts 42
2.14 The class of junior bank accounts 43
2.15 Value Requirement for a unidirectional association. 48
2.16 State Requirement for a unidirectional association. 49
2.17 Value Requirement for the association between savings accounts and current accounts 52
2.18 State Requirement for the association between savings accounts and current accounts 54
2.19 Invariant for the association between savings accounts and current accounts in class `CurrentAccount` 55
2.20 Setter in the controlling class `SavingsAccount` (see figure 2.2b) 57
2.21 Transition Requirement in the controlling class `SavingsAccount` (see figure 2.2b) 58
2.22 Basic setter for a bidirectional association in the non-controlling class 59
2.23 Transition Requirement and Setter in the class `CurrentAccount` (see figure 2.2b) 60

2.24 Transition Requirement in the controlling class `SavingsAccount` (see figure 2.2a) 61
2.25 The association between current and savings accounts (1/2) . . 62
2.26 The association between current and savings accounts (2/2) . . 63
2.27 Transition Requirement without invariants 64
A.1 The pattern for a properties α and δ 129
B.1 The pattern for a unidirectional association Foo referring Bar: class Foo 133
B.2 The pattern for a unidirectional association Foo referring Bar: class Bar 137
C.1 The pattern for an association Foo_Bar 139
C.2 The pattern for an association Foo_Bar 145

There is only one thing that makes a dream
impossible to achieve: the fear of failure.
– Paulo Coelho

Chapter 1

Introduction

The March 1949 issue of Popular Mechanics [16] wrote: "*Where a calculator like ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1000 vacuum tubes and perhaps weigh only 1½ tons.*" This quote is only sixty five years old and originates from a time where the very first computers were constructed. The quote must remind us of what a spectacular evolution computer science has passed through. Today computers are everywhere. Nearly everything connected to some sort of energy source (electricity, batteries, gasoline,...) contains a microchip and software: airplanes, cars, televisions, cell phones or toys. In his talk "The Promise, The Limits, The Beauty of Software", Grady Booch said that the 2010s will be the age of the transparency (software will become more and more invisible) and the 2020s will be the age of total dependence (living and working without touching software will become almost impossible) [14].

The automotive industry is an excellent example to illustrate this evolution [61, 65]. In 1977, the first electronic control unit (ECU), containing only a few kilobytes of software, was used for electronic spark timing in the General Motors Oldsmobile Toronado. The first applications were designed directly in machine code or C. In 1981, GM produced cars with already about 50.000 lines of code. Current cars have ECUs about anywhere containing tens of millions of lines of software code controlling everything from brakes to the volume of your radio. The Mercedes-Benz S-class radio and navigation system requires over 20 million lines of code alone. With software, functionality deemed impossible before can now be added to the car. A car can now monitor its own state and is aware of the world around it making it possible to give traffic information or to provide parking assistance. In 1983, GM engineer Jonas Bereisa predicted

that "*software development will become the single most important consideration in new product development engineering*" [37]. Broy et al. [61] estimate that 80% of innovations in a car come from computer systems. Electronics and software take 35 to 40 percent of the cost of a car, with software development contributing about 13 to 15 percent of that cost [65].

The automotive example perfectly proves the big challenges software developers faced and still face. Namely, since the very beginning of software development there was an unstoppable demand for (i) higher productivity, (ii) better quality and (iii) larger and more complex software systems. To be able to respond to that demand, the way software systems are built had to evolve rapidly and adequately. Both processes and techniques used to develop software systems needed to evolve to enable building larger and more complex systems. *Software engineering* is the research field that studies these processes and techniques.

In an interview with the CHIPS magazine [38], Grady Booch summarized how software engineering responded: *"The history of software engineering has been one of growing levels of abstraction – we see this in our languages (assembly to FORTRAN to Ada to Java), our methods (structured analysis and design to object-oriented design), and our platforms (programming running on raw iron to basic operating systems to platforms such as .NET and J2EE). This growth has occurred simply as a meaningful engineering response to the growth in complexity in the kinds of software systems we as an industry are asked to create. Ours is a ruthless industry: it has its fads, but ultimately, those things that do not add value are quickly discarded. Building quality software-intensive systems that matter is fundamentally hard, and just as Fred Brooks observes, I don't see any development in the near future that will change that. Software development is fundamentally hard, and the way we counter that complexity is by abstraction."*

1.1 Background

1.1.1 Software Engineering

In the early years of computer science, software didn't get much attention. Hardware problems claimed nearly all attention: it was not yet possible to build a reliable, functioning computer. Moreover, hardware was also by far the most expensive part of a computer. During the 1960s, a huge turnaround took place: hardware became more reliable, powerful and also a lot cheaper. In 1965, Moore wrote in his article *Cramming more components onto integrated*

circuits [107]: “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.” Ten years later, he adjusted this observation to a doubling every two year. Today, this prediction is known as Moore’s law, which states simplified that the processing power of computers will double every two years.

The increased processing power made it possible to run more complex software. Together with the rapidly decreasing cost for hardware and increasing cost for building software, this caused the attention shift from hardware to software. Developing more complex software appeared to be far from easy and the same problems kept resurfacing in software projects. These problems were threefold: (1) software projects exceeded estimated budgets, (2) software projects ran over time, (3) software lacked the required quality (inefficient, missing features, bugs). In *The Humble Programmer* [79], Edsger Dijkstra called this *the software crisis*: “The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he had to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis?”

In 1968 a NATO conference [108] was organized to discuss these problems and adopted the term *software engineering* as “*its (deliberately provocative) title*” [115]. Until then developing software was merely a craft. Adopting engineering principles, proven to be successful in other disciplines, was the proposed solution to the software crisis. Developing software had to adopt a more methodical and industrial approach. In *Computer: A History of the Information Machine* [63] Campbell-Kelly and Aspray described it as follows: “The Garmish conference began a major cultural shift in the perception of programming. Software writing started to make the transition from being a craft for a long-haired programming priesthood to becoming a real engineering discipline. It was the transformation from an art to a science.”

Software engineering consists of two major components. On the one hand, software development methodologies or processes define the way software must be built. Professionalizing software engineering implies developing better methodologies to increase productivity. On the other hand, software building techniques used to construct the software must be improved. Better techniques imply amongst others increased productivity and higher quality. The next sections explain both parts in more detail.

1.1.2 Software Development Process

The software development process or methodology is structured into a number of activities to cope with the complexity of producing and maintaining software. IEEE defines *software development* [132] as follows : "The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use. Note: These activities may overlap or be performed iteratively." The most common activities are analysis, architectural design, design, implementation, deployment, verification & validation and maintenance.

- **Analysis.** The goal of this activity is twofold: (1) the requirements phase is the process of identifying, specifying and validating the requirements of the system. These requirements comprise both the functional and nonfunctional[1] requirements that must be solved by the software system. (2) The domain analysis phase is the process of studying the real world, also called the problem domain.

  To be able to describe the (functional and nonfunctional) requirements precisely, the real world must first be understood very well because the requirements of the software system are based upon the facts and events that can occur in the real world. We agree with the claim of Devos saying that functional requirements cannot be precisely described without detailed knowledge of the real world [75]. The study of the real world results in a *conceptual model.* The conceptual model is a complete and formal description of the problem domain containing facts (e.g. bank accounts), behavior (e.g. depositing money on a bank account) and business rules (e.g. the balance of a bank account may not drop below the credit limit). It's important to stress that the conceptual model must be unambiguous, because when developers at later phases in the development process need to make assumptions about the model, they will most likely end up with false assumptions leading to problems with the software system.

  This activity is a particularly critical stage as errors at this stage inevitably lead to problems. The later (in the software development pro-

[1] Although nonfunctional requirements (security, performance,...) are an important part of a software system, these requirements are outside the scope of the research and not handled in the remainder of this text.

cess) the mistake is detected, the higher the cost to repair it. It can be compared to knitting a sweater: the sooner you detect a missed stitch, the easier to repair it.

- **Architectural design.** The architectural design phase identifies the overall structure, or *architecture*, of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed [126]. The result of this phase is an *architectural model*. The architectural model is used as an aid in discussions with the system stakeholders. This phase investigates if the system can meet the non-functional requirements, for example redundant components and fault-tolerance mechanisms can be added to improve availability or safety-critical features can be identified and put in a dedicated subsystem.

- **Design.** The components identified in the architectural model are further elaborated. The requirements identified during the analysis phase are *transformed* into software elements. Software quality factors and non-functional requirements steer the decisions made during this phase. The result of this phase is a *design model*. The design model must contain all details about the software system to be built. This phase is a very intensive phase that requires lots of decisions. Decisions the designer has to make are for example which software libraries or components can be (re)used, which software patterns are suitable, what are the needed substructures, what elements belong to the interface, which classes are needed, . . . To facilitate this decision-making process the design model is built in steps by gradually adding details to it. In a first step, a high-level design model is elaborated followed by one or more lower level design models resulting in a design model that serves as basis for the next phase, the implementation phase.

- **Implementation.** In this phase, the software system described by the design model is *transformed* into *executable code*. Although one may think the opposite, developing the executable code with a programming language is often not only labor intensive but also has its own challenging problems. Executable code must be precise, formal and detailed enough to enable a machine to understand it. More details must be added to the design model to make this possible. The executable code is the ultimate *model* expressing how the requirements are transformed into a software system. Therefore, it is important that code is written in a

clean way because a software engineer other than the author who wants to understand every detail of the software system will at some point have to look at the code too.

- **Deployment.** During deployment, the software is brought into use. The software system is *installed* in a production environment. Some customization may be performed by setting some parameters to let the software system behave the way the customer wants.
- **Verification and Validation.** While analysis, architectural design, design, implementation and deployment are activities that can also be identified as a phase during the development process, this activity can not. The activity of verifying and validating must be performed at the level of each before mentioned phase or activity. The goal of this activity is to guard that the different models meet the proposed criteria. *Executing* conceptual models can aid in detecting errors or deficiencies at the analysis level [48]. Testing a piece of (executable) code verifies if the code is conform to specification [128].
- **Maintenance.** Maintenance is a twofold activity: on the one hand errors detected after deployment must be corrected, and on the other hand the requirements might change in time and the software system needs to be adjusted. This activity is somewhat special as its performance mostly triggers the execution of all previous described activities. The most challenging task of this activity is to keep all models synchronized and consistent.

The *process* of software development is the way in which we produce software through combining the different activities in a some way. A good process is as important as good methodology for each individual activity. Different processes have been used over the years.

Waterfall model. Royce presented the first software development process model [117]. The process identifies all above activities and presents them as a cascade where each activity *flows* into the next like a *waterfall*. Each activity is actually also a phase and each phase results in models that must be *committed* before the next phase is allowed to start. More precisely, first the analysis phase is completely performed and approved before the design phase starts. Analogously, the design phase must be completed and approved before implementation starts. Testing starts after implementation and when done the maintenance phase starts. In practice, this strict separation is relaxed and phases can overlap to allow feedback between consecutive phases.

Design deficiencies discovered during the implementation phase may trigger adjustments of the design model. Although not strictly *sequential*, at some point each phase is *closed*, often leading to undesired behavior of the software system. This process should only be used when the requirements are well known in advance and expected to be stable.

Incremental Development. Instead of following a linear path, most software development processes follow an iterative, incremental approach. Incremental development is based on the idea of developing an initial version of the software system, followed by several intermediate versions until the final version of the system is developed [126]. In an incremental development process, the different activities are interleaved with rapid feedback across the activities. Developing different versions of the software systems facilitates customer feedback. Generally, it is easier for customers to give meaningful comments on a working piece of software than on the different models. When a given version does not meet the expectations of the customer only the last increment has to be revised. Moreover, intermediate versions can often already be used by the customer. An incremental approach also adapts easier to changing requirements. On the other hand, an incremental development process is more difficult to manage because the overall progress is less clear. And more important, the overall structure of the system can become poor making it harder and harder to incorporate next increments.

Incremental development is now the most common approach and is very well suited for small and medium-sized software systems [126]. Large and complex systems require a good overall structure or architecture and therefore a combination of both approaches is advised. The architecture is developed before the different increments. If the requirements are known and understood from the beginning these increments can also be defined in advance, the approach is then said to be *plan-driven*. In the other case, a *value-driven* or *agile* approach [80], also known as Extreme Programming [46], namely the individual increments are not defined in advance, can be used.

1.1.3 Software Development Techniques

(Programming) languages. To enable higher productivity and better quality, the techniques used to build software systems must have a high *quality* too. *Raising the level of abstraction* of (programming) languages has been the core strategy to achieve that goal. The first generation programming language, also known as machine language, was very low level and platform specific. Because developing programs was very difficult and error prone, quickly a second gen-

eration programming language arose, *the assembly language* [90]. Although still platform specific, they were easier to use and understand. Third generation programming languages, also know as high-level programming languages, are machine independent. The constructs of these programming languages are closer to the human way of thinking and need to be *compiled* into a number of machine-level instructions. Fourth generation programming languages aim to be even closer to human language and are designed for a specific purpose, for example MATLAB [27] is a fourth generation language. Finally, there are also fifth generation programming languages. These languages try to step away from actual *programming* an algorithm to solve a problem and try to solve the problem by describing constraints. Fifth generation languages are for example used in the domain of artificial intelligence.

Third generation programming languages like Java also raised their level of abstraction in their evolution from the first version to their current version through introducing new constructs. Ways the Java programming language itself is improved are among others the introduction of new elements like a new keyword (for example `assert` in version 1.4 [3]), new language features (for example generics in version 5.0 [7] or lambda expressions in version 8 [26]), small changes to improve the ease of use (for example autoboxing/unboxing or enhanced for loops [9]) and new or improved libraries or frameworks (for example the Swing framework [2] or the JDBC API [10]). Research to keep improving languages is still going on (for example anchored exceptions [135]).

The Unified Modeling Language (UML) [36] is a modeling language that can be used at different stages prior to implementation. The UML is the result of a joined effort of the *Three Amigos*. Rumbaugh, Booch and Jacobson attempted to reconcile their methods, respectively OMT [119], Booch [56] and OOSE [92], into a Unified Method. The result of their effort was not a method but a modeling language: the Unified Modeling Language. Their goal was not to formulate a new language, but to bring together the best of their methods because each was quite good at some area but none covered the whole process. An outstanding value of the UML is its standardization.

Patterns. Next to improving (programming) languages, experience leads to guidelines and best practices for certain kinds of problems. Those solutions can be reused by other people by imitating the *patterns*. Since Gamma et al. published a catalog of *design patterns* [83] the use of patterns is a well-accepted practice in software development. Design patterns are the result of the experience of developers and by describing them in a structured way newcomers can benefit from insights. Applying patterns means applying proven

good solutions. Design patterns don't necessarily work in all circumstances, but have the advantage of being flexible. Developers can tune a pattern to fit in a concrete situation. This advantage is at the same time a disadvantage because a pattern is only available as a recipe. Sometimes, a pattern can grow to a language feature (for example the *Typesafe enum pattern* described by Bloch [54] became in Java 5 a language feature [9]). Patterns often illustrate the shortcomings of programming languages.

Model Driven Development. Today, there is a growing trend towards using models instead of code and informal diagrams. Model Driven Development (MDD) [123] is a software paradigm that emphasizes the use of models as the primary artifact in all phases of the development life-cycle. The implementation can then be generated (automatically or semi-automatically) by transforming the higher-level models to lower-level models and in the end to executable code. Each transformation adds more platform specific details to the model. This approach certainly values the models at the higher levels better.

The Object Management Group (OMG) [1] proposed a standardized approach to MDD: Model-Driven Architecture (MDA) [59, 82, 94]. MDA is built around several OMG standards. The Meta-Object Facility (MOF) [33] is a four-layered architecture for *meta-modeling*. MOF is used to define the Unified Modeling Language (UML) [96, 120], the general-purpose modeling language used to develop the models at the higher levels of abstraction. The Object Contraint Language (OCL) [32, 143] describes rules (constraints or behavior) in for example UML models. The principles of Design by Contract (DBC) [106] offer an approach to define the behavior.

MDA defines two kinds of models: the Platform Independent Model (PIM) is the more abstract model providing a formal specification of the software system abstracting away platform specific details, and the Platform Specific Model (PSM) augments the PIM with technical details from a specific implementation technology. A PIM can be *transformed* into several PSM's. The process can be further subdivided by providing more layers of abstraction: there can be multiple PIM's and PSM's at different levels of abstraction. Important benefits claimed by the MDA are that the development of a software system is protected against advancing technology and changing requirements are better supported. The knowledge about the system is captured in the PIM and by defining new transformations new technology can be used. When business rules change, those changes are made to the PIM and propagate (almost) seamless to the lower-level models.

1.1.4 No Silver Bullet

> *Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any – no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. If this is true, building software will always be hard. There is inherently no silver bullet.*

In his article, *No Silver Bullet* [60], Brooks differentiates between *essential* complexity and *accidental* complexity. Accidental complexity is caused by the way software is developed. These problems are in some sense created by the software developers, but can be solved by advancing and improving technology. Better software development processes and better software building techniques reduce this complexity step by step. In the early years of software development, when software was written in assembler, this complexity was obviously bigger.

Essential complexity is caused by the problem to be solved by the software system. If the problem has a high complexity, solving it will also be complex. No matter how good our tools or techniques are, the inherent complexity of a problem will not go away. In a talk about *the software crisis*, Dijkstra says that the software crisis arose from the given that “programming is apparently much harder than we think” [144].

One can ask if the software crisis was not more a matter of perception and of managing realistic expectations. Aren’t we just ever pushing the limit of what is possible? And aren’t we doing a very good job? By reducing the accidental complexity through the building of better tools, better methods, better techniques,... the productivity and quality of software has grown tremendously. Moreover very complex and large software systems can be built today. Difficult problems from the past became trivial problems today. By continuing to reduce accidental complexity, we succeed at solving problems with ever more essential complexity.

Software is perceived as something infinitely malleable. This is a characteristic no other engineering discipline working with “real, physical” materials has. This characteristic must not lead to the expectation that changing software should be easy. For sure, editing code files is easy but that is not

exactly the same as changing the software system. Changing software in the first place means changing essential complexity and one should never underestimate that complexity. That complexity can never be removed. Changing software however also introduces accidental complexity, for example maintaining consistency between all models. A realistic goal is to reduce this accidental complexity as much as possible.

The goal of this dissertation is to contribute to the reduction of the *accidental* complexity of building software systems.

1.2 Contributions

This dissertation contributes on three different places in the development process to reduce the *accidental* complexity. The first contribution proposes improvements at *implementation stage* by presenting new techniques in the form of patterns and a language construct. The second contribution is situated at the *analysis stage* and/or *the highest levels of the design phase*. Another language construct is proposed as a new technique. Finally, the third contribution proposes techniques to improve transformation form higher-level models to lower-level models.

The contributions of this dissertation are threefold :

- **Approach to implement properties and associations.** Typical properties and associations are accompanied with requirements restricting the values of properties or objects an object can be associated with. These requirements can be divided in three different types : (1) requirements restricting the values independent of the state of the object (2) requirements restricting allowed combinations of values of different properties and associations (3) requirements restricting transitions to new values in view of current values. Implementing properties or associations with respect to these requirements in a clean way avoiding errors, code duplication and facilitating redefinition in subclasses is not straightforward. A first objective of this dissertation is to introduce a family of patterns that allow the developer to implement properties and associations and their requirements in a correct way. A second objective of this dissertation is to set a first step towards a language extension to avoid the technical code of the patterns.

- **Improving behavioral modeling in conceptual models.** The goal of a conceptual model is to describe the real world as it is observed

and not yet the software system that needs to be developed. Because analysts are mostly familiar with building software systems, they are sometimes tempted to use techniques used to build software when building the conceptual model. These techniques are however not always a proper solution. When describing behavior with respect to the history, the technique of reifying is often used because the modeling language lacks expressive power to reason about events. A third objective of this dissertation is to extend the expressive power to model behavioral facts through introducing a new construct to allow the analyst to model these behavioral facts in an elegant way.

- **Transformation tool.** From the start of the development, models are used to describe the software systems. In general, these models are worked out using different modeling languages. The chosen modeling languages can also vary from project to project (for example to describe database structures Entity-Relationship (ER) diagrams [45] are an often used alternative for UML). These different modeling languages mostly share common generic composition structures. Transformations from the higher-level models (the conceptual model is the highest level) to lower-level models and at some stage to source code can be (semi-)automated. A fourth objective of this dissertation is to build a prototype transformation tool that offers constructs for building composition structures in modeling languages and concepts for defining transformations that benefit from these constructs.

Besides the research described in this dissertation, some work has been done in the context of the EROOS method. EROOS, Entity-Relationship Object-Oriented Specifications, is an object-oriented analysis method developed at the research group on Software Development Methodology of the department of Computer Science of the KU Leuven. It has served as a framework for ongoing research on object-oriented research on object-oriented software development [48, 50, 51, 75, 76, 77, 78, 122, 134]. The contribution resulted in this publication: *P. Bekaert, G. Delanote, F. Devos, and E. Steegmans. Specialization/generalization in object-oriented analysis: strengthening and multiple partitioning. In J.-M. Bruel and Z. Bellahsene, editors, Advances in Object-Oriented Information Systems, pages 34-43, 2002* [49].

1.3 Overview of the Text

This dissertation is organized in 5 chapters. In addition to this introduction, the remainder of this dissertation is organised in the following chapters:

Chapter 2, *A Pattern-based Approach towards Expressive Specifications of Properties and Associations*, proposes a family of patterns to implement *properties* and *bidirectional associations.* A language construct is proposed as an alternative for the patterns. First, a number of principles are presented that steered the development of the patterns. Next, a taxonomy for the requirements that typical accompany properties and associations is presented. Finally, the family of patterns is presented, first for properties and then for associations. The code samples are written in Java. The work of this chapter is (partially) presented in *G. Delanote, J. Boydens, and E. Steegmans. A pattern-based approach towards expressive specifications for property concepts. In L. Lavazza, R. Oberhauser, A. Martin, J. Hassine, M. Gebhart, and M. J'antti, editors, ThinkMind // ICSEA 2013, The Eighth International Conference on Software Engineer- ing Advances, pages 249-257, 2013* [72].

Chapter 3, *Concepts for Abstracting away Object Reification at the level of Platform Independent Models (PIMs)*, proposes a language construct to model behavior at the level of conceptual models (or the higher-level design models). The chapter starts with the presentation of key principles for conceptual modeling. Problems with UML to model behavior in the context of history are described, and a solution is presented via the proposal of a new operator: the #-operator. Finally, a guideline on when to use that operator is defined. The work of this chapter is (partially) presented in *G. Delanote and E. Steegmans. Concepts for abstracting away object reification at the level of platform independent models (PIMs). In R. Machado, J. Fernandes, M. Riebisch, and B. Schätz, editors, Proceedings of The Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pages 94-102, 2006* [74].

Chapter 4, *A Framework for Executing Cross-Model Transformations Based on Pluggable Metamodels*, proposes a prototype for a framework for metamodels and model transformations. Generally, a metamodel is built using a meta-metamodel. But developing a metamodel benefits also from a concrete framework for metamodels that offers generic structural constructs. Such a framework not only eases the building of a metamodel, the definition

of transformation algorithms can also favor from it. Generic transformation logic can also be offered in a transformation framework. The work of this chapter is presented in *G. Delanote, S. De Labey, K. Vanderkimpen, and E. Steegmans. A framework for executing cross-model transformations based on pluggable metamodels. In Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT07), pages 315-325, 2007* [73].

Last, we conclude this dissertation in Chapter 5 with a summary and overview of the major contributions and identify some directions for future work.

Intellectueel proberen een ervaring te begrijpen is zoiets als proberen een vlam vast te pakken. Als je probeert om dergelijke dingen intellectueel te snappen, dan heb je van het vuur alleen de as te pakken.
– Bert Hellinger

Chapter 2

A Pattern-based Approach towards Expressive Specifications of Properties and Associations

Chapter Summary

Properties and associations are typically accompanied with requirements restricting properties ascribed to objects or associations of objects with other objects. Three different types of requirements are identified to facilitate the (re-)definition of requirements at different positions in the class hierarchy: *Value Requirements*, *State Requirements* and *Transition Requirements*. A *separation of concerns* between the development of methods describing requirements and the methods describing the state changes is reached by encapsulating the description of the requirements in their own *inspectors*. A family of patterns describes how all methods describing the state and behavior of a property or association collaborate. Finally, a first step towards a language extension to avoid the technical code of the patterns is presented.

2.1 Preamble

The research of this chapter started in 2002 in the context of the course Object-Oriented Programming [127]. The course turns concepts and principles underlying the object-oriented paradigm into concrete coding advices and coding rules. Both specification and implementation are equally important.

At that time, there were different trends concerning object-oriented programming. One trend was built around Smalltalk [85], the first true object-oriented programming language, and focused especially on implementing adaptable and extendible software systems. Techniques like reflection and meta-programming [66] originated from this trend. Another trend was built around strongly typed languages where Eiffel [22] had a pioneering role. Eiffel differs from other object-oriented programming languages in the sense that the language itself is not only a programming language, but also a software development method, based on a small number of key ideas from software engineering and computer science [104]. Essential to Eiffel is Meyer's reliability mechanism, Design by Contract [106], which is an integral part of both the method and language. New languages, like Java [24] and C# [18], were influenced by both Smalltalk and Eiffel. Both languages however have poor support for Design by Contract.

With regard to the software development process, this research is situated in the implementation activity.

Discussions with students and mentors in the context of the course learned that programming properties and associations with their inherent requirements is far from an easy task. Mostly, the problems arose from the inability to differentiate between the different kinds of requirements, or, in other words, from trying to handle all requirements uniformly. Some requirements must hold at all times, while other requirements must only hold to be allowed to evolve to the next state. Some requirements are related to a single characteristic (property or association), while other requirements are related to multiple characteristics at the same time. This led for example very often to code duplication. Students generally have trouble to see the difference between those requirements and to put the required checks at the right places.

Requiring adaptability and extendibility in both dimensions, time and space, makes this task even harder. Redefinition in the dimension *time* means that certain requirements change at a later point in time, while redefinition in the dimension *space* means that certain requirements need to change at the level of a subclass. Writing correct specification that leaves room for redefinition has proven to be very difficult.

The patterns are the result of a growing understanding of the problem. The first important insight was to encapsulate the requirements in their own methods, *inspectors* and extract that code and specification from the mutators. In a next step, requirements that must always hold, and thus describe the *class invariants*, were separated from the requirements restricting transitions. Both were described in their own dedicated inspector. This intermediate version of the patterns was used in the implementation of the framework presented in chapter 4. Requirements involving multiple characteristics still often led to code duplication and therefore requirements related to a single characteristic were separated from requirements related to multiple characteristics. At this stage, the patterns worked well for properties but still struggled with associations because multiple classes were involved.

The results of this research are partially presented in November 2013 in The Eighth International Conference on Software Engineering Advances [72].

Recently, languages start to support properties better but don't pay much attention to their requirements. Concerning associations, offered techniques like reference semantics and data structures are insufficient. van Dooren [137] proposed a code inheritance relation for reusing general purpose classes as components for other classes. Verification of code has become an important topic [91]: structuring requirements in well defined manner certainly facilitates verification.

2.2 Introduction

Object-oriented languages were built to increase the quality of software applications [81]. Encapsulation, inheritance, polymorphism and dynamic binding incontestable increase the quality of software. Encapsulation hides technical details of a class. Internal data of a class is shielded from unwanted, direct access from a random place in the program. Declarative descriptions of the behavior of objects hide operational details. Inheritance benefits the reusability and adaptability of code. The use of a class can be extended and code duplication can often be avoided. Inheritance makes it possible to extend and to specialize existing code. Polymorphism and dynamic binding reduce complexity by allowing an object to take more than one form and still show the correct behavior.

Object-oriented programming languages use classes as abstract data types. A class is a blueprint for a collection of objects with identical characteristics and behavior. Characteristics comprise both properties and associations. An

example of a property is the balance of a bank account, while the relation between a person and a bank account representing the holdership is an example of an association. Generally several requirements have to be enforced for those characteristics. Examples of requirements for the property balance of a bank account are (i) the balance is represented with an (effective) decimal number and is rounded to two significant numbers after the decimal point and (ii) the balance must never be below the credit limit. Requirements for an association are for example (i) each bank account must have exact one (effective) holder and (ii) holdership can only be transferred to a relative. Current object-oriented programming language constructs lack expressiveness to describe those requirements in an integrated way.

Although properties and associations with their inherent requirements are ubiquitous in software applications, no standardized solution exists to implement them. In this chapter, we present a pattern to implement characteristics with their requirements. Using software patterns is an important way to increase the quality of software systems [83, 105]. Gamma et al. write in the introduction of Design Patterns, Elements of Reusable Object-Oriented Software [83]: *Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable.* We identify three different kinds of requirements. The pattern tackles each kind in an appropriate way. The ideas and concepts in the pattern to develop properties versus associations are very similar, but because the realization of the pattern for associations is significantly more complex we present them separately.

Although using patterns is a significant step forward, Meyer says in Object-Oriented Software Construction [104]: *"To go beyond their mere pedagogical value, patterns must go further. A successful pattern cannot just be a book description."* Different strategies exist to overcome the disadvantages of patterns. The best known strategies are software components [43, 104], language support [57, 64, 84] and tool support [62, 64]. We will indicate where tools can support the usage of the pattern. But as we also firmly believe in more expressive language concepts to improve the quality of software systems, we illustrate also how a new language construct can replace the pattern.

Overview

In section 2.3, the context and motivation to develop a family of patterns is described. The principles that steered the development of the patterns are presented in section 2.4. Section 2.5 investigates the different kinds of requirements that describe the business rules that restrict properties and associations. In section 2.6, the patterns are built step by step for properties. Next, in section 2.7 the patterns are first developed for unidirectional associations and then extended to bidirectional associations. The used approach in the patterns is to enforce requirements before changing some characteristics, section 2.8 presents an alternative approach. Finally, we conclude in section 2.9.

2.3 Motivation

2.3.1 Software Quality

Why is software quality so important? In [100], Mann notes: "*In the last 15 years alone, software defects have wrecked a European satellite launch, delayed the opening of the hugely expensive Denver airport for a year, destroyed a NASA Mars mission, killed four marines in a helicopter crash, induced a U.S. Navy ship to destroy a civilian airliner, and shut down ambulance systems in London, leading to as many as 30 deaths.*" It's evident that software quality is important, but it's not so easy to define software quality. Meyer states that software quality is best described as a combination of several factors. Of all quality factors *modularity* (extendibility and reusability) and *reliability* (correctness and robustness) stand out [104].

We have centered the specification and development of our pattern along the following quality factors.

O1 - Correctness. Software must perform its task as defined by the specification. The pattern defines specific methods to work out the different aspects of the implementation of a characteristic forcing the developer to think about each aspect in isolation.

O2 - Extendibility. Software must be adaptable to future changes of the specification. These changes can be in space (through adding a subclass that redefines some aspects) or in time (changes to the specification in the future). The pattern provides the necessary hook methods to be able to change the specification easily. The pattern also guides the developer to specify and implement each aspect only once.

O3 - Testability. Testing the correctness of software must be as easy as pos-

sible. Different aspects of the implementation of a characteristic are worked out in separate methods. The methods are designed in such a way that they can be tested in isolation.
O4 - Understandability. A programmer must understand as easy as possible the source code of a software system. Dividing a big problem into smaller problems is a well-known strategy to make a problem easier to understand. The pattern separates the code, the developer has to write, from boilerplate code to make the code more readable.
O5 - Reusability. Software should be usable in different applications. Extendibility already mentions the provided hook methods to change the specification easily. These methods make it also easy to reuse the software in a (slightly) adapted form in another application.
O6 - Expressiveness. The ease for a developer to write software. By forcing the developer to implement the different methods, the pattern also guides the developer through the different aspects of implementing the characteristic. This way the developer can think more on *what* must be implement instead of on *how* he can accomplish it.

We raise the ambition level for each of these objectives when compared to the current state of the practice.

2.3.2 Patterns

Patterns have been introduced by Christopher Alexander in the field of architecture. In [40], Alexander writes '*Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*".

In the mid-nineties, the idea of design patterns was introduced in the field of (object-oriented) software development [83, 114]. A design pattern names and describes a recurring design problem and provides a reusable solution to that problem. Gamma et al. define a design pattern as follows: "*A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design*" [83, p.3].

A design pattern results from the experience of software developers. A key quality for each design pattern in Gamma et al. [83] is the fact that it is a proven design solution preventing us from reinventing the wheel again and again [141]. The pattern presented in this chapter is built on the experience gained from discussing pieces of code from and with students in the

context of the course Object-Oriented Programming [127]. We observed students struggling over and over again with the same question: "How and where do I implement this (kind of) requirement without having to duplicate the code or refactor my code in a later stage when new requirements are added?"

It is generally claimed that applying design patterns yields *better* software [43, 53, 64, 105]. Bishop [53] says design patterns are one of the most successful and recognizable abstraction tools that software engineers have at their disposal. Prechelt [113] says following advantages for design patters are claimed:

1. Using patterns improves programmer productivity and program quality.
2. Novices can increase their design skills significantly by studying and applying design patterns.
3. Patterns encourage best practices, even for experienced designers.
4. Design patterns improve communication, both among developers and from developers to maintainers.

Ampatzoglou et al. [41] nuance these claims. They studied the effect of the Gamma et al. patterns [83] on software quality and concluded "*Patterns are about trade-offs. Design patterns enhance one quality attribute in the expense of another.*"

The pattern presented in this chapter differs in a crucial way with the Gamma et al. patterns: where the latter act at the level of classes or objects, the former concentrates on characteristics from classes. The number of classes involved is restricted to one for properties and two for bidirectional associations. We will describe how the presented pattern enhances the quality factors listed in section 2.3.1.

2.3.3 Language Support

Although patterns are a valuable instrument, they have their drawbacks. Ampatzoglou et al. [41] also included some metrics in their study. Using patterns had for example never a positive effect on the number of code lines (LOC). We consider software quality factors like modularity and reliability far more important. Tool support can be a powerful aid in reducing the implementation overhead because patterns typically contain a significant amount of boilerplate code. In Eclipse [21], custom templates can be defined to generate skeletons of methods.

When considering *reusability* a design pattern offers a design solution that can be used over and over again. But this also means, you cannot reuse a

worked-out solution in terms of code. Gamma et al. [83] provide a step-by-step approach on how to apply a design pattern, the last step says: "*7. Implement the operations to carry out the responsibilities and collaborations in the pattern. The Implementation section offers hints to guide you in the implementation.*" Meyer [104] summarizes it as follows: *A successful pattern cannot just be a book description; it must be a software component.* In [64], Chambers, Harrison, and Vlissides debate about whether design patterns should be supported by tools rather than by languages. Chambers advocates language designers should study design patterns to gain inspiration in developing new abstraction mechanisms in their languages.

The pattern presented in this chapter covers properties and associations which are the core of a class. As almost every characteristic comes with specific requirements, we believe that it's reasonable to add a new construct to programming languages to cover the goal of the pattern.

2.4 Principles and Notation

We follow the principles and notations introduced in the book *Object Oriented Programming with Java* [127]. The programmer can choose between three different paradigms to deal with exceptional circumstances: nominal, defensive and total programming.

Nominal programming uses preconditions to prohibit method invocations under exceptional conditions. Methods only guarantee their effect when the conditions described in the preconditions are met, if those conditions are violated the effect of the method is undefined. As an example, consider a method for withdrawing money from a bank account. Trying to withdraw a negative money amount from a bank account can be seen as an exceptional case. With the nominal programming paradigm a precondition stating that only positive money amounts are valid for withdrawal may be added.

Defensive programming uses exceptions to signal that methods have been invoked under exceptional conditions. When the normal effect of the method cannot be achieved, an exception is thrown to signal the exceptional case to the caller. The state of all involved objects remains unchanged compared to the state upon entry of the method throwing the exception. As an example, a method for withdrawing money from a bank account may throw an exception when invoked with a negative money amount to withdraw.

Total programming turns exceptional conditions into normal conditions. All methods always end in a normal way. The developer must handle all pos-

sible conditions under which a method can be invoked, or in other words a method must be able the handle all possible combinations of values of its arguments. As an example, a method for withdrawing money must also end in a normal way when invoked with a negative money amount. The developer could choose to leave the balance of the bank account unchanged when the method is called with a negative money amount.

In the remainder of this chapter, the examples are worked out in using the defensive programming paradigm. Transformation to the other paradigms is straightforward. The next sections introduce some important principles that steered the development of the pattern for implementing properties and associations.

2.4.1 Design By Contract principle

The Design By Contract approach (DBC) is developed by Bertrand Meyer as part of the Eiffel programming language [22, 104, 106]. Design by Contract is an approach to specify and implement object-oriented software components based on precisely defined *contracts* between them. The key concepts of DBC are *preconditions*, *postconditions* (introduced by Hoare [89]) and *invariants*. These concepts are used to provide the interfaces of the software components with formal and precise specifications. In general, a contract implies obligations and benefits for both parties, the *supplier* and the *client*. Software components collaborate with each other on the basis of mutual obligations and benefits. The client of a software component, the *caller*, has the obligation to ensure the preconditions to obtain the benefit, namely to be guaranteed of the postconditions. On the other hand, the supplier, the *callee*, has the obligation to fulfill the postconditions. The supplier has the benefit he may assume the preconditions.

Commonly used languages like Java, C++ [98] and C# [18] have no support for DBC. However several third-party tools have been developed for those languages. Tools for java are for example: Contract4J [19], JContractor [93]. The Java Modeling Language [97] is a behavioral interface specification language that can be used to specify the behavior of Java modules. Object-oriented languages with native support for DBC are for instance Sather [34], Nice [35] and Spec# [31].

2.4.2 Command-Query Separation principle

The command-query separation principle (CQS-principle) was devised by Bertrand Meyer [104]. The principle states that we should make a clear distinction between *queries* and *commands*. In this text, we use the terms *inspector* instead of query and *mutator* instead of command. Inspectors return information about the state of some objects. Mutators change the state of some objects. The CQS-principle prohibits methods to combine both aspects: inspectors should not change the observable state of one or more objects and mutators should not return a result. Meyer defines this rule informally as *asking a question should not change the answer* [104, p. 751]. Following the CQS-principle leads to simple and readable code and tremendously helps reliability, reusability and extendibility. The CQS-principle also benefits Design By Contract [106] because inspectors can be used in the assertions specifying the semantics of the class.

We further distinguish between basic queries and derived queries. A basic query returns part of the state of an object. The (observable) state of an object is determined by the set of all basic queries. This set must be minimal, which means that it may not be possible to leave out a basic inspector from the set and still be able to retrieve the full (observable) state of the object. The result of derived queries and the effect of mutators is always directly or indirectly specified in terms of basic queries.

2.4.3 Liskov Substition Principle

Creating class hierarchies through inheritance is an important instrument for writing programs. Subclasses are allowed to extend or change functionality of superclasses. However, it's important to avoid undesired effects in the behavior: the definition of an overriden method may not contradict with the definition of the method at the level of the superclass. The Liskov Substitution Principle [99] (LSP-principle) is the guiding principle to describe which changes are allowed when overriding a method. Broadly speaking, the principle states that it must always be possible to substitute in a transparent way objects of a superclass by objects of its subclasses. The LSP-principle is a behavioral subtyping principle that guides developers in building solid class hierarchies by demanding that subclasses preserve behavior from superclasses completely.

Preconditions and postconditions are the main instruments to describe the definition of a method. *Preconditions* state the conditions that must hold each

time a method is called. It's the responsibility of a client to satisfy the imposed preconditions when invoking a method. *Postconditions* state the conditions that the method must satisfy when it returns. It's the responsibility of the developers of a method to satisfy the stated postconditions. When a subclass overrides a method, the definition in the subclass must still comply with the definition of the superclass. The LSP-principle states that preconditions can only be weakened at the level of a subclass. A method at the level of a subclass may only require less or weaker conditions from a client and is not allowed to increase those conditions. The LSP-principle states that postconditions can only be strengthened at the level of a subclass. A method at the level of a subclass may only add or increase effects to the method and is not allowed to remove or decrease effects. Strengthening postconditions is only possible if the definition of the method at the level of the superclass is not completely determined. If the definition of a method is not completely determined, the method is said to be *non-deterministic*.

The LSP-principle has also an impact on other ingredients in the definition of a class. The LSP-principle describes how class invariants, representation invariants, exceptions, return types, formal arguments and access rights can be changed at the level of the subclass. For a more detailed description of the LSP-principle we refer to work from Meyer [104] and Steegmans [127].

2.4.4 Open-Closed Principle

One of the typical characteristics of software is that it evolves, requirements frequently change over time. It is almost impossible to foresee all required functionality: new or changed functionality often needs to be added to the application. The application must be *open* for future changes or extensions. On the other hand clients using an application need the guarantee that the definition of the functionality will not change over time. Thus, once available for clients the contracts of an application must be *closed*. The open-closed principle [101] was introduced by Meyer [104] and says that *software entities should be open for extension, but closed for modification.* Once a class is published or made available for clients, the only reason to modify it should be to correct errors. Inheritance is the key technique to extend or tune a class towards a specific context. The principle however does not imply that new methods can no longer be added to a class. A class is allowed to offer new behavior over time, as long as this behavior does not interfere with the existing code. This principle is an essential requirement to reach a high level of reusability.

2.4.5 No-Choice Principle

Devos [75] defined the principle of no-choice at the level of conceptual modelling as *the principle of no-choice states that a given set of real-world facts S can be only represented by just one model. A method should not allow different equivalent alternatives for expressing a given set of real-world facts. It is not the responsibility of the analyst to decide how to model reality. An analyst has to concentrate on the requirements and the real-world facts themselves.* The same applies at the implementation level. A programmer should spend his time on solving (real) problems and not on technical issues. As an example consider the GOTO-statement. Early programs using the goto-statement are mostly very hard to read and understand, they consist of so-called "spaghetti-code". To solve those problems, at first, programming patterns were used to structure the technical code and to write the problem solving code at the right place. Step by step, better (more expressive) language features appeared which made the GOTO-statement always a worse alternative before it disappeared from most programming languages. The programming language and the programming principles must enable the programmer to write clean and readable code, by giving them the right tools and guidelines to focus on the unique problem they have to solve.

2.4.6 Notation

In Java, the contract of a class is worked out in documentation comments, which can be processed by javadoc [25]. Tags structure the different pieces of the specification in the documentation. The specification of a class is described both formally and informally. The informal specification is written in natural language, while Java boolean expressions are used to write the formal specification.

The following tags are used in the code snippets throughout this chapter:

- @basic: denotes a basic query
- @effect: specifies the semantics of a mutator in terms of another mutator
- @invar: denotes a class invariant
- @post: specifies a postcondition of a mutator
- @return: specifies the result of a derived query
- @throws: specifies the exception that must be thrown when the specified assertion evaluates to true

2.5 Requirements

This chapter focuses on properties of classes, e.g. the balance of a bank account, and associations between classes, e.g. the association between a person and a bank account describing the holdership relation. Properties and associations are almost always restricted by business rules. For example, the balance of a bank account may never drop below the credit limit or the holder of a bank account must always be an adult person.

We have identified three different types of requirements and discuss each of them in the next sections : (1) Value Requirements, (2) State Requirements and (3) Transition Requirements.

2.5.1 Value Requirements

These requirements are used to specify the most basic kind of business rules in that they restrict the range of values that a characteristic, property or association, can have. Meeting its Value Requirements is a necessary condition for an object to be in a steady state, or in other words to meet its invariants. A Value Requirement never takes into account other characteristics of the class at stake.

For *properties* a Value Requirement restricts the set of values offered by its type further. A Value Requirement is for instance used to enforce that the credit limit of a bank account always needs to be below 0.

For *associations* a Value Requirement restricts the kind of objects (type) that can participate in the association. At the level of the definition of an association, the only decision that needs to be made is whether non-effective objects (null values) are acceptable. Prohibiting non-effective objects in an association with restricted multiplicity (zero or one), means declaring the multiplicity to one. This type of requirement is also know as *existential dependency.* Considering generalization/specialization, a redefinition, that restricts the kind of objects a specialization can be linked with, is also a Value Requirement. The requirement that a savings account always needs to be linked with a current account is a Value Requirement (existential dependency). The requirement that junior current accounts and junior savings accounts, specializations of current account and savings account respectively, are attached to the right specialization is enforced with a Value Requirement.

2.5.2 State Requirements

Mostly, business rules restrict possible values for a characteristic when considered in combination with values from other characteristics. State requirements are by nature *symmetric*. A State Requirement involving characteristics α and β is always a State Requirement for both characteristics. Meeting its State Requirements is the other necessary condition for an object to be in a steady state. The union of all Value Requirements and State Requirements describe all the invariants of a class. The business rule stating that the balance of a bank account must never be below the credit limit is specified by a State Requirement. In the context of associations, the business rules requiring that a bank card attached to a blocked bank account is itself also blocked and that the credit limit of a bank account is restricted to -1.000 euro when the holder of the bank account is not an adult person are also State Requirements.

2.5.3 Transition Requirements

These very specific requirements specify the business rules that restrict the evolution of values of characteristics. It's perfectly possible that a (new) value for a characteristic meets all value and State Requirements but is not acceptable because of the current state of the object. The business rule imposed by a bank limiting the amount of money that can be withdrawn from a bank account is a Transition Requirement. Although 1.000 euro is a correct balance, it's not an acceptable balance after a withdraw operation when the current balance is 10.000 euro and the withdraw limit is 5.000 euro. As an example for associations, consider marriage as a relation between two persons. Two persons can meet all requirements to be allowed to be married with each other, for example both being adult, but can not really marry as long as at least one of them is married (to someone else).

2.6 Properties

In this section, we build the pattern for properties step by step. These steps give a good indication of what an iteration of the development process can consist of. It is possible to elaborate the different requirements independent of each other. Typically a pattern contains boilerplate code, we will highlight those parts in the code listings. A code editor should be able to generate this code. In Eclipse [21], custom templates can be defined to generate skeletons of methods.

In the examples illustrating the different parts of the pattern, we mostly omit the informal specifications in this text to reduce the size of the code snippets. This doesn't mean informal specification, specification in natural language, is less important. In particular, informal specification increases the readability or understandability of the code.

The example used throughout the next paragraphs describes a class of bank accounts. Each bank account has two characteristics, namely a balance and a credit limit. Both characteristics are decimal values and the balance must never be less than the credit limit. The amount of money that can be deposited or withdrawn in a single transaction must be restricted to 1000 euro. To explain the pattern in the context of inheritance, we introduce a class of junior bank accounts, a subclass of bank accounts. The balance and credit limit of junior bank accounts are restricted to integer values. At the level of the subclass, two new characteristics are introduced: each junior bank account has an integer value as upper limit and a blocked state (boolean). While the credit limit can no longer be less than -1.000 euro, the upper limit must at least be 1.000 euro and must not exceed 10.000 euro. The upper limit is an immutable characteristic. Of course, the balance is not allowed to exceed the upper limit.

2.6.1 Representation

Each observable characteristic is part of the state of an object and is revealed by a basic query. The basic query can be compared with the getter from Enterprise JavaBeans (EJB) [23, 118]. The return type of the basic query reveals the chosen type for the characteristic. The characteristic can internally be stored using one or more *instance variables* with the same or different types. The implementation of the basic query has to perform necessary transformations between stored and observable information. Like EJB, we introduce also a *setter* to change the characteristic to a given value. The basic query and this setter are the only two methods that are allowed to access the instance variables that represent the characteristic. By consequence, we limit the possible transformations between internal representation and observed value of a characteristic to these methods. This decision has a direct positive impact on quality factors like correctness, adaptability, testability. If at some point the internal representation needs to be changed, these changes are very local which clearly improves the adaptability of the code. The chance for errors is also reduced because the transformation from internal to observable state is not repeated over and over in different places, preventing the developer from

forgetting to update an occurrence of the transformation for example. Finally, restricting the number of methods to test clearly makes testing easier and more reliable.

When clients of a class are not allowed to change the value of a characteristic directly and need to manipulate the characteristic through more complex mutators, the latter mutators must be implemented in terms of this setter. When there exists a default value for the characteristic then that value is always explicitly added to the declaration, even if that value is the default value of the type of the internal representation. Thus, absence of a default value in the declaration means this characteristic must always be initialized during construction.

```
1  private BigDecimal balance=BigDecimal.ZERO;
2
3  /**
4   * Return the balance of this bank account
5   */
6  @Basic
7  public BigDecimal getBalance(){
8    return balance;
9  }
10
11 /**
12  * Set the given balance as the balance of
13  * this bank account
14  * @post  new.getBalance() == balance
15  */
16 public void setBalance(BigDecimal balance){
17   this.balance = balance;
18 }
```

Listing 2.1: Representation of the property balance

Listing 2.1 illustrates the internal representation with default value, basic query and setter for the characteristic balance. As the stored and observed values are equal the implementation of both methods is trivial.

2.6.2 Value Requirements

For each property *a Boolean inspector* is introduced to validate the Value Requirements. This inspector is the only place where these requirements are specified and implemented. Because the result of this inspector is by definition independent of the state of the object, the inspector is a class method

(`static` in Java). By convention the name of the inspector checking the Value Requirement for a property α is `isProperValueFor`α`(T` α`)`.

According to the principle of completeness, all business rules must be enforced in the application. Calling the setter with an actual argument that violates the Value Requirement is an exceptional situation and must be signaled. The setter is adapted accordingly.

```
1  /**
2   * @return   if  ((creditLimit==null) || (creditLimit.signum() > 0))
3   *           then  result == false
4   */
5  public static boolean isProperValueForCreditLimit(
6                                           BigDecimal creditLimit){
7    return  (creditLimit != null) && (creditLimit.signum() <= 0);
8  }
9
10 /**
11  * @post   new.getCreditLimit() == creditLimit
12  * @throws IllegalArgumentException
13  *          !isProperValueForCreditLimit(creditLimit)
14  */
15 public void setCreditLimit(BigDecimal creditLimit)
16                                   throws IllegalArgumentException{
17   if (! isProperValueForCreditLimit(creditLimit))
18     throw new IllegalArgumentException();
19   this.creditLimit = creditLimit;
20 }
```

Listing 2.2: Value Requirement of the property credit limit

Listing 2.2 illustrates the inspector and setter for the characteristic credit limit. The specification of the inspector is worked out in a non-deterministic way. It specifies only which values are certainly not acceptable as value for the credit limit of a bank account. Notice however that the signature of the inspector `isProperValueForCreditLimit()` implies that only true or false can be returned as result. This way subclasses can decide to further restrict possible values or to explicitly confirm what values are always acceptable. If the developer of a class wants to prevent subclasses from redefining the Value Requirement, the specification must be made deterministic.

Using boolean inspectors to encapsulate requirements is a key technique throughout the pattern. The technique improves the quality of the code in multiple ways.

Firstly, it improves the testability of code. Unit tests don't have to change the state of an object to test the implementation of requirements. Thus, the result of these tests can not be influenced by code changing the state of an object. In the case of Value Requirements, an object isn't even required because these inspectors are static by definition.

Secondly, avoiding code duplication improves adaptability. These inspectors can be used in all places (e.g. constructor, setter, advanced mutator,...) where (value) requirements need to be checked. When requirements change, only the inspector encapsulating the requirements needs to be adapted.

Thirdly, subclasses often want to tune requirements to their specific needs. By isolating these requirements in inspectors, it is very clear how a subclass must adapt requirements, namely by redefining the inspector in a Liskov-compliant way. The pattern is developed in such a way that if only a part of the requirements (e.g. only Value Requirements) must be redefined, the other parts seamless can be reused. This also avoids the need to redefine the mutator (setter) of the property and therefore also reduces the chance of introducing errors.

Finally, it also significantly improves the readability of code. Encapsulating this part of the problem in its own inspector prevents that readers of the code have to search for this code hiding in mutators or other methods. As listing 2.2 already illustrates it also improves the readability of the setter.

2.6.3 State Requirements

A State Requirement describes a constraint that restricts acceptable value combinations of characteristics. A characteristic can be involved in multiple State Requirements. Each State Requirement is also described by *a Boolean inspector*. This inspector is again the only place to specify and implement the State Requirement at stake. The inspector has an argument for each characteristic involved in the State Requirement. Thus, this inspector is also a class method. A State Requirement is always symmetric, which means that in theory all involved properties need an analogous inspector. The only differences between those inspectors are the actual argument and the method name. In the example, these methods could be `balanceMatchesCreditLimit(BigDecimal balance)` and `creditLimitMatchesBalance(BigDecimal creditLimit)`. Although textual not literally, this is specification and code duplication that must to be avoided. Each subclass that needs to redefine this State Requirements, would need to redefine the inspector for all involved properties. To avoid this du-

plication, an argument is provided for each involved property. The remaining specification and implementation of all involved properties can easily use this inspector. An extra advantage is that this makes it also easier to test these inspectors, because the test doesn't need an object anymore.

Obviously, the value from each involved characteristic must meet the Value Requirement to have an acceptable combination of values. By convention the name of the inspector checking a State Requirement involving properties α and β is `isProper`$\alpha\beta$`(T1` α`, T2` β`)`. We will illustrate in the paragraph about Transition Requirements how these inspectors are integrated in the setter.

```
1  /**
2   * @return  if (!isProperValueForBalance(balance))
3   *          then result == false
4   * @return  if (!isProperValueForCreditLimit(creditLimit))
5   *          then result == false
6   * @return  if (creditLimit.compareTo(balance)>0)
7   *          then result == false
8   */
9  public static boolean isProperBalanceCreditLimit(
10                     BigDecimal balance, BigDecimal creditLimit){
11   return isProperValueForBalance(balance) &&
12          isProperValueForCreditLimit(creditLimit) &&
13          (creditLimit.compareTo(balance) <= 0);
14 }
```

Listing 2.3: State Requirement between balance and credit limit

Listing 2.3 illustrates the State Requirement between the properties balance and credit limit. The specification of this inspector is also non-deterministic, it is however also possible to close the specification and make it deterministic.

2.6.4 Invariant

The invariants for a class are described by the union of all Value Requirements and State Requirements. We say that a characteristic α meets its invariants if it meets the Value Requirement and all the State Requirements it is involved in. For each characteristic α we introduce *a Boolean inspector* to check whether a given value meets its invariants with respect to the current state of the object. By convention the name of this inspector is `canHaveAs`α`(T` α`)`. As this

method is the sum of the Value Requirement for α and all State Requirements where α is involved in, this method can be generated as a whole. With respect to the property α, the object is in a steady state if the current registered value for α meets its invariants. The inspector `hasProper`α`()` specifies the invariant for α. This method can also be generated.

Listing 2.4 illustrates these methods for the property balance. The inspector `canHaveAsBalance` is non-deterministic to allow new State Requirements in future subclasses. If new State Requirements are undesired the developer of this class can declare the inspector `final` and make the specification deterministic. The inspector specifying the State Requirement between balance and credit limit will be used in both the invariant inspector for balance and credit limit. It's important to note that if the developer wants to prevent subclasses to change the invariant with respect to α, it does not suffice to close the specification of the inspector `canHaveAsBalance`, all inspectors checking parts of the requirements of α must also be closed or deterministic. In other words, `isProperValueFor`α`()` and each `isProper`$\alpha\beta$`()` must have a deterministic specification.

2.6.5 Transition Requirements

A new value for a property must at least always meet the requirements described by the invariant. But often specific requirements restrict possible transitions when we take into account the current state of that property. The *Boolean inspector* `canHaveAsNew`α`(T` α`)` checks whether the given α is an acceptable new value with respect to the current state of the object. First of all, the new value must meet its invariants. The extra Transition Requirements are added on top of them. The setter uses this inspector as guard for new values.

Listing 2.5 illustrates this inspector and the adapted setter. Often a public setter will not be desired, mutators like `withdraw` and `deposit` are preferred above `setBalance`. It suffices to change the access modifier to `protected` (`private` doesn't allow subclasses to define custom mutators) and custom mutators can easily be specified in terms of this setter. Listing 2.6 shows the definition of the mutator `withdraw`.

2.6.6 Construction

Construction is an event with very specific semantics. After the complete construction process an object must be in a steady state. Because that is also

```
1  /**
2   * @invar hasProperBalance()
3   */
4  public class BankAccount {
5  ...
6  /**
7   * @return result==canHaveAsBalance(getBalance())
8   */
9  public final boolean hasProperBalance(){
10   return canHaveAsBalance(getBalance());
11 }
12
13 /**
14  *@return if (!getClassObject().isProperValueForBalance(balance))
15  *          then result == false
16  *@return if (!getClassObject().isProperBalanceCreditLimit(
17  *                                          balance, getCreditLimit()))
18  *          then result == false
19  */
20 public boolean canHaveAsBalance(BigDecimal balance){
21   return   getClassObject().isProperValueForBalance(balance) &&
22            getClassObject().isProperBalanceCreditLimit(balance,
23                                                   getCreditLimit());
24 }
25 }
```

Listing 2.4: Invariant from the property balance

```
1  /**
2   * @return   if  (!canHaveAsBalance(balance)
3   *           then  result == false
4   * @return   let  BigDecimal  difference =
5   *                   getBalance().subtract(balance).abs()  in
6   *           result == difference.compareTo(MAX_DELTA)<=0
7   */
8  public boolean canHaveAsNewBalance(BigDecimal balance){
9    return  canHaveAsBalance(balance) &&
10           (getBalance().subtract(balance).abs().
11                                  compareTo(MAX_DELTA)<=0);
12 }
13
14 /**
15  * @post new.getBalance() == balance
16  * @throws IllegalArgumentException
17  *           !canHaveAsNewBalance(balance)
18  */
19 public void setBalance(BigDecimal balance)
20                          throws IllegalArgumentException{
21   if (!canHaveAsNewBalance(balance))
22     throw new IllegalArgumentException();
23   this.balance = balance;
24 }
```

Listing 2.5: Transition requirement of the property balance

```
1  /**
2   * @effect setBalance(getBalance().subtract(balance))
3   */
4  public void withdraw(BigDecimal balance)
5                           throws IllegalArgumentException{
6   setBalance(getBalance().subtract(balance));
7  }
```

Listing 2.6: Advanced mutator withdraw

the first state of the object we don't have to compare the initial value of a characteristic with its previous value (there is none). Even when there is value assigned in the declaration to the instance variable, we don't consider that value as a 'previous' value. An immediate consequence is that we can't use the setter in the constructor. Because we still want to restrict the manipulation of the instance variable(s) to a single method we need to introduce a more basic setter: `register`α`(T` α`)`.

```
1  /**
2   * @post   new.getBalance() == balance
3   * @throws  IllegalArgumentException
4   *              !isProperValueForBalance(balance)
5   */
6  protected void registerBalance(BigDecimal balance)
7                                      throws IllegalArgumentException{
8    if (!isProperValueForBalance(balance))
9      throw new IllegalArgumentException();
10   this.balance = balance;
11 }
```

Listing 2.7: Basic setter for the property balance

Listing 2.7 illustrates the basic setter for the property balance. Because this setter will be used in the constructor, only the Value Requirement can be checked in this setter. This setter is also necessary when we want to introduce a complex mutator that manipulates two via State Requirements related properties. The developer will have to build a custom transition checker for that mutator, but that is a rather trivial task as all building blocks are available. Indeed, each Value Requirement and State Requirement is specified in its own inspector.

A steady state after construction means that all Value Requirements and State Requirements must be met. Unfortunately, we can't use the inspector `canHaveAs`α`(T` α`)` because this inspector assumes all other properties β, γ,... already have their value. As there is no order in the different assertions of the specification, using them is impossible. So we are forced to repeat the invariant conditions in the specification of the constructor. Fortunately, we can describe the semantics of the constructor in terms of other mutators, more in particular the basic setter, through the `@effect`-tag. This way we reduce the complexity of the specification and implementation. So we only need to

list all State Requirements in the `@throws`-clause. Listing 2.8 illustrates the constructor for the class of bank accounts.

```
1  /**
2   * @effect   registerBalance(balance)
3   * @effect   registerCreditLimit(limit)
4   * @throws   IllegalArgumentException
5   *           !isProperBalanceCreditLimit(balance, creditLimit)
6   */
7  public BankAccount(BigDecimal balance,BigDecimal creditLimit)
8                                   throws IllegalArgumentException{
9    if (!isProperBalanceCreditLimit(balance, creditLimit))
10     throw new IllegalArgumentException();
11   registerBalance(balance);
12   registerCreditLimit(creditLimit);
13 }
```

Listing 2.8: Construction of a bank account

2.6.7 Inheritance

On the one hand, a subclass can specialize a superclass. The subclass can adjust the semantics of inherited features. The Liskov Substitution Principle (LSP) acts as a guideline to describe allowed adjustments. On the other hand a subclass can extend the superclass with new features. We will illustrate how the pattern copes with specialization and extension.

A subclass may want to redefine the Value Requirement of a property. This means we need to be able to override the inspector checking the Value Requirement. Because the inspectors checking the Value Requirement are class methods and Java doesn't allow to override `static` methods, the way a Value Requirement is implemented in the pattern needs to be adapted. Clearly, these inspectors need to be instance methods but on the other hand they have class semantics, as their result is defined independent of the state of the object. Therefore, we move these methods to a *static inner class*. This static inner class implements the Singleton Pattern [83]: the object of the static inner class represents the outer class. The marker interface [55] `ClassObject` designates the static inner class.

```
1  public class BankAccount {
2    public static class COBankAccount implements ClassObject{
3    private static COBankAccount instance;
4
5    protected COBankAccount(){}
6
7    public static COBankAccount getInstance(){
8      if (instance == null)
9        instance = new COBankAccount();
10     return instance;
11   }
12
13   public boolean isProperValueForBalance(...) {...}
14   public boolean isProperValueForCreditLimit(...) {...}
15   public boolean isProperBalanceCreditLimit(...) {...}
16   }
17 }
```

Listing 2.9: ClassObject inner class for the class BankAccount

Listing 2.9 illustrates the inner class for the class of bank accounts. The methods with class semantics can be moved without modification, except from the removal of the keyword `static`, to the inner class. The specification and implementation of the instance inspectors using these methods can easily access them through the singleton object.

A first advantage of moving the inspectors with class semantics into an inner class is that, although they are instance methods, they can easily be identified as methods with class semantics. In other words, it supports the understandability or readability of the code. A second advantage is that they make it impossible for the developer to use the state of the object erroneously. The No-Choice principle with respect to this issue is enforced by the compiler. A third advantage is that it is still possible to test these methods without needing an instance of the outer class. The advantage that encapsulating these requirements in static inspectors gives, is retained.

If class B is a subclass of A, then the inner class of B must be a subclass of the inner class of A to be able to override methods from the inner class of A.

Listing 2.10 illustrates the redefinition of the inspector checking the Value

```
1  public class JuniorBankAccount extends BankAccount{
2    public static class COJuniorBankAccount extends COBankAccount{
3    /**
4     * @return  if (!super.isProperValueForBalance(balance))
5     *          then result == false
6     * @return  if (balance.scale()!=0)
7     *          then result == false
8     */
9    @Override
10   public boolean isProperValueForBalance(BigDecimal balance){
11     if (!super.isProperValueForBalance(balance))
12       return false;
13     return balance.scale() == 0;
14   }
15   }
16 }
```

Listing 2.10: Redefinition of the Value Requirement of the property balance

Requirement for the property balance. An extra constraint is added on top of the constraints defined in the class of bank accounts. The Liskov Substitution Principle restricts the possible redefinitions of the inspector. Only the undefined or non-deterministic part of the definition in the superclass can be changed by making it partially or completely deterministic.

The application now has two versions of the inspector checking the Value Requirement. The pattern must always use the right version. More in particular, the inspector must be invoked against the right 'class object'. *Dynamic binding* ensures using the right version of an instance method. Therefore, an instance method is introduced to retrieve the right 'class object'.

Listing 2.11 illustrates how the right Value Requirement inspector is invoked through 'dynamic binding'.

Adding new properties to the subclass is now straightforward. If a State Requirement involves a property α from the superclass, the inspector `canHaveAs`α`(T` α`)` needs to be redefined at the level of the subclass. Listing 2.12 illustrates how the new State Requirement between the properties balance and upper limit is added to the inspector checking the invariant constraints for balance. Listings 2.10 and 2.12 illustrate that redefinitions are easily developed. Value Requirements, State Requirements and Transition Requirements can be redefined independent of each other.

```
1 public class BankAccount {
2   public COBankAccount getClassObject(){
3     return COBankAccount.getInstance();
4   }
5
6   public boolean canHaveAsBalance(BigDecimal balance){
7     return  getClassObject().isProperValueForBalance(balance) &&
8             getClassObject().isProperBalanceCreditLimit(balance,
9                                                   getCreditLimit());
10  }
11 }
12
13 public class JuniorBankAccount extends ... {
14   @Override
15   public COJuniorBankAccount getClassObject(){
16     return COJuniorBankAccount.getInstance();
17   }
18 }
```

Listing 2.11: 'Dynamic binding' of a 'class method'

```
1 public class JuniorBankAccount extends BankAccount{
2 /**
3  *  @return if (!super.canHaveAsBalance(balance))
4  *          then result == false;
5  *  @return if (!getClassObject().isProperBalanceUpperLimit(
6  *                                    balance, getUpperLimit()))
7  *          then result == false
8  */
9 @Override
10 public boolean canHaveAsBalance(BigDecimal balance){
11   if (!super.canHaveAsBalance(balance))
12     return false;
13   return getClassObject().isProperBalanceUpperLimit(balance,
14                                                getUpperLimit());
15 }
16 }
```

Listing 2.12: A State Requirement involving the balance and the upper limit

2.6.8 Language Construct

Appendix A shows that an inherent problem with patterns is that they generate quite some boilerplate code. The need for patterns signals a lack of expressiveness of programming languages. Therefore, we present an extension to increase that expression power. Listings 2.13 and 2.14 illustrate how the example is completely worked out with a new language construct `Property`.

```
1  /**
2   * The balance of this bank account
3   * @Value balance != null
4   * @State balanceExceedsCreditLimit
5   *        balance.compareTo(creditLimit) >= 0
6   * @Trans balance.subtract(new.balance).abs().
7   *                                    compareTo(MAX_DELTA) <= 0
8   */
9  Property BigDecimal balance;
10
11 /**
12  * The credit limit of this bank account
13  * @Value creditLimit != null
14  * @Value creditLimit.signum() <= 0
15  * @State balance.balanceExceedsCreditLimit
16  */
17 Property BigDecimal creditLimit;
```

Listing 2.13: The class of bank accounts

The importance of specification is upgraded, by making it an integral part of the construct. The specification describes the different kinds of requirements. They act as guards to validate values in an update operation. The new language constructs must be an integral part of the programming language and consequently recognized by compiler. In this dissertation, a solution that can be compiled is not worked out, the language construct is illustrated with javadoc tags. Three new tags are introduced to specify the semantics of a `property`, one for each kind of requirement we identified in section 2.5. The assertions used in the specification are Boolean expressions. (1) Each Value Requirement is preceded with a `@Value`-tag. A Value Requirement may be split over multiple tags. (2) Each State Requirement is preceded by a `@State`-tag. Each property can be involved in an unlimited number of State Requirements. (3) Finally, a Transition Requirement is preceded by a `@Trans`-tag.

```
1 /**
2  * The balance of this junior bank account
3  * @Value balance.scale() == 0
4  * @State upperLimit.balanceDoesNotExceedUpperLimit
5  * @Trans !isBlocked
6  */
7 @Override
8 Property BigDecimal balance;
9
10 /**
11  * The credit limit of this junior bank account
12  * @Value creditLimit.compareTo(new BigDecimal(-1000)) >= 0
13  * @Value creditLimit.scale() == 0
14  */
15 @Override
16 Property BigDecimal creditLimit;
17
18 /**
19  * The blocked state of this ...
20  */
21 Property boolean isBlocked;
22
23 /**
24  * The upper limit of this bank account
25  * @Value upperLimit >= 1000
26  * @Value upperLimit <= 10000
27  * @State balanceDoesNotExceedUpperLimit
28  *        balance.compareTo(new BigDecimal(upperLimit)) <= 0
29  */
30 @Immutable
31 Property int upperLimit;
```

Listing 2.14: The class of junior bank accounts

A State Requirement is always symmetric, which means it applies equal to all properties involved. To avoid the need to duplicate the specification, requirements can be given a name. For value and Transition Requirements this name is optional, for State Requirements the name is mandatory. The actual specification is added to one of the involved properties, while the other properties refer to this specification through the name of the requirement. A name has to be unique to a property. Names of requirements can be used unqualified within their own class hierarchy as long there is no ambiguity. The fully qualified name of a requirement is `package.class.property.requirement`. By avoiding the duplication we fully support Parnas' principle [110] saying

that each fact must be worked out in one, and only one, place.

The specification is by definition non-deterministic. The semantics of an assertion Γ in a Value Requirement, State Requirement or Transition Requirement is:

```
if !(Γ)
then result == false
else result == Undefined
```

Thus, when the assertion Γ evaluates to false, the submitted value is not acceptable. On the other hand, when the assertion evaluates to true the value may be acceptable. The semantics of the Value Requirements of credit limit in listing 2.13 is that non-effective positive or zero decimal numbers are certainly not a good value for a credit limit. Negative values *can* be good values. Subclasses are allowed to further specify the *open* part. The requirements specified in a subclass are *added* to the requirements specified in the superclass. The Value Requirement of the credit limit in the class of junior bank accounts for instance now specifies that only strictly negative integer numbers are acceptable values.

Evaluation So far the pattern has only been applied to academic problems. These experiments show that about 70% of the code for defining properties is boilerplate code. As an example the full definition of class of bank accounts counts 360 lines of Java code. About 250 of these lines are boilerplate code. The typical Java programmer is not tempted to write all these lines in original definitions of classes. In particular, he will not be eager to encapsulate the different kinds of requirements in Boolean inspectors such as `isProperValueForBalance()`, `canHaveAsBalance()`, `canHaveAsNewBalance()`, etc. This either leads to duplicate code because the same requirement is repeated over and over again in different parts of the class definition, or it compromises adaptability in time and space. We therefore believe that more advanced language constructs are needed to introduce properties in classes. We still need to experiment with this pattern in the scope of industrial software systems. We expect the same results with respect to the mere definition of properties in such large systems. It is difficult to estimate how much of a complex software system is dealing with properties and associations. It is our experience that for such software systems this part is about one third of the system. The pattern gives the programmer the opportunity to focus more on the business at stake.

2.7 Associations

Next to properties, associations are a key instrument to develop software. In this section, the pattern is first build for unidirectional associations. In this text, only associations with restricted multiplicity are treated. In a second step, the pattern is extended to bidirectional associations. A bidirectional association can to some extent be seen as the union of two unidirectional associations. Those two unidirectional associations are however never allowed to behave like unidirectional associations, they are inseparable.

The structure of the pattern for associations is exactly the same as for properties. When dealing with bidirectional associations, some specific challenges need to be solved. These challenges arise from the fact that the code for describing a bidirectional association is spread over two classes.

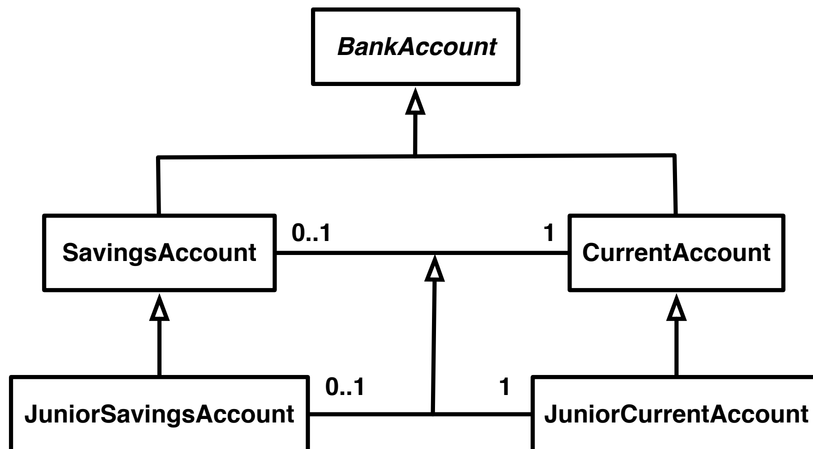


Figure 2.1: A bidirectional association with existential dependency

The example used throughout the next paragraphs reuses the bank account class from the previous section. Figure 2.1 illustrates the example. Two subclasses of `BankAccount` are defined, namely one representing savings accounts and one representing current accounts. The credit limit of savings accounts is redefined to an immutable property with predefined value 0. A savings account must at all times be attached to exactly one current account (existential dependency). The sum of the balances of the savings account and the attached current account must always be at least zero. If a current account

has no savings account attached to it, its balance is not allowed to be negative. Both savings accounts and current accounts have a junior variant. The junior accounts are only allowed to be attached to each other. The non-standard inheritance arrow between the two association lines in figure 2.1 is added to emphasize both association-lines refer semantically to the same association. The bottom association-line describes the extra requirements concerning the participants.

2.7.1 Unidirectional Associations

In this section, only one direction of the association between savings accounts and current accounts is considered, namely from current accounts to savings accounts. A current account can be attached to zero or one savings account. To enforce the constraint concerning the sum of the balances a bidirectional association is needed, so only the constraint forbidding a current account to have a negative balance when not attached to a savings account is retained in this section.

In a unidirectional association, the referred class should be unaware of the referring class. Besides exceptional cases, when the referred class needs functionality to observe or manipulate the association, the unidirectional association should be made bidirectional. By consequence, all code describing the unidirectional class will be in the referring class. In the example, `CurrentAccount` is the referring class.

2.7.1.1 Representation

The representation of an association with restricted multiplicity is trivial and completely comparable with the representation of properties. As internal representation, an instance variable is provided to store a *reference* to an object of the referred class, `SavingsAccount`. A basic query, the *getter*, returns a reference to the object and a *setter* is introduced to store a (new) reference. Variables with reference semantics can also store the null reference. When considering associations with restricted multiplicity, 0..1 means the null reference is allowed while 1..1 means the null reference must be forbidden. In the latter case, the referring class is said to be *existential dependent* from the referred class. An object of the referring class can never exist without an object of the referred class.

2.7.1.2 Value Requirements

In the context of associations, a Value Requirement restricts the kind of objects (type) that can be referred to by objects of the referring class. At the defining level of the association, this is trivial as the return type, when well-chosen, of the basic query defines the kind of objects that are allowed. However, a subclass of the referring class may want to restrict the kind of objects it wants to be attached to a subclass of the initial referred class. The decision whether non-effective values, represented by the null reference, are allowed is also Value Requirement.

As for properties, a *Boolean inspector* is introduced to validate the Value Requirements. In the context of associations, these requirements remain state-independent. More specifically, nor the state of the referring object nor the state of the referred object may be used in this inspector. As explained in section 2.6.7, state-independent methods are moved to the static inner class. By convention, the name of the inspector checking the Value Requirement for an association between S and R, where R is the referred class, is `isProperValueForR(R r)`. When a role name is available, the name of the inspector uses it: `isProperValueForRole(R r)`. Role names are required when a class has at least two associations with the same class. In the next sections, when a class name is used in a method name, it must be in an analogous way substituted by role names.

Listing 2.15 illustrates the inspector for the association where current accounts refer to savings accounts. To explain the need for the inspector better, inheritance is considered immediately. Listing 2.15 also shows the redefinition of the inspector at the level of the subclass `JuniorCurrentAccount`: the possible values a junior current account can refer to are restricted to junior savings accounts. Thus, as explained in section 2.6.7, invoking the inspector against the right "class object" implies the right version of the inspector will be executed.

2.7.1.3 State Requirements

A State Requirement describes a constraint that restricts acceptable values of characteristics based on the status of the involved object(s). In the context of associations, two different types of State Requirements can be distinguished. The first type describes constraints between a property of one participant and a property of the other participant. The requirement that the sum of the balances of a savings account and the attached current account must always

```
1  public static class COCurrentAccount extends COBankAccount{
2
3  /**
4   * @return one of boolean
5   */
6  public boolean isProperValueForSavingsAccount(
7                                        SavingsAccount savings){
8    return true;
9  }
10
11 public static class COJuniorCurrentAccount
12                                       extends COCurrentAccount{
13 /**
14  * @return   if ((savings !=null) &&
15  *                (!savings instanceof JuniorSavingsAccount))
16  *           then result == false
17  */
18 @Override
19 public boolean isProperValueForSavingsAccount(
20                                       SavingsAccount savings){
21   return  (savings == null) ||
22           (savings instanceof JuniorSavingsAccount);
23 }
```

Listing 2.15: Value Requirement for a unidirectional association.

be at least zero is an example of such a constraint. The second type describes constraints between a property of a participant and the state of involvement in the association. The requirement that the balance of a current account is not allowed to be negative if there is no savings account attached to it is an example of this type of constraint.

With unidirectional associations, the first type can not be enforced because the referred type is unaware of the referring type. Changing the value of the property of the referred type can violate the constraint but the referred type can not check the validity of new values. If the requirements contain a constraint of this type a bidirectional association is needed. This kind of constraint will be explained in section 2.7.2.2. Both types of constraints are worked out in the same way as State Requirements for properties (see section 2.6.3). An inspector `isProper`α`r(T` α`, R r)` is introduced to check a State Requirement of the second type.

```
1  /**
2   * @return  if (! isProperValueForBalance(balance))
3   *           then result == false
4   * @return  if (! isProperValueForSavingsAccount(savings))
5   *           then result == false
6   * @return  result == matchesBalanceSavingsAccount(balance,
7   *                                                   savings)
8   */
9  public final boolean isProperBalanceSavingsAccount(
10                     BigDecimal balance, SavingsAccount savings){
11   if (! isProperValueForBalance(balance))
12     return false;
13   if (! isProperValueForSavingsAccount(savings))
14     return false;
15   return matchesBalanceSavingsAccount(balance, savings);
16 }
17
18 /**
19  * @pre isProperValueForBalance(balance)
20  * @pre isProperValueForSavingsAccount(savings)
21  * @return  if (savings != null) && (balance.signum() < 0))
22  *           then result == false
23  */
24 protected boolean matchesBalanceSavingsAccount(BigDecimal balance,
25                                           SavingsAccount savings){
26   return (savings !=null) || (balance.signum() >= 0);
27 }
```

Listing 2.16: State Requirement for a unidirectional association.

Listing 2.16 illustrates the State Requirement between the property balance and the (unidirectional) association between current accounts and savings accounts. This listing also illustrates how an extra assistant-inspector makes it possible to provide a complete definition of the inspector `isProperBalanceSavingsAccount`. This inspector is made `final` to underline that the definition is complete. Only the code to check the real problem must be written by the developer in the assistant-inspector.

2.7.1.4 Invariant

The invariant of a unidirectional association is also described by the union of all Value Requirements and State Requirements. The Boolean inspector checking whether a given value for the association meets its invariants is `canHaveAsR(R r)`, with R the name of the class or, if available, the role name. The inspector checking if the currently registered object meets its invariants, is `hasProperR(R r)`. The specification and implementation is entirely analogous with properties.

2.7.1.5 Transition Requirements

Specific requirements restricting the possible transitions for a unidirectional association are also again described by a boolean inspector, `canHaveAsNewR(R r)`. Next to the requirement for the new value to meet the invariants, additional constraint can be defined to restrict the allowed transitions. Unlike State Requirements, Transition Requirements are allowed to use the state of the referred object. A Transition Requirement could for example describe that the balance of the new savings account, if effective, must be higher than the balance of the currently referred savings account.

The basic setter `registerR(R r)`, setter `setR(R r)` and constructor are exactly the same as for properties. Appendix B shows a complete skeleton for the pattern for a unidirectional association.

2.7.2 Bidirectional Associations

A bidirectional association has two unidirectional associations as starting point. In two places, the pattern for a bidirectional association differs from the "simple sum" of two unidirectional associations. Firstly, the two unidirectional parts are not allowed to behave like unidirectional associations. They are inseparable. Secondly, to be in a steady state the two involved objects must refer to each other.

When defining a bidirectional association, some functionality belongs naturally to the association and not to one of the participants. The participants of the association are the obvious candidates. To improve the readability of the code, one participant is always chosen as the "controlling" participant. The controlling participant acts as the manager of the association. In case one participant is existential dependent of the other, i.e. the participant must at all times be attached to the other participant, that participant is preferably chosen as controlling participant. It might seem counter intuitive to choose the existential dependent participant as controlling participant. The reason for this choice is pure technical. The choice has no influence on the offered interface. Choosing a controlling participant increases readability as it prevents randomly scattering of association code between the two participants. In the example, the class representing savings accounts will be the controlling participant because it is existential dependent on the class representing current accounts.

2.7.2.1 Value Requirements

Both the participants of a bidirectional association have an inspector to check the Value Requirements of the association. Each inspector has the responsibility to check the value constraint from the class it is defined in as the referring class to the referred class. Both inspectors together describe the Value Requirements of the association as a whole.

Listing 2.17[1] illustrates the inspector checking the value constraint for the association between current and savings accounts. This inspector can be a useful tool for clients of the classes representing the current and savings accounts, but is not a necessary inspector for the remainder of the pattern. Next sections will show how both parts of the Value Requirement come in a natural way together in the necessary places.

This inspector is introduced in the "controlling class" `SavingsAccount` as a "static" method, i.e. a method in the inner class `COSavingsAccount`. The implicit class object for savings account can not be used because dynamic binding on both arguments is necessary to invoke the right version of the Value Requirements inspector of both sides. Note that if the restriction of possible participants in subclasses must redefined in a symmetric way,

[1] In the listing the specification is left out to reduce the size of the listing. This should not be interpreted as if specification is less important. In this case, specification and implementation are more or less copies. Because using code in examples is more common, the code is retained.

```
1  public final boolean isProperValueForSavingsAccount_CurrentAccount(
2                       SavingsAccount savings, CurrentAccount current){
3    if ((savings == null) && (current == null))
4      return false;
5    return  ( (savings == null) ||
6              savings.getClassObject().
7                          isProperValueForCurrentAccount(current) )
8          &&
9            ( (current == null) ||
10             current.getClassObject().
11                         isProperValueForSavingsAccount(savings) );
12 }
```

Listing 2.17: Value Requirement for the association between savings accounts and current accounts

both sides must redefine their part of restriction. In the example, junior current and junior savings accounts can only be attached to each other, in other words, the redefinition is symmetric. The Value Requirement inspector in the subclass `JuniorSavingsAccount` restricts the possible current accounts to junior current accounts. Analogously, the inspector in the subclass `JuniorCurrentAccount` restricts the possible savings accounts to junior savings accounts.

2.7.2.2 State Requirements

Associations introduce two kinds of State Requirements: the first type describes constraints between a property of one participant and a property of the other participant. The second type describes constraints between a property of a participant and the state of involvement in the association. Section 2.7.1.3 already explained the second type. Remark that requirements of this type are always worked out in the class of the property and not necessarily in the controlling class.

The requirement that the sum of the balances of a savings account and the attached current account must always be at least zero is a requirement of the first type. A boolean inspector `isProper`$\alpha\beta$`(T1` α`, T2` β`, Bar.COBar classObject)` is introduced in the

controlling class `Foo`. α is a property of the class `Foo` and β a property of class `Bar`. If using only the names of the properties is ambiguous, the name is preceded by the class name or, if available, the role name: `isProperFoo`α`Bar`β.

Each involved characteristic must meet its Value Requirement to be able to have an acceptable combination of values. Listing 2.18 illustrates the State Requirement for the bank account example. To be able to check if a balance for a savings account and a balance for a current account meet the requirement, it is necessary to say to which concrete type of savings account (or current account) the object has. So the exact question is "Do the given balance for a savings account of the given kind and the given balance for a current account of the given kind meet the requirement?" The kind of current account is given by an argument, in this example `CurrentAccount` or `JuniorCurrentAccount`, namely the class object representing one of those classes is given. The kind of savings account is given by the implicit argument, namely the class object the inspector is invoked against. By using the class object, no concrete instance of the class is needed. Given the class object for both participants, the right version of the Value Requirement can be used to check both balance values. An assistant-checker `matches`$\alpha\beta$ is introduced to check the actual problem.

One of the followed principles is the Open-Closed Principle (see section 2.4.4). Suppose the classes `BankAccount`, `SavingsAccount`, `CurrentAccount` and `JuniorSavingsAccount` are published, and therefore closed, and now the class `JuniorCurrentAccount` needs to be added with the restriction that a junior current account may only be attached to a savings account if the sum of the balances is at least 100. Because `SavingsAccount` is the controlling class, the inspector `matches`$\alpha\beta$ is in that class. Obviously, introducing an "artificial" subclass of `JuniorSavingsAccount` to redefine the inspector is not a proper solution. Therefore, a hook method is provided in the other participant: `matches`$\alpha\beta$`Helper(T1` α`, S.COS classObject, T2` β`)`.

2.7.2.3 Invariant

To meet its invariant for a bidirectional association, a class needs to meet the invariant for the unidirectional part where it is considered as the referring class. As explained in section 2.7.1.4, this is checked by the inspector `canHaveAsR(R r)`. But in the context of a bidirectional association, both unidirectional associations must be consistent to have a steady state. For a bidirectional association, an object of a class can only be in a steady state if (1) it is not attached to an object of the referred class or (2) it references an object of the referred class *and* the referenced object references the referring

```
1 public static class COSavingsAccount extends COBankAccount{
2 /**
3  * @pre classObject != null
4  */
5 public final boolean isProperSABalanceCABalance(
6                         BigDecimal SABalance,
7                         BigDecimal CABalance,
8                         CurrentAccount.COCurrentAccount classObject){
9   if (!isProperValueForBalance(SABalance))
10    return false;
11  if (!classObject.isProperValueForBalance(CABalance))
12    return false;
13  return  matchesSABalanceCABalance(SABalance,CABalance,
14                                              classObject)
15      &&  classObject.matchesSABalanceCABalanceHelper(
16                                              SABalance,this,CABalance);
17 }
18
19 /**
20  * @pre classObject != null
21  * @pre isProperValueForBalance(SABalance)
22  * @pre classObject.isProperValueForBalance(CABalance)
23  * @return  if (currentBalance.add(savingsBalance).signum() < 0)
24  *          then result == false
25  */
26 protected boolean matchesSABalanceCABalance(
27                         BigDecimal SABalance,
28                         BigDecimal CABalance,
29                         CurrentAccount.COCurrentAccount classObject){
30  return (SABalance.add(CABalance).signum() >= 0);
31 }
32 }
```

Listing 2.18: State Requirement for the association between savings accounts and current accounts

object back. This has to be repeated in both sides to enforce the invariants concerning the bidirectional association are met by both participants. Listing 2.19 illustrates the invariant concerning the bidirectional association for the class `CurrentAccount`.

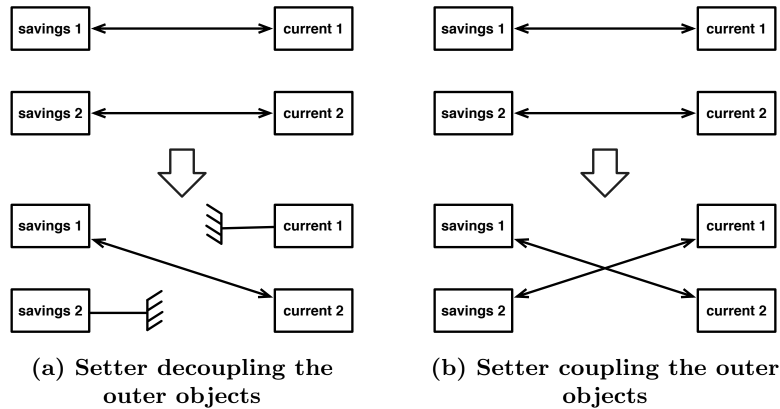
```
1 public final boolean hasProperSavingsAccount(){
2   return  canHaveAsSavingsAccount(getSavingsAccount()) &&
3           ( (getSavingsAccount()==null) ||
4             (getSavingsAccount().getCurrentAccount()==this) );
5 }
```

Listing 2.19: Invariant for the association between savings accounts and current accounts in class `CurrentAccount`

2.7.2.4 Transition Requirements

Until now, the bidirectional association was considered as the ”sum” of two unidirectional associations. These unidirectional associations are however not allowed to behave as such, they should behave as one atomic whole. A logic consequence is that there are no longer setters available to change only one reference. The setter is obliged to set a consistent binding in two directions for all involved objects. When setting a binding for a bidirectional association with restricted multiplicity up to four objects can be involved. The pattern provides two generic setters with a different strategy to leave all objects in a steady state. The two different strategies are illustrated in figure 2.2. The setter wants to attach `savings 1` and `current 2` to each other. Obviously, `savings 2` and `current 1` must also be left in a steady state. The first option, illustrated in figure 2.2a, leaves them unattached. However, this option is only available if no participant is existential dependent to the other. In the example, objects of the class `SavingsAccount` must at all times be attached to an object of the class `CurrentAccount`. Thus, when there is existential dependency the second option, illustrated in figure 2.2b, must be used. This setter attaches the “outer” objects to each other. Without existential dependency both setters are available. While these two setters will probably cover most situations, the developer of the bidirectional association can still elaborate another custom setter.

Both setters are in fact composite setters: both *set* four references which can be compared to setting four unidirectional associations. When defining a



(a) Setter decoupling the outer objects

(b) Setter coupling the outer objects

Figure 2.2: The two generic setters for associations with restricted multiplicity

bidirectional association, both participants define their own boolean inspector `canHaveAsNewR(R r)` to check the Transition Requirements seen from their own perspective. So, this inspector is written as if the association is unidirectional with the containing class as referring class. As explained in section 2.7.1.5, these inspectors can use characteristics of both participants. These inspectors are the building blocks for the transition setter that needs to be build for each generic setter. Listing 2.20 illustrates the setter that attaches the outer objects to each other. Listing 2.21 illustrates the new inspector checking all four "reference" changes. The code in the listing is written as if the association has multiplicity 0..1 - 0..1. In the example, a savings account must at all times be attached to a current account. This makes it possible to remove the unnecessary null-checks on lines 9 and 14 in listing 2.20 and lines 6 and 11 in listing 2.21. This setter and inspector are written in the controlling class.

The setter uses the basic setters `registerR(R r)`. These basic setters can not be used outside the generic setters or the constructor. The applicability of the basic setter in the non-controlling class `CurrentAccount` can be strongly restricted by adding some preconditions to the method. This basic setter is illustrated in listing 2.22. If the basic setter is used to set a reference to an effective savings account, a precondition requires that the reference in the opposite direction, from the given savings account to the implicit argument of the method, is already set (line 2). If the basic setter is used to detach the

```
1  public final void setCurrentAccount(CurrentAccount current)
2                                   throws IllegalArgumentException{
3    if (!canHaveAsNewCurrentAccountSavingsAccount(current))
4      throw new IllegalArgumentException();
5    CurrentAccount oldCurrentFromThis = getCurrentAccount();
6    SavingsAccount oldSavingsFromArgument =
7                      current!=null?current.getSavingsAccount():null;
8    registerCurrentAccount(current);
9    if (current != null)
10     current.registerSavingsAccount(this);
11   if (oldSavingsFromArgument != null)
12     oldSavingsFromArgument.
13                     registerCurrentAccount(oldCurrentFromThis);
14   if (oldCurrentFromThis != null)
15     oldCurrentFromThis.
16                     registerSavingsAccount(oldSavingsFromArgument);
17 }
```

Listing 2.20: Setter in the controlling class `SavingsAccount` (see figure 2.2b)

implicit argument from its savings account, a precondition requires that the reference in the opposite direction is already removed (line 4).

In the non-controlling class, an inspector and setter with the same semantics can be provided. These methods are easier as they can be defined in terms of the methods in the controlling class. Listing 2.23 illustrates these methods.

By describing the Transition Requirements in both sides from their own perspective, duplication in conditions can arise. For example, suppose current accounts can be blocked and when blocked their binding with the savings account is not allowed to change or a bank in the world Utopia could require that the sum of the balances of the newly attached current and savings account is at least 5000. A Transition Requirement will however not by definition be duplicated, the same bank in the world Utopia could also require that if a current account is going to be attached to a savings account, the balance of that new savings account must at least be 2500. As showed previously in similar situations, duplication can be solved by introducing an assistant-inspector. This inspector can be put in the controlling class and a helper

```
1  public final boolean canHaveAsNewCurrentAccountSavingsAccount(
2                                              CurrentAccount current){
3    //Associaton 1: this-current
4    if (!canHaveAsNewCurrentAccount(current))
5      return false;
6    if (  (current != null) &&
7          (!current.canHaveAsNewSavingsAccount(this))  )
8      return false;
9
10   //Association 2: current.getSavingsAccount()-getCurrentAccount()
11   if (current != null){
12     //if current == null, there is no second association
13     if (  (current.hasSavingsAccount()) &&
14           (!current.getSavingsAccount().
15             canHaveAsNewCurrentAccount(getCurrentAccount()))  )
16       return false;
17     if (  (getCurrentAccount()!=null) && (!getCurrentAccount().
18           canHaveAsNewSavingsAccount(current.getSavingsAccount()))  )
19       return false;
20     }
21   return true;
22 }
```

Listing 2.21: Transition Requirement in the controlling class `SavingsAccount` (see figure 2.2b)

method can again be put in the other participant.

Listing 2.24 illustrates the Transition Requirement that is used by the setter that disconnects the “outer” objects. The setter is very similar to the one in listing 2.20, except for lines 13 and 16 where the argument of both basic setters is replaced by `null`.

```
1  protected void registerSavingsAccount(SavingsAccount savings){
2    assert savings == null ||
3           savings.getCurrentAccount() == this;
4    assert savings !=null ||
5           !this.hasSavingsAccount() ||
6           getSavingsAccount().getCurrentAccount()!=this;
7    if (! getClassObject().isProperValueForSavingsAccount(savings))
8      throw new IllegalArgumentException();
9    this.savings = savings;
10 }
```

Listing 2.22: Basic setter for a bidirectional association in the non-controlling class

```
1  public final boolean canHaveAsNewCurrentAccountSavingsAccount(
2                                        SavingsAccount savings){
3    if (savings!=null)
4      return savings.canHaveAsNewCurrentAccountSavingsAccount(this);
5    else
6      if (getSavingsAccount() != null)
7        return getSavingsAccount().
8                 canHaveAsNewCurrentAccountSavingsAccount(null);
9      else
10       // This object is the only one involved
11       return canHaveAsNewSavingsAccount(null);
12 }
13
14 public final void setSavingsAccount(SavingsAccount savings)
15                               throws IllegalArgumentException{
16   if (!canHaveAsNewCurrentAccountSavingsAccount(savings))
17     throw new IllegalArgumentException();
18   if (savings != null)
19     savings.setCurrentAccount(this);
20   else
21     if (getSavingsAccount() != null)
22       getSavingsAccount().setCurrentAccount(null);
23 }
```

Listing 2.23: Transition Requirement and Setter in the class `CurrentAccount` (see figure 2.2b)

```
public final boolean canHaveAsNewCurrentAccountSavingsAccount(
                                        CurrentAccount current){
  //Association 1: this-current
  if (!canHaveAsNewCurrentAccount(current))
    return false;
  if (   (current != null) &&
         (!current.canHaveAsNewSavingsAccount(this))  )
    return false;

  //Association 2: current.getSavingsAccount()-NULL
  if ((current!=null) && (current.hasSavingsAccount()))
    if (!current.getSavingsAccount().
                              canHaveAsNewCurrentAccount(null))
      return false;

  //Association 3: getCurrentAccount()-NULL
  if (getCurrentAccount()!=null)
    if (!getCurrentAccount().canHaveAsNewSavingsAccount(null))
      return false;
  return true;
}
```

Listing 2.24: Transition Requirement in the controlling class `SavingsAccount` (see figure 2.2a)

2.7.2.5 Language Construct

For associations, a similar extension as for properties is proposed. Listings 2.25 and 2.26 illustrate how the example is worked out with the new construct `BiDirAssociation`. If an association is bidirectional, two classes are involved and they both introduced their "view" of the association. By definition, these views must be mirror images of each other as they represent an inseparable whole. The part of the code in the other side representing the same association is identified by the keyword `mirroredBy` followed by the name of the association. Obviously, both parts must be defined as bidrectional. The `@Value`, `@State` and `@Trans` assertions have exactly the same semantics as with properties.

```
1 public class CurrentAccount{
2 /**
3  * The balance of this bank account
4  * @State savings.PositiveBalanceIfNoSavings
5  * @State CurrentAccount.current.PositiveSumOfBalances
6  */
7 @Override
8 Property BigDecimal balance;
9
10 /**
11  * The blocked state of this current account
12  */
13 Property boolean blocked;
14
15 /**
16  * The savings account of this current account
17  * @State PositiveBalanceIfNoSavings
18  *        if (savings == null)
19  *        then balance.signum() >= 0
20  * @State CurrentAccount.current.PositiveSumOfBalances
21  * @Trans CurrentAccountNotBlocked
22  *        ! blocked
23  */
24 BiDirAssociation SavingsAccount savings mirroredBy current;
25 }
```

Listing 2.25: The association between current and savings accounts (1/2)

```
1 public class SavingsAccount{
2
3 /**
4  * The credit limit of this bank account
5  * @Value creditLimit.signum() == 0
6  */
7 @Override
8 Property BigDecimal creditLimit;
9
10 /**
11  * The balance of this bank account
12  * @State CurrentAccount.current.PositiveSumOfBalances
13  */
14 @Override
15 Property BigDecimal balance;
16
17 /**
18  * The current account of this savings account
19  * @Value current != null
20  * @State PositiveSumOfBalances
21  *        balance.add(current.balance).signum() >= 0
22  * @Trans SavingsAccount.current.CurrentAccountNotBlocked
23  *
24  */
25 BiDirAssociation CurrentAccount current mirroredBy savings;
26 }
```

Listing 2.26: The association between current and savings accounts (2/2)

2.8 Another Approach?

In this chapter, a family of patterns is presented to tackle the problem of implementing properties (or associations) with their requirements. The number of needed methods are in some sense an indication of the complexity of the problem. The purpose of the patterns is to make sure objects are in a steady state (meet their invariants). Except in an “intermediate” state (half way through the change of an association between two objects), an object should never violate its invariants. The patterns provide an inspector, `hasProper`α`()`, to check the invariants.

To enforce the invariants, each mutator checks through the offered inspectors in advance if all invariants of all involved objects *in their new state* will be met and the mutator must react appropriately to the results of the check (for example leave the state unchanged and throw an exception).

The patterns only provide a mutator to change each property (or association) in isolation, therefore there is only one involved object (at least two and a maximum of four for associations) which simplifies the check. But most software systems need to offer more complex mutators that change multiple characteristics in an *atomic* way. For each such complex mutator, the developer has to write a *new* custom inspector that checks if the invariants of all involved objects will be met *in their new state*. Although the pattern is built in such a way that it offers all needed building stones (for example `isProper`$\alpha\beta$`(T `α`, S`β`)`), the development of such an inspector is not straightforward.

However, we believe that the complexity can be reduced dramatically if the invariants can be checked afterwards. An essential element to be able to develop such an approach, is that *transaction support* is needed. Figure 2.3 presents the general structure of mutators in this approach. The development of the inspectors checking the value and State Requirements remains unchanged. The inspector checking the Transition Requirements however must not check the invariants anymore. Listing 2.27 illustrates the adapted inspector: the code that is removed is showed in comment. The implementation of the mutators has to be extended with the use of transactions [52, 86]. This alternative approach becomes really valuable if transaction mechanisms are offered by the programming language as First-Class Concepts as proposed by Boydens [58].

```
1   public boolean canHaveAsNewα(T α){
2 //      if (!canHaveAsα(α))
3 //         return false;
4      return ...;
5   }
```

Listing 2.27: Transition Requirement without invariants

This alternative approach, however, can not be generally applied. In situations where a rollback to the original situation is impossible, the invariants must be checked before the actual changes are applied. Suppose for example an application controlling an ATM. The signal to give money to the client must not be sent to the machine before it has been checked wether the given bank account code is valid and the balance of the bank account doesn't drop below the credit limit. Of course one can always create an artificial intermediate state which is obviously not an elegant solution.

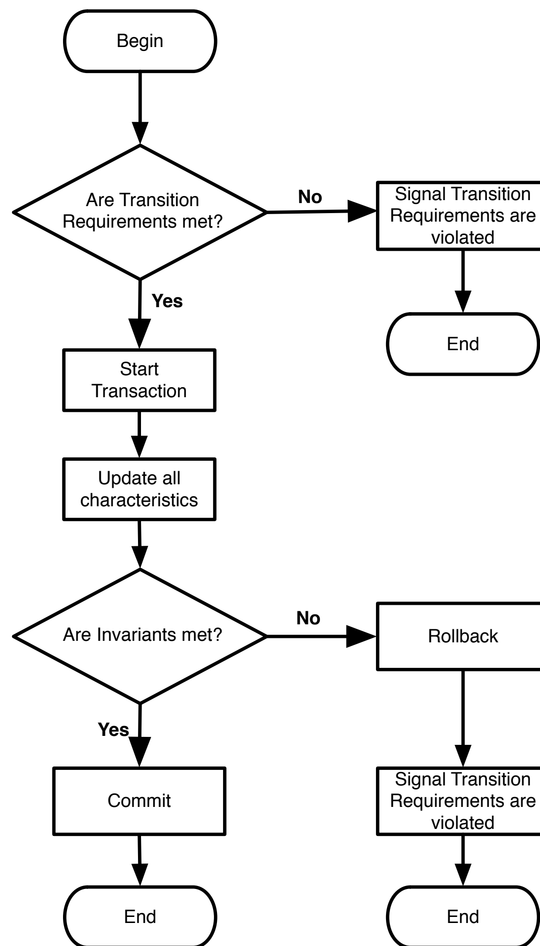


Figure 2.3: Another approach to develop complex mutators.

2.9 Conclusion

Although properties and associations with their inherent requirements are ubiquitous in software applications, no standardized solution exists to implement them. In this chapter, we developed a family of patterns to implement characteristics with their requirements. We started with outlining some important principles for implementing software systems. These principles steered the development of the patterns.

First, we have identified three types of requirements: Value Requirements, State Requirements, and Transition Requirements. *Value requirements* are used to specify the most basic kind of business rules in that they restrict the range of values that a characteristic, property or association, can have. A *State Requirement* describes a constraint that restricts acceptable value combinations of (a set of) characteristics. *Transition requirements*, then, specify the business rules that restrict the evolution of values. The combination of all Value Requirements and State Requirements define *the invariants* of a class. Transition requirements correspond to *preconditions* of mutators of a class. Design by Contract [104, 106] only offers class invariants and preconditions to impose restrictions. Because Design by Contract does not offer any other concepts apart from preconditions and class invariants to capture requirements, we are convinced that any requirement is either a Value Requirement, State Requirement or Transition Requirement.

With this taxonomy in hand, we started developing the pattern. The basic idea was to separate the concerns of checking the validity of values for characteristics from the concerns of the actual implementation of the change. All requirements are encapsulated in a number of *inspectors*. The Command-Query Separation Principle guarantees these inspectors can harmlessly be called as often as needed.

The inspectors used to check Value Requirements and State Requirements can perform their checks independent of the current state of an object and are therefore class methods. However, when the specification is non-deterministic a subclass is allowed to redefine the specification according to the rules described by the Liskov Substitution Principle. In Java, class methods (`static` methods) can not be *overriden*, hence a solution with a *static inner class* is worked out that enables overriding the method definition and still makes it possible to call the inspector without needing an object of the class. The patterns define how all inspectors and *basic mutators* collaborate.

The patterns for *associations* introduced some specific challenges because two objects are involved. The Open-Closed Principle caused the addition of

some *helper-methods*: inspectors checking requirements that involve characteristics of both classes actually should have two *implicit* arguments. Java forces us to choose a class to put the inspector in. The patterns advise to choose a *controlling class* to avoid the unnecessary distribution of methods over multiple classes. When a new association is set up between two objects, possibly two more objects are involved. The patterns provide two different mutators to make sure all involved objects are left in a steady state afterwards.

Experiments with the patterns in small academic problems show that about 70% of the code of the patterns is boilerplate code. Because the programmer only has to *rewrite and not rethink* this code, the risk for bugs is reduced. As such, the programmer can focus on the heart of the business rules without having to think about the technical part of the implementation. In addition, the patterns are such that they support adaptability in both dimensions, space and time.

Finally, some ideas are presented to show how the patterns can be replaced by language constructs. In the proposed idea, the specification is no longer a *simple comment*. The different types of requirements that we have identified for properties and associations now become an integral part of the definition of the characteristic.

Wat je streven ook is, welke droom je ook najaagt, welke verandering je ook verlangt, vergeet niet : het is al begonnen. Het verlangen als zodanig is de eerste stap op weg naar verwezenlijking.
– Daphne Rose Kingma

Chapter 3

Concepts for Abstracting away Object Reification at the level of Platform Independent Models (PIMs)

Chapter Summary

Conceptual models introduce accidental complexity when they contain technical aspects in order to describe real-world facts. Such complexity is introduced by enforcing ("locking in") decisions that should have been made in a later activity in the software development process. UML and OCL lack expressive constructs to reason about *event occurrences*, even more so when the *historical aspect* of such occurrences becomes important. This work presents a new operator, the #-operator, that allows analysts to treat events as first-class citizens. By assigning a property, that represents the execution time, to events, it becomes possible to model historical event information without the need to introduce irrelevant facts in the conceptual model.

3.1 Preamble

The research of this chapter started in November 2004 in the context of the FWO-funded[1] project "*Formal support for the transformation of software models*". Partners of this project are the KULeuven and University of Antwerp. On the KULeuven side, research is done related to the question "Which concepts are needed to describe behavior of a software system on a platform independent way?". Research to find answers to the question "How can transformations between models be described and managed?" is performed at the University of Antwerp.

At the time this research started, Model Driven Architecture (MDA) was a very popular research topic in the search for good software development processes. The Unified Modeling Language (UML) is an important part of MDA. UML, an OMG standard, became the standard notation for analysis and design of software systems. The goal of MDA is to transform Platform Independent Models to Platform Specific Models and Platform Specific Models to code in a (semi-)automated way.

With regard to the software development process, this research is situated in the analysis activity and/or at first stages of the design activity. In MDA terminology, the result of the analysis activity is a Computational Independent Model (CIM). The higher level design models are Platform Independent Models (PIM). It is not clear what exactly the difference is between the CIM and the highest level PIM, if there is any. Therefore and because the notion CIM isn't often used in literature, we use the term PIM in the remainder of this chapter. The highest level PIM corresponds to the conceptual model.

UML mainly focuses on structural aspects in models. Modeling behavior gets less attention and therefore UML lacks high-level constructs to describe behavior. The Object Constraint Language (OCL) is used to describe the semantics of behavior. In our research, we promote events to first-class citizens. A new operator, the #-operator, is introduced to reason about events. Through the use of the operator all occurrences of an event against an object can be retrieved. This collection can for example be used to specify business rules. The first results of this research are presented in March 2006 at The Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software [74]. This paper served as the basis for sections 3.4 to 3.6. The remainder of the chapter presents the further research done, namely the introduction of the @-operator. With this operator, the value of

[1] Fund for Scientic Research (FWO) in Flanders.

a property at a given time can be retrieved. Ideas about *abstract method* (similar to an abstract class, commonalities between events can be described in a parent method) are not elaborated. The research regarding this chapter stopped end 2006. The fact that OCL now contains the ^^-operator, which is very similar to our #-operator, proves there was a need to treat events as first-class citizens.

Today, software systems are more and more developed in an agile matter. Although OCL is a standardized language and in a standardized way linked with UML, it is not very often used. UML is used for informal descriptions of the software system. The description of the semantics is postponed to the implementation phase.

3.2 Introduction

The object-oriented software development process is composed of a number of subprocesses, namely analysis, design, implementation and verification. The major purpose of the *analysis* process is to establish the requirements for the software system to be developed. One of the results of the analysis process is a conceptual model describing the problem domain in all its facets. The conceptual model will therefore be as close as possible to the facts, as they are observed in the external world. In the context of Model Driven Architecture (MDA) [82, 94] the result of the analysis activities is a *base* Platform Independent Model (PIM). The base PIM expresses business rules and functionality undistorted, as much as possible, by technology [125].

During the object-oriented design process the conceptual model, or base PIM, is transformed into an operational model, or a Platform Specific Model (PSM). Software quality factors and other non-functional business requirements will influence design decisions [49]. In general, an MDA model will have multiple levels of PIMs. Each PIM except the base model includes some platform independent technical decisions. Transforming a PIM to a PIM at the next level adds more technical information and makes it in this way possible to map it more precisely to a PSM in the next step. For example, persistence or security are aspects that can be added to a second-level PIM [125]. In the MDA these transformations will be automated as much as possible, needing some PSM creators to write transformation definitions [94, p.147].

Object-orientation has its roots in the *programming languages* Smalltalk 80 [85] and Simula 67 [68]. In the nineties, the object-oriented approach to software development has been generalized from programming to object-

oriented design and object-oriented analysis. This explains why, in the current state-of-the-art, many conceptual models contain technical decisions, or in other words design decisions. Indeed, in modeling artefacts from the external world, conceptual modelers are often forced to decide *how* artefacts must be supported in the ultimate software system. We believe that the sole purpose of analysis is to describe *what* the software system is all about. The purpose of design is then to decide *how* elements of conceptual models are to be realized in the best possible way. We therefore claim that concepts offered to model at the higher levels of abstraction (the base PIM) must be extended or redefined.

This chapter starts from the observation that modeling facts with regard to the history of events is not straightforward. A way to solve this problem is to model events as objects. This technique is referred to as reification [48]. Information about the history can now be retrieved through the use of queries. However reification is rather about *how* to represent an observed fact than about modeling the observed fact. While the fact was first observed as an event, it is still represented by an object because some properties of the fact could otherwise not be modeled. This chapter proposes an extension of the Unified Modeling Language (UML) [121] and the Object Constraint Language (OCL) [12] to prevent the need of reification in base PIMs. We further show how these base PIMs can be transformed in second-level PIMs.

Overview

The remainder of the chapter is structured in the following way: section 3.3 presents some key principles for conceptual modeling. Section 3.4 describes two different modeling techniques that are often used to model behavior, namely a technique using properties and a technique using reification. To improve the technique of reification we introduce a new UML concept. We show that both techniques do not have the desired level of abstraction. Therefore we introduce a new OCL concept to model some aspects concerning behavior at the base PIM level in section 3.5. Section 3.6 describes how base PIMs using this concept can be transformed into two different second-level PIMs (or PSMs). In section 3.7, we investigate the challenges introduced by objects as arguments of events. Depending on the context, the same observed fact is modeled in the one conceptual model as an object while in the other conceptual model as an event. In section 3.8, a principle that guides the analyst in his decision making is defined. Finally, we conclude in section 3.9.

3.3 Principles for Conceptual Modeling

In this section, we present key principles for conceptual modeling that have been proposed by Devos [75] and Van Baelen [134] in related research. In general, they found these principles to be "quality criteria" of modeling methodologies: the quality is higher if (1) they obey more principles, and (2) a principle is satisfied in a broader set of scenarios/situations. Today, no method satisfies all principles in all situations, and probably no method will ever satisfy all principles at the same time. The purpose of these principles is to improve current methods to come closer to this ideal. The principles will be used to evaluate the conceptual models in the next sections.

3.3.1 Principle of Uniqueness

Devos [75] and Van Baelen [134] define the *Principle of Uniqueness*, also known as *the Principle of No-Choice*, as follows: a given set of real-world facts can only be represented by exactly one conceptual model, except for trivial isomorphisms. There should not exist equivalent alternatives for modeling a given set of real-world facts. An analysis method must offer the analyst a palette of modeling concepts and guidelines such that he does not have to spend time and energy on the question *how* to model certain real-world facts. The task of collecting, understanding and representing the relevant real-world facts is already hard enough and deserves the full attention of the analyst. Avoiding (almost) equivalent models from the same set of real-world facts, also prevents the need for *interpretation* of the conceptual model in the next phases of software development. A method must investigate and evaluate all possible alternative concepts and make sure a well-balanced set of concepts and guidelines at the right level of abstraction is offered.

This unique representation mapping must be true in both directions: a given set of relevant real-world facts can only be represented by one conceptual model, but a given conceptual model can also only represent one unique set of real-world facts. Van Baelen identifies this observation as a distinct principle: *the Principle of Unambiguity* [134, 87]. In this dissertation, we don't rely on the principle of Unambiguity.

3.3.2 Principle of No Redundancy

Devos [75] and Van Baelen [134] define the *Principle of No Redundancy* as follows: a real-world fact should only be represented once and at one place in

the conceptual model. It should be very clear which modeling concept reflects a real-world fact. This traceability must be present in both directions: it must be clear how to map a real-world fact to the conceptual model and it must also be easy to trace back from a conceptual model fact to the real-world fact. This promotes the adaptability of the conceptual model because when a real-world fact changes, it is obvious which parts of the conceptual model need to be adapted. It also avoids unnecessary complexity because the analyst does not need to be concerned about the consistency of the conceptual model. Similar to the principle of uniqueness, this eases the task of the developers in the next phases because they don't need to search the conceptual model for multiple concepts that actually represent only one real-world fact.

3.3.3 Principle of Completeness

Devos [75] and Van Baelen [134] define the *Principle of Completeness* [147] as follows: *all* relevant real-world facts must be represented in the conceptual model. Developers in the next phases of the software development process should never be forced to focus on identifying missing real-world facts or business rules. If some facts are not present, the conceptual model is considered incomplete. This will almost certainly lead to errors or arbitrary decisions in later phases and eventually result in a software system that does not correspond to the initial set of real-world facts and requirements. Developers in later phases of the software process should never be required to interpret or complement a conceptual model. If vague or ambiguous parts of the conceptual model are observed in later phases, the analyst should be asked to clarify or complete the conceptual model. This should not be confused with changing requirements or missing information. It is of utmost importance that the stakeholders provide the analysts with *all* relevant facts. Imprecise, incomplete and non-exhaustive information can lead to serious problems and failures in the system being developed. Changing or new requirements lead to another set of relevant real-world facts and by consequence to an adapted conceptual model.

Construct deficit, defined by Wand as the given that a real-world fact can not be represented by any modeling construct [142], implies the impossibility to always deliver a complete conceptual model.

3.3.4 Principle of Preciseness

Devos [75] and Van Baelen [134] define the *Principle of Preciseness* as follows: all relevant real-world facts must be modeled in a formal way [69]. Natural language elements should only be added to further clarify parts of the conceptual model. These textual additions must always be able to be removed without removing any representation of a real-world fact. Representation in natural language often leaves room for interpretation and is by consequence by definition ambiguous.

3.3.5 Principle of Minimalism

Devos [75] and Van Baelen [134] define the *Principle of Minimalism* as follows: only the relevant real-world facts must be represented in the conceptual model. A conceptual model should not contain elements that are not relevant with respect to the requirements. Extra elements are considered as noise and add unnecessary complexity to the conceptual model. One of the goals of conceptual modeling is to bridge the gap between the complexity of the real world and the aimed simplicity of the software system. Therefore, it is very important for the analyst to guard the boundaries of the problem domain. Although it is often the habit of analysts to anticipate future extension of the problem domain through modeling irrelevant real-world facts, they should not. This habit mostly only results in oversized software systems. This is also postulated by the agile software development community and expressed in the Agile Manifesto [47, 102] as the principle of "*Simplicity is Essential*" [134]. In [46], Beck writes: "Extreme Programming is making a bet. It is betting that it is better to do a simple thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway." The idea is: model only what is in the requirements. Don't develop a framework for the future, you might never need it. It only adds complexity to the model.

On the other hand, the modeling concepts should not force the analyst to model or express facts they don't want to. For example, if the problem domain does not require to specify the upper bound multiplicity of an association end, the analyst should not be forced to provide an upper bound only because the model construct requires an upper bound.

Hayes et al. [87] refer to this principle when they claim "notations should be *abstract*".

3.3.6 Principle of No History

Devos [75] and Van Baelen [134] define the *Principle of No History* as follows: a conceptual model must be independent of the order in which real-world facts are added. In other words, if the relevant real-world facts were added in a different order the same conceptual model must be obtained. The history of discovering and adding real-world facts to the model is totally irrelevant for the later phases in the development process. The *Principle of Uniqueness* implies this Principle, because if a given set of real-world facts can only be represented by a single conceptual model, it can by definition not contain any modeling order information. However, no method accomplishes the Principle of Uniqueness, often quite some alternatives are possible to model the same real-world fact. As long as this is true, the Principle of No History is an important principle on its own. The order of adding real-world facts can for example be the cause of unnecessary complexity. It is possible that the analyst has to reconsider earlier modeling decisions to obtain the "best possible" model given the set of real-world facts.

3.4 Platform Independent Models

In the conceptual model, or base PIM, we strive for a high level of abstraction. In this section we show through an example that a problem domain with the need to reason about (occurrences of) an event can be modeled in two different ways using the concepts of UML2.0. We show that both models have their drawbacks resulting from the fact that they do not reach the desired level of abstraction.

3.4.1 Running Example

The example that will be used throughout this chapter concerns a small part of a library system. We only consider the concept `book`. In this example, we assume that a book has no relevant properties. We model one event, expressing that books can be loaned for a certain period (`loan(period)`). The argument `period` refers to the actual loan period. We further model three queries to ask (i) how many times a book is loaned (`nbLoans()`), (ii) when the last loan for a book started (`latestLoan()`) and (iii) the average[2] loan period of a book (`avgLoanPeriod()`). For simplicity, the examples in this chapter will

[2] We choose to define the average of an empty set as 0. This means that the average loan period of a book that never has been loaned, is defined as the period with duration 0.

not model constraints, such as the constraint stating that different loans of the same book are not allowed to overlap.

In the next sections, this example is first developed with two existing modeling techniques: in section 3.4.2 the example is worked out with properties. Section 3.4.3 uses reification to model the example. Thereafter, the model is developed with the proposed extension in section 3.5.

3.4.2 PIM Using Properties

Figure 3.1 illustrates a possible conceptual model for the example. The main issue is that we use *properties* to model the problem domain. Besides the event and the queries, three properties are introduced to calculate the result of the queries.

Book
nbLoans:NATURAL latestLoan:DATE totalLoanPeriod:DURATION
loan(period:DURATION):void nbLoans():NATURAL latestLoan():DATE avgLoanPeriod():DURATION

```
Context Book::loan(period:DURATION) : void
  post: self.nbLoans = self.nbLoans@pre + 1
  post: self.latestLoan = now
  post: self.totalLoanPeriod =
          self.totalLoanPeriod@pre + period

Context Book::nbLoans() : NATURAL
  post: result = self.nbLoans
```

```
Context Book::latestLoan() : DATE
  pre: self.nbLoans > 0
  post: result = self.latestLoan

Context Book::avgLoanPeriod() : DURATION
  post: result =
          if self.nbLoans = 0
          then 0
          else self.totalLoanPeriod/self.nbLoans
```

Figure 3.1: PIM using Properties.

The query `nbLoans()` uses the property `nbLoans`. A postcondition is added to the event `loan(period)` to specify the appropriate update of the property after each occurrence.

Analogously, the query `latestLoan()` uses the property `latestLoan`. A second postcondition is added to the event `loan(period)` to specify the ap-

propriate update of this property after each occurrence. The precondition of this query (this query is only meaningful after the first loan of the book) is specified through the property `nbLoans`.

The result of the query `avgLoanPeriod()` cannot be defined using a single property that is updated after each occurrence of the event `loan(period)`. Indeed, to specify the average of a collection of periods we need to know the sum of all periods (`totalLoanPeriod`) and the number of periods (`nbLoans`) in that collection.

Evaluation. An advantage of the approach using properties is that the resulting conceptual model is compact: we do not introduce extra concepts (classes). However, while only the average loan period is of interest, it is necessary to model the *total loan period*. In other words, artefacts that are not directly relevant need to be modeled. Moreover, each time a new property is introduced a postcondition needs to be added to the appropriate event which leads to a bad maintainability of the conceptual model. In other words, the conceptual model is not easily adaptable nor extendible. The complexity of a class grows as the number of properties grows. At a certain point, a conceptual model using reification is probably a better solution. The next section presents the conceptual model using reification.

There are several (equivalent) alternatives to this model using the same technique, namely using properties to *store* (history) information. Another analyst could have chosen to introduce a property `avgLoanPeriod`, and to recalculate the value with each occurrence of the event `loan`. Yet another analyst could have opted to store all loan periods in a multivalued property and calculate the result of the queries `nbLoans` and `avgLoanPeriod` with this new property. This clearly means that there is more than one conceptual model to represent the given set of real-world facts. Moreover, no real guidelines are available to the analyst to help him decide which model needs to be delivered. Thus, the Principle of Uniqueness is certainly violated.

The Principle of No Redundancy is also violated. Consider for example the query `latestLoan`, the definition of that query is distributed over three places: it uses the property `nbLoans` and the specification of the event `loan` must also be adapted appropriately. This introduces complexity and the need for the analyst to make sure all elements stay consistent. Suppose the requirements change and the query `latestLoan` is no longer needed, then the analyst has not only to remove the query but he is also forced to check if each of the used properties is still needed in the model.

This approach also violates the Principle of Minimalism. The analyst has

introduced a property `totalLoanPeriod` while this property is irrelevant with respect to the requirements[3]. Queries related to history of events will often require to *store* information in a property that is not directly required.

The Principle of No History is in this small example not violated, but using properties in the context of history can easily introduce history or order in the conceptual model. Suppose the query `totalLoanPeriod()` was also required, then both this query and `avgLoanPeriod()` can be written in terms of the properties `totalLoanPeriod` and `nbLoans`. But they can also both be written in terms of the properties `avgLoanPeriod` and `nbLoans`. It's reasonable to assume that the order of adding will influence the choice.

3.4.3 PIM Using Reification

In this section a conceptual model using reification is presented. In paragraph 3.4.3.1 the conceptual model is developed using UML2.0. We identify some technical problems in that solution which we try to solve by introducing a new UML concept in paragraph 3.4.3.2.

3.4.3.1 PIM With UML2.0

Figure 3.2 illustrates an alternative conceptual model for the example using reification. The model introduces the class `Loan` as a reification of the event `loan(period)` in the class of books. Each time the event `loan(period)` occurs an object of the class `Loan` must be created.

The query `nbLoans()` counts the objects of the class `Loan` associated with the book at stake. The query `latestLoan()` gives the most recent `creationTime`[4]. The specification of the query `avgLoanPeriod()` is also straightforward. However, figure 3.2 contains even more specification. This specification is necessary because we need to specify the one-to-one mapping between the objects of the class `Loan` and the occurrences of the event `loan(period)`. We also need to specify that the value of the argument `period` must be the same as the value of the property `period` in the class `Loan`. Moreover, the value of this property must never change. The postcondition of the event `loan(period)` states that there must be a new `Loan`-object with a correct value for the property `period` after an occurrence of the event. On the

[3] In this example, the analyst could also have decided to introduce the property `avgLoanPeriod` to avoid irrelevant properties, but this option is only available because the query `nbLoans()` happens to be required too.

[4] We assume the presence of the property `creationTime` for every object. The time at which an object has come into existence is registered in this property.

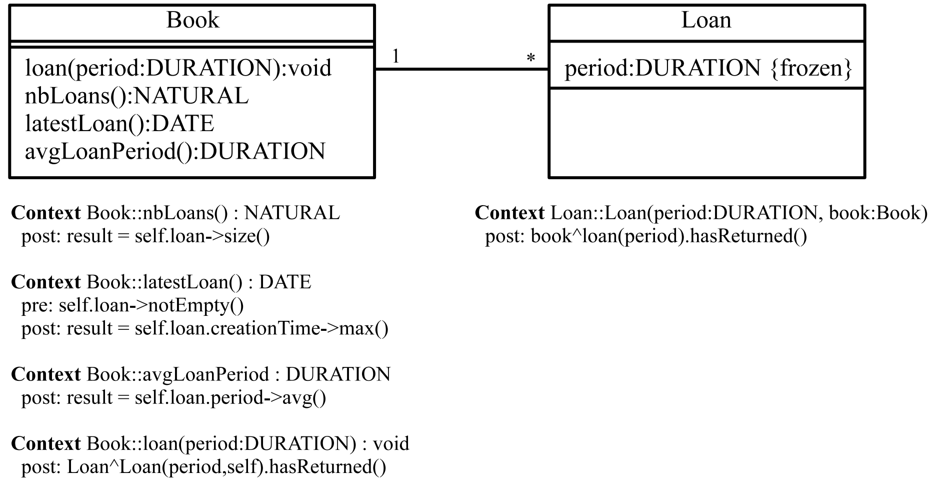


Figure 3.2: PIM using Reification.

other hand the postcondition of the constructor `Loan(period,book)` states that there must have been an occurrence of the event `loan(period)`. In OCL 1.4.2 [13, Sect.8], it was not possible to write the above specification, this problem is solved in OCL 2.0 [12]. We should also specify that the destruction time of each `Loan`-object must be the same as the destruction time of the `Book`-object it is associated with. This is not possible in OCL 2.0. To solve this problem and to avoid the need for the above "technical" specification (representing the pattern to model reification) we propose a new UML concept introduced in the next paragraph.

3.4.3.2 Method Class

Figure 3.3 illustrates the new UML concept. The concept is used to express a tight bond between a class, referred to as a *Method Class*, and an operation of another class. The concept can be compared with association classes in UML. Association classes reify associations between two classes; method classes reify operations of classes.

The semantics of a method class is as follows. For each occurrence of `event()` of the class `Class` there must be exactly one object of the class `Event` and, the other way around, for each object of the class `Event` there must be exactly one occurrence of `event()`. Every parameter of an event has a corresponding immutable property in the method class. The same is true

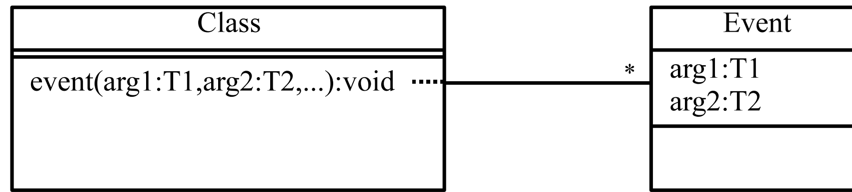
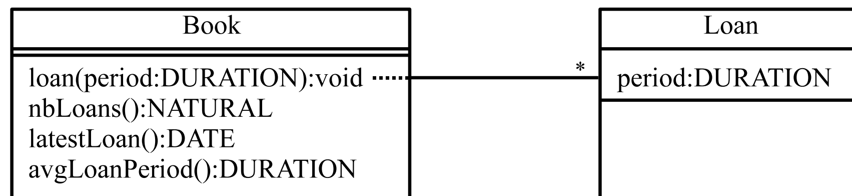


Figure 3.3: The Method Class.

for the result of an event, if any. The destruction time of each object of the class `Event` is the same as the destruction time of the corresponding object of the class `Class`. Finally, the execution time of an event is reflected in the property `creationTime` ascribed to objects of (method) classes.

Figure 3.4 shows that using the concept of a Method Class makes the PIM using reification more readible.



Context Book::nbLoans() : NATURAL
post: result = self.loan->size()

Context Book::latestLoan() : DATE
pre: self.loan->notEmpty()
post: result = self.loan.creationTime->max()

Context Book::avgLoanPeriod : DURATION
post: result = self.loan.period->avg()

Figure 3.4: PIM using Method Class.

Evaluation. The most important problem with this approach is the violation of the Principle of Minimalism. While the fact "Loan" is conceived as an *instantaneous event* with respect to the given set of real-world facts, the analyst is forced to model the fact as an object to be able to model the required queries. As a consequence, the analyst is forced to think about *the lifetime* of the fact and to invent a meaningful sense for the *destructor*. In this example, one could argue that the lifetime of the object can coincide with the period specified in the argument. But this depends on the semantics of the

argument, the class `Loan` reflects the actual loan of a book, so its lifetime can reflect the actual duration of the loan and the destruction time can reflect the actual returning of the book. The argument `period` however doesn't necessarily match with the actual duration, it can also refer to the intended duration of the loan while the actual duration can be shorter or longer. Moreover, most events don't have a `period` argument reflecting some duration, for example the event `withdraw` in the class `BankAccount`.

The principle of No Redundancy is also violated. The specification of the same fact, "loan", is distributed over a class and an event. The use of the method class reduces the amount of overhead specification, but a single fact still requires two different constructs.

Consider the following example, a class of bank accounts with a balance, two events, one to withdraw money and one to deposit money, and a query to return the number of occurrences of the event withdraw. If the analyst first models the withdraw with the related query, he must introduce a class `BankAccount` and a class `Withdraw` for the reified event. The class `BankAccount` will introduce two queries, `nbOfWithdraws` and `getBalance()`. There is no need to introduce a property to reflect the balance because it can be derived from the argument of the event withdraw (modeled as a property of the class `Withdraw`). To model the event deposit, the analyst adds an event to the class `BankAccount`, but also a property. If that property reflects the deposited amount, then the Principle of Minimalism is violated because an irrelevant fact is modeled. Moreover, this results in a strange model where two equivalent facts are modeled differently. If the analyst decides to let the property reflect the real balance, the Principle of No Redundancy is violated because the specification of withdraw needs to be extended to update the property. The analyst can also decide to reify deposit, but this means there are two ways to model the same set of real-world facts and thus violates the Principle of Uniqueness. Finally, traces of history are possibly violating the Principle of No History : the semantics of the needed property in the class `BankAccount` can be different for example.

3.4.4 Conclusion

In this section we illustrated two different ways of modeling issues concerning events.

The first approach uses properties resulting in compact models. On the other hand, the complexity grows rapidly as the number of properties grows. This approach tends to violate the following principles: Principle of Unique-

ness, Principle of No Redundancy, Principle of Minimalism and Principle of No History.

The second approach reifies the event. In models using reification, we are obliged to specify the tight connection between the event and the constructor of the reified event and between the arguments of the event and the properties of the reified event. We introduced the Method Class to decrease the complexity of the specification. The occurrence of an event always coincides with the creation of an object. In principle, each event can be reified. As a result, the only remaining events in a conceptual model would be constructors and destructors of classes. In particular, all properties are by definition immutable. Van Gestel developed MOOSE [138] that follows this approach. This approach has the advantage of offering very clear modeling guidelines. Practice has however proven that using reification to the limit gives rise to excessively complex conceptual models. Moreover, objects and events are naturally different from each other. An event can not always straightforward being conceived as an object. The most critical difference lies in the lifetime of an object, in particular it is not always easy to perceive semantics of the destruction of an object representing a reified event. There reification as a general technique does not satisfy.

By consequence, the modeler always has to make the trade-off between using properties and using reification when modeling aspects concerning events. Bekaert offers a number of criteria to indicate when to use reification and when not to do so [48]. Although in some situations one technique is clearly preferable above the other, there is a region where it is not clear which technique is better used. The decision is often made on the basis of quality requirements of the software system: when compactness (number of classes) is important properties will be chosen; when adaptability or extendibility is important reification will be chosen. However, at the level of the base PIM these requirements are not in order.

In our opinion, the main reason for this trade-off is that UML does not offer appropriate concepts to reason about behavior. The concepts used in this section lack expressiveness to model the problem domain at a desirable high level of abstraction. Both models suggest in some way a design (and implementation). In other words, the PIMs presented in this section belong at the lower levels but not at the base level.

3.5 The Base PIM

In this section we elaborate a new way to express issues related to events at the level of the base PIM. In its current form, OCL only supports the application of events and queries against objects and classes. The language has only few facilities to reason about occurrences of events and queries. We therefore introduce a new OCL-operator: the `#`-operator. We first describe the semantics of the operator and then illustrate its use in the example.

3.5.1 Semantics Of The # Operator

The OCL expression `obj#event()` denotes a collection of all occurrences of the event `event()` against the object `obj`. The evaluation of the left operand of the operator must yield an instance of a class. The event after the # must be an event of the class to which the object `obj` belongs. The shorthand for collect [12, sect. 7.6.2] is also applicable: when we apply an event to a collection of objects, then it will automatically be interpreted as a *collect* over the members of the collection with the specified event.

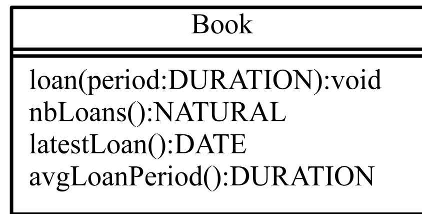
Elements of the collection resulting from the #-operator are thus event occurrences. To some extent, event occurrences are similar to plain objects. Arguments involved in event occurrences can be retrieved by means of the traditional dot operator. Assume an event `e` with arguments `a` and `b`, the OCL expression `obj#e.a` yields a collection of values supplied as the actual argument in invoking that event against the given object. We also assume the presence of the default property `executionTime` (analogously to the `creationTime` property of an object), which returns the time the event occurred. If the event is a query, the result of the query can be obtained by means of the `result()` operator [12, sect. 7.7.2].

This operator is probably an example of an operator that will only be used at the level of base PIMs.

3.5.2 The Example Revisited

In figure 3.5, the base PIM is worked out for the small part of the library system, introduced in section 3.4.1. The base PIM uses the #-operator, eliminating the need to reify the event `loan(period)` or to introduce properties.

The specification of the introduced queries is straightforward. In the specification of the query `nbLoans` we only need a standard operation on a col-




```
Context Book::nbLoans() : NATURAL
  post: result = self#loan()->size()

Context Book::latestLoan() : DATE
  pre: self#loan()->notEmpty()
  post: result = self#loan().executionTime->max()

Context Book::avgLoanPeriod : DURATION
  post: result = self#loan().period->avg()
```

Figure 3.5: The Base PIM using the #-Operator.

lection. The specification of the query `latestLoan` uses the default property `executionTime` of an event. Finally, the query `avgLoanPeriod` uses the dot-operation to retrieve the value of the argument of an occurrence of an event.

At the base PIM level we also assume the presence of useful functions like `avg` (average) and `max` (maximum). These functions are of course only available if the type of elements of the collection is appropriate.

Evaluation. In our opinion this model is a more abstract representation of the problem domain than the models in figures 3.1 and 3.4. The #-operator gives us the possibility to specify the semantics of the different queries without the need to think about *how* to model the queries. This model has the advantages of both models, namely it is compact, adaptable and extendible.

The lack of expressive power forces the analyst to add irrelevant facts in the model with properties (extra properties) or the model using reification (lifetime and destructor for instantaneous fact), in this model only the relevant facts are present. Thus, this model obeys the Principle of Minimalism.

The specification of each modeled fact is not distributed over multiple places. There is for example no need to insert a postcondition or a property to add a query. When one of the queries has to be removed from the model, the analyst does not have to change other model elements to keep the model consistent. Therefore, the model also obeys the Principle of No Redundancy.

Traces of history in the models with properties or using reification are

especially unwanted if they are reflected in irrelevant model elements (for example irrelevant properties). However, it is not always that clear whether or not we introduce unwanted history in our model. Suppose the query `totalLoanPeriod()` is also required in the example of the library. The specification of the query `avgLoanPeriod()` can then be written in terms of the queries `totalLoanPeriod()` and `nbOfLoans()` or using the `avg()`-function as done in figure 3.5. One could argue that choosing for the definition in terms of the other queries is a trace of history. We dont fully agree with this statement. Indeed, the general principle of *divide and conquer* is still a valuable approach in building high-level models. In general, the approach suggests to solve complex problems in terms of several subproblems. Applied to the specification of behavior in high-level models, the principle suggest to introduce auxiliary functions in specifying more complex functions. This is not the same as introducing history traces in models.

3.6 Transformation from PIM to PSM

One of the main objectives of MDA is to transform a (base) PIM into one or more PSMs. As stated in [125] an MDA model will have multiple levels of PIMs. The base PIM is transformed in second-level PIM(s). In the current state of the art, most transformations cannot be fully automated and probably we will never reach a state where full automation is possible. In that context, it is important to keep PIMs of different levels (and PSMs) consistent, by maintaining traceability.

In this section we will show that base PIMs, as described in section 3.5, can be transformed into two different second-level PIMs: the first PIM uses properties (cfr. figure 3.1) and the second PIM uses reification (cfr. figure 3.4). In this context, transforming a base PIM to a second-level PIM means eliminating the more abstract #-operator.

3.6.1 PIM Using Properties

In the first transformation, we generate a PIM using properties to specify the result of the queries. In the next paragraphs we describe the algorithm to transform the specification of a query using the #-operator in a specification involving properties.

The first step in the transformation is a *normalization process.* All the queries in the specification at the level of the base PIM are transformed into

queries that either do not involve the #-operator in their specification, or into queries that can be transformed into an attribute (a stored property). We refer to the latter kind of queries as *attributable queries*. A query is attributable if its specification involves a set, and the result of the query can be specified in terms of the result of that query applied to a single element, combined with the result of the query applied to the remaining elements. A precondition is considered as an *anonymous query*. When its specification contains the #-operator, the query is made explicit and treated in the same way as the other queries.

In the base PIM for the library system, the query `nbLoans()` is attributable. Indeed, the size of the set of event occurrences is equal to the size of a set with one event occurrence plus the size of the set of remaining event occurrences. The same is true for the query `latestLoan()`. The maximum of the set of execution times is equal to the maximum of a single execution time, and the maximum of the remaining execution times. The precondition of `latestLoan()` is made explicit in the attributable query `loanIsExecuted`. The query `avgLoanPeriod` on the other hand is not attributable. We therefore expand the definition of the average function in the specification, yielding another query `totalLoanPeriod`. It is easy to see that this query is attributable. After normalization, we get the query specifications shown in figure 3.6a.

The second step in the transformation is to *introduce attributes* for each of the attributable queries. The specification of the queries is now written in terms of the introduced attributes. The specification of each attribute contains a `semantics` clause specifying unambiguously the meaning of the attribute and, if possible, an intial value for the attribute. The attribute can initially be *undefined*, meaning that the result of every query using this attribute in its specification is initially undefined. These queries are *non uniform services*.

In the example, this leads to four attributes. The query `nbLoans()` introduces an attribute `nbLoans`. The initial value for this attribute is straightforward since the size of an empty set is zero. On the other hand, the attribute `latestLoan` is initially undefined. Indeed, there is no maximum of an empty set. The specification of these two attributes (the specification of all attributes can be viewed in figure 3.7) is shown in figure 3.6b.

The last step in the transformation is to *add postconditions to the event on which the #-operator is applied*. In particular, a postcondition is added for each query that has been transformed into an attribute. We start with postconditions stating that upon completion of the event, each of the attributes must store a value that is consistent with the specification of the attributed

$$
\begin{aligned}
&Context\ Book :: nbLoans()\ :\ NATURAL\\
&\quad post:\ result\ =\ self\#loan() \rightarrow size()\\
&Context\ Book :: latestLoan()\ :\ DATE\\
&\quad pre:\ self.loanIsExecuted()\\
&\quad post:\ result\ =\ self\#loan().executionTime \rightarrow max()\\
&Context\ Book :: loanIsExecuted()\ :\ BOOLEAN\\
&\quad post:\ result\ =\ self\#loan() \rightarrow notEmpty()\\
&Context\ Book :: avgLoanPeriod()\ :\ DURATION\\
&\quad post:\ result\ =\ if\ nbLoans()\ =\ 0\\
&\qquad\qquad\qquad then\ 0\\
&\qquad\qquad\qquad else\ totalLoanPeriod()/nbLoans()\\
&Context\ Book :: totalLoanPeriod()\ :\ DURATION\\
&\quad post:\ result\ =\ self\#loan().period \rightarrow sum()
\end{aligned}
$$

(a) Step 1: Normalization.

$$
\begin{aligned}
&Context\ Book ::\ property\ NATURAL\ nbLoans\\
&\quad Initial\ value\ :\ 0\\
&\quad Semantics\ :\ self\#loan() \rightarrow size()\\
&Context\ Book ::\ property\ DATE\ latestLoan\\
&\quad Initial\ value\ :\ undefined\\
&\quad Semantics\ :\ self\#loan().period \rightarrow max()
\end{aligned}
$$

(b) Step 2: Introduction of Attributes for the Attributable Queries.

$$
\begin{aligned}
&Context\ Book :: loan(period : DURATION)\ :\ void\\
&\quad post:\ self.nbLoans\ =\ self\#loan() \rightarrow size()\\
&\quad post:\ self.latestLoan\ =\ self\#loan().executionTime \rightarrow max()\\
&\quad post:\ self.totalLoanPeriod\ =\ self\#loan().period \rightarrow sum()\\
&\quad post:\ self.loanIsExecuted\ =\ self\#loan() \rightarrow notEmpty()
\end{aligned}
$$

(c) Step 3a: Introduce for each Attribute a Postcondition.

Figure 3.6: Specification of the Different Transformation Steps.

$$
\begin{aligned}
&Context\ Book :: loan(period : DURATION)\ :\ void\\
&\quad post:\ self.nbLoans\ =\ (self\#loan()@pre \cup\\
&\qquad\qquad self\#loan()@current) \rightarrow size()\\
&\quad post:\ self.latestLoan\ =\ (self\#loan()@pre.executionTime \cup\\
&\qquad\qquad self\#loan()@current.executionTime) \rightarrow max()\\
&\quad post:\ self.totalLoanPeriod\ =\ (self\#loan()@pre.period \cup\\
&\qquad\qquad self\#loan()@current.period) \rightarrow sum()\\
&\quad post:\ self.loanIsExecuted\ =\ (self\#loan()@pre \cup\\
&\qquad\qquad self\#loan()@current) \rightarrow notEmpty()
\end{aligned}
$$

(d) Step 3b: Split off the Current Occurrence.

$$
\begin{aligned}
&Context\ Book :: loan(period : DURATION)\ :\ void\\
&\quad post:\ self.nbLoans\ =\ self\#loan()@pre \rightarrow size()\\
&\qquad\qquad + self\#loan()@current \rightarrow size()\\
&\quad post:\ self.latestLoan\ =\\
&\qquad\qquad if\ self\#loan()@pre.executionTime \rightarrow isEmpty()\\
&\qquad\qquad then\ self\#loan()@current.executionTime \rightarrow max()\\
&\qquad\qquad else\{self\#loan()@pre.executionTime \rightarrow max(),\\
&\qquad\qquad self\#loan()@current.executionTime \rightarrow max()\} \rightarrow max()\\
&\quad post:\ self.totalLoanPeriod\ =\ self\#loan()@pre.period \rightarrow sum()\ +\\
&\qquad\qquad self\#loan()@current.period \rightarrow sum()\\
&\quad post:\ self.loanIsExecuted\ =\ self\#loan()@pre \rightarrow notEmpty()\ OR\\
&\qquad\qquad self\#loan()@current \rightarrow notEmpty()
\end{aligned}
$$

(e) Step 3c: Distribute the Application of the Query.

$$
\begin{aligned}
&Context\ Book :: loan(period : DURATION)\ :\ void\\
&\quad post:\ self.nbLoans\ =\ self.nbLoans@pre\ +\ 1\\
&\quad post:\ self.latestLoan\ =\ if\ (not(self.loanIsExecuted@pre))\\
&\qquad\qquad then\ now\\
&\qquad\qquad else\ (this.latestLoan@pre,\ now) \rightarrow max()\\
&\quad post:\ self.totalLoanPeriod\ =\ self.totalLoanPeriod@pre\ +\ period\\
&\quad post:\ self.loanIsExecuted\ =\ self.loanIsExecuted@pre\ OR\ true
\end{aligned}
$$

(f) Step 3d: Simplify the Postconditions.

Figure 3.6: Specification of the Different Transformation Steps. (Ctd)

query. This yields the specification for the event `loan()` as shown in figure 3.6c.

Because each of the queries is attributable, their results can be specified in terms of a single element combined with the result of the query applied to all other elements. This property is essential in arriving at a specification in which the value of the attribute upon completion of the event is specified in terms of the value of that attribute upon entry combined with the specific characteristics of the current occurrence of that event. In a first step, the current occurrence of the event (`obj#event()@current`) is split of from all past occurrences (`obj#event()@pre`). This yields the postconditions as shown in figure 3.6d.

The next step is to distribute the application of the query over the current occurrence of the event and all past occurrences. This yields specifications in which the result of the query is specified in terms of its result applied to all past occurrences and in terms of its result applied to the current occurrence. The resulting specifications are shown in figure 3.6e.

The last step is to simplify the postconditions. The result of the query applied to the current occurrence is straightforward and the result of the query applied to all past occurrences is the value of the attribute before the current occurrence. The simplified specification is shown in figure 3.6f.

In the second postcondition we observed that the condition could be substituted by an attribute. This is subject for further research but we think that we will always be able to do such substitutions; if not, there will probably be an incomplete specification in the PIM.

Some postconditions can even be further simplified. The result of the whole process is the PIM shown in figure 3.7.

To facilitate maintaining consistency between PIMs of different levels, some extra semantic information is added to the PIM in figure 3.7. Each property has a *semantics*-clause. This information indicates the connection between this property and the query in the base PIM. However, this concept must still be further refined : the link between a non-attributable query from the base PIM and the introduced properties at the second-level PIM can be made clearer for example.

3.6.2 PIM Using Reification

The second transformation from the base PIM in figure 3.5 to a PIM using reification (using the method class) is straightforward. This second-level PIM is shown in figure 3.4. The model introduces the method class `Loan` as a reifi-

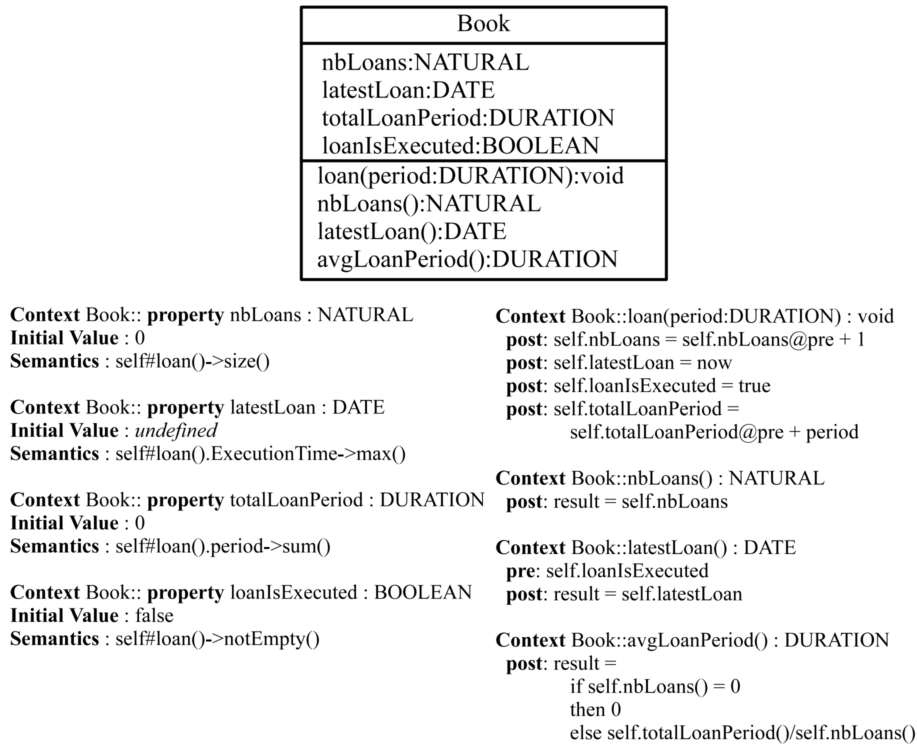


Figure 3.7: The PIM using Properties.

cation of the event `loan(period)` in the class of books. In specifications of events and queries, each occurrence of an expression of the form `obj#event()` is transformed into the expression `obj.event`. When the base PIM uses the default property `executionTime`, the second-level PIM uses the default property `creationTime` of the method class.

The presence of the method class at this level is a clear trace that this concept represents the reification of an event from the base PIM. In this way, the method class helps maintaining the consistency between the base PIM and second-level PIMs.

3.7 Objects as Arguments

The example used in the previous sections has only one argument. The argument, `period`, is a value and is thus by consequence immutable. It doesn't

matter when the value of the argument of a given occurrence of the event `loan()` is requested, the result will always be the same value. Values have by definition no lifetime, they are perpetual. However, events can also have objects as arguments. The event `loan(period)` can for example have `reader` as a second argument, representing the reader, an instance of the class `Person`, of the book who loans the book from the library.

Opposed to values, instances of classes or objects, do have a life cycle. Each object comes into existence at a some point in time, and ceases to exist at a some later (or potentially the same, but certainly never earlier) point in time. Objects have a restricted lifetime. Moreover, objects have a state. That state is determined by the properties assigned to it (such as for instance the weight of a person) and the associations it is involved in (such as for instance the membership relation between a person and a library). Generally, the state of an object evolves over time (e.g. the weight of most people goes up and down through time).

The impact of both facts, namely that objects have a lifetime and a evolving state during that lifetime is investigated in more details in the next sections. Section 3.7.1 investigates the impact of the life cycle of an object, while section 3.7.2 treats the evolving state.

3.7.1 Life cycle of an object

Quite often, when reasoning about objects in conceptual modeling the main focus is on the *living* objects. Although there are no mechanisms like *garbage collection* at the conceptual level, objects that cease to exist often *disappear* from the view because they become unreachable from other living objects, i.e. the object that ceased to exist is no longer involved in one or more associations with a living object.

With the availability of the #-operator, objects become reachable in yet another way from other objects. As soon as a given object is involved in an event at least once as an argument, the given object can be reached from the object the event is invoked against. While it is often reasonable to accept that objects that cease to exist no longer participate in associations, it is not reasonable to expect that also all objects, that have an event where the object that ceases to exist once was an argument of, also cease to exist. Suppose a library wants to send an invitation for a happening to all readers that loaned a book. Figure 3.8 illustrates the query to retrieve all readers of a book using the #-operator. This query returns a set of persons regardless if they are still *living* or not. However, the library probably only wants to send invitations to

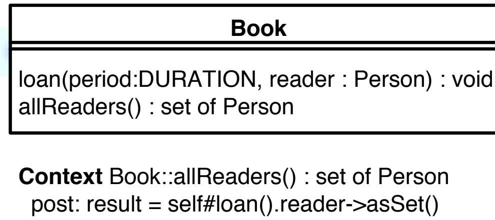


Figure 3.8: A query for retrieving all readers of a book.

living readers. Thus, the modeler needs to be able to distinct between living objects and objects that ceased to exist. This need is even more urgent in the case when departing from this set of all readers an event is invoked on each reader that changes the state of that reader. Once an object ceased to exist, its state should not change anymore.

EROOS[5] [44, 129] is a conceptual modeling method that differentiates between living objects and objects that ceased to exist. The collection of all objects of a class is partitioned in two disjunct sets: the *population* and the *archive*. From the moment it comes into existence an object resides in the population until the moment it ceases to exist and moves to the archive. Figure 3.9 shows the detailed graphical representation of a class that shows the population and archive explicitly. The inner rectangle with the character '$^\circ$'

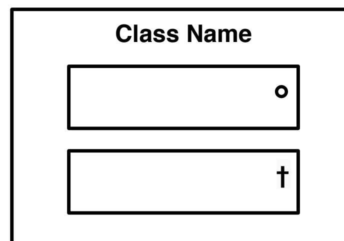


Figure 3.9: EROOS Class: population and archive.

in it represents the population, while the rectangle with the character '†' represents the archive. Next to a constructor, EROOS also provides in each class a *destructor* and an implicit property *Destruction Timestamp* reflecting the

[5]EROOS was originally an acronym for "Entity-Relationship Object-Oriented Specifications", but is currently considered to be a proper noun. [134]

moment an object ceases to exist. The partitioning in population and archive also gives the modeler more expression power concerning associations because he can explicitly state to which partition a participant must belong. The characters representing the population ('$^{\circ}$') and archive ('†') can also be used in the specification of functionality. When no explicit notation of the population or archive made, the population is assumed to be the targeted partition.

We claim that UML should be extended with a similar distinction between population and archive. Using the EROOS characters to denote both partitions the specification of the query `allReaders()` in figure 3.8 is a shorthand for: `post: result = self#loan().reader°->asSet()`. Figure 3.10

$$
\begin{array}{l}
\textbf{Context}\ Book :: allLivingReaders() : bag\ of\ Person \\
\quad post:\ result\ =\ self\#loan().reader^{\circ} \\
\textbf{Context}\ Book :: allDeadReaders() : bag\ of\ Person \\
\quad post:\ result\ =\ self\#loan().reader^{\dagger} \\
\textbf{Context}\ Book :: allReaders() : bag\ of\ Person \\
\quad post:\ result\ =\ self\#loan().reader^{\circ\dagger}
\end{array}
$$

Figure 3.10: The partition notation for a query retrieving all readers.

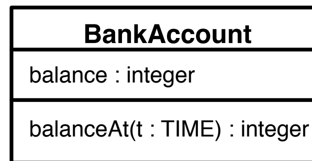
illustrates the three different possibilities to write the query to retrieve all readers of a book. The first query returns a bag (a person can read a book more than once) of all readers still alive *at the moment the query is executed*. The second query returns a bag of all readers that are dead at the moment the query is executed and the third query returns the union of both previous bags. A mutator changing the value of a property may only be used together with objects residing in the population. In other words, given the queries `allDeadReaders()` and `allReaders()` an event is not allowed to use these queries and execute a mutator to change a property of each of the retrieved readers. For example the following specification is not allowed `post: self#loan().reader°†.changeTitle('Mr./Mrs.')`.

3.7.2 Evolving state of an object

Using the #-operator, information about the past can be obtained through retrieving occurrences of an event (see figure 3.5). When objects are involved as arguments of that event, the state of that object can become part of the required information. However, the state of an object generally evolves during the lifetime of the object. As a consequence, the modeler should be able to

indicate which state must be used. In other words, the value of a property *at a given time* should be able to be retrieved.

The expression `obj.x@t` gives the value of the property `x` of the object `obj` at the moment in time `t`. The result of this expression is by definition *undefined* if the moment in time `t` lies in the future (`now < t`) or lies before the creation timestamp of the object `obj` (`t < obj.creationTime`). Figure 3.11



Context BankAccount::balanceAt(t : TIME) : integer
post: result = self.balance@t

Figure 3.11: The @-operator.

illustrates the @-operator in a simple example. There is no problem if the moment in time `t` lies between the destruction time of the object and now (`obj.destructionTime < t < now`) because objects in the archive can still be queried.

The @-operator is however not really a new operator. Expressions with the same semantics can be written with the #-operator. The @-operator is actually introduced as syntactical shorthand for a complex expression that will occur often enough to justify the introduction of this operator. Especially in business applications, history information is often needed.

The value of a property can only change through the execution of an event, or more precisely a mutator. Typical mutators for the `balance` of a bank account are `withdraw()` and `deposit()`. A more straightforward mutator `changeBalance()` is also possible. All these mutators have in common that at some point in their specification is written "*set the value of the property* `balance` *to* `someValue`". Or in other words, the more complex mutator (for example `withdraw()`) invokes at some point the basic mutator `setBalance()`. In general, we assume that each property α of a class is by definition accompanied by a basic mutator `set`α`()`. The expression `obj.`α` = value` is just another syntactical notation of `set`α`(value)`.

Given this insight, it is clear that the value of a property α at a given

time `t` equals to the value of the argument of the last occurrence before `t` of the basic mutator `set`α`(T new`α`)`. If we further assume that the default value of property at creation time of the object is also *set* by this basic mutator, there will always be at least one occurrence of that basic mutator. Figure 3.12 shows

$$obj.x@t$$
$$\equiv$$

if $((now < t)$ *or* $(t < obj.creationTime))$
then $Undefined$
else
let $setXBeforeT \;=\; obj\#setX(x) \rightarrow select(s \;\mid\; s.executionTime \;\leq\; t)$ **in**
  let $latestExecutionTime \;=\; setXBeforeT.executionTime \rightarrow max()$ **in**
    $obj\#setX(x) \rightarrow select(s \mid s.executionTime = latestExecutionTime).x$

Figure 3.12: Definition of the @-operator.

the definition of the @-operator in terms of the #-operator. If the given `t` lies outside the valid range then the result of the expression is `Undefined`. On the other hand, when `t` lies within the lifetime of the object `obj` the result is the argument of the last occurrence of `setX(x)`. First the set of all occurrences of the basic mutator `setX()` is reduced to only those occurrences that happened before the given time `t` resulting in the set `setXBeforeT`. Because the basic mutator is also used to set the default value, this set can not be empty. Next, the highest `executionTime` from the reduced set of occurrences is selected. And finally, the basic mutator with that `executionTime` is selected. This selection always returns exactly one occurrence of the mutator. The value of a property can never be set to two new values at exactly the same time. The argument of that occurrence is the required value.

Associations are almost similar to properties: participants are also set through a basic mutator. The only difference is that a property has always a value, while an object does not always have to participate in a relation when the multiplicities allow it. When an object does not participate in an association at a given moment `t`, the result of the expression, for example `account.holder@t`, is an empty set.

3.8 Object or event?

The Principle of Uniqueness states that a given set of real-world facts must lead to a unique conceptual model. With the #-operator, the need to reify

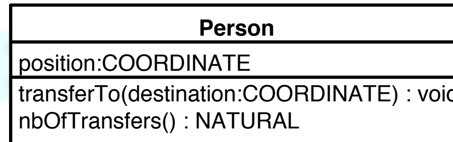
events because of technical reasons is eliminated. However, this does not mean that a given real-world fact will always be modeled as an event (or vice versa as an object). The way a given real-world fact must be modeled depends on the complete *set of relevant real-world facts*. The way a given real-world fact must be modeled depends on the complete *set of relevant real-world facts*. First, a principle is defined to guide the developer in his decision making and then the principle is illustrated with an example. The principle is a refinement of the well-known rule of thumb that in informal descriptions of the external world nouns correspond to objects and verbs correspond to events [39, 42, 126, 129].

> **Guiding Principle.**
> **[Object]** *If the lifetime, and consequently also the destructor, of a fact is of importance for the software system, then the fact should be modeled as an object.*
> **[Event]** *If a fact is instantaneous, in other words the lifetime nor destructor are of importance for the software system, the fact should be modeled as an event.*

Figure 3.13 illustrates an example of a *single* real-world fact modeled in two different ways because the set of relevant real-world facts differ, more in particular the relevant properties of the fact differ. Suppose the goal is to model a software system to manage locations of persons and their transfers. In both systems, it's irrelevant how (by plane, car,...) these transfers are performed. In the first system, modeled in figure 3.13a, two properties of the *fact "transfer"* are relevant: (1) the changing coordinate and (2) the total number of transfers. In the second system, modeled in figure 3.13b, three properties are relevant: (1) the changing coordinate, (2) the total number of transfers and (3) the total transfer time. The time a single transfer takes is only known when a person arrives at the destination.

In the first system, the fact "transfer" is modeled as an event. With respect to the set of relevant real-world facts, it is irrelevant that a transfer in reality takes some time. In the given context, it is observed as an *instantaneous* event and thus is also modeled as an event.

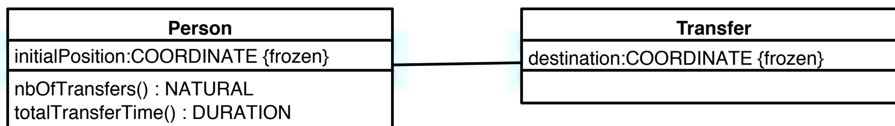
In the second system, the fact "transfer" is modeled as an object. With respect to the set of relevant real-world facts, the time a transfer takes is relevant, or in other words each transfer has a *lifetime*. The construction of an object of the class `Transfer` has a clear meaning. But more importantly, the destructor also has a *clear and relevant* meaning, namely it reflects the



Context Person::transferTo(destination:COORDINATE) : void
post: self.position = destination

Context Person::nbOfTransfers() : NATURAL
post: result = self#transferTo()->size()

(a) Transfer modeled as an event.



Context Person::nbOfTransfers() : NATURAL
post: result = self.transfer->size()

Context Person::totalTransferTime() : DURATION
post: result = self.transfer†->collect(destructionTime-creationTime).sum()

(b) Transfer modeled as an object.

Figure 3.13: The *same* fact "transfer" modeled given two different sets of real-world facts.

moment a person arrives at his destination[6].

3.9 Conclusion

In this chapter, we identified some problems with base platform independent models (PIMs) as they are developed today. More in particular, as illustrated in this chapter, base PIMs often contain design decisions. In this chapter, we focused on aspects concerning modeling behavior. More in particular, we focused on situations where the use of reification is often used as modeling technique.

We have started our research from the observation that UML has no ap-

[6]Similarly to the property `creationTime`, we assume the presence of the property `destructionTime` for every object *in the archive*. The time at which an object ceased to exist is registered in this property.

propriate facilities to reason about event occurrences, more in particular when elements of the history are important. Based on this observation, we have reviewed two common techniques to circumvent this lack of expressiveness: (1) PIMs based on Properties (section 3.4.2) and (2) PIMs based on Reification (section 3.4.3). Both techniques solve the original problem, but they do so by introducing new challenges. PIMs based on properties often need to introduce irrelevant facts, Moreover the specification tends to get spread over several elements in the model leading to problems of consistency, adaptability and maintainability. PIMs based on reification lead to overly complex models. Moreover, representing events as objects is not always straightforward because an object always has a lifetime and therefore the construction and destruction of that object must have a significant meaning. The method class (section 3.4.3.2) solves some technical issues related to reification.

We believe however that both these modeling techniques do not belong at the level of the base PIM, but at the level of lower level PIMs. Hence, we defined a new operator, the #-operator, to be able to model facts concerning history related to events. Using the #-operator, the collection of all occurrences of an event for a given object can be retrieved. Models using the #-operator do not have the disadvantages identified in the models based on properties or reification. We further showed how base PIMs containing the #-operator can be transformed into the second-level PIMs based on properties or reification. Next to values, events can have objects as arguments. To be able to treat objects properly with respect to their lifecycle, we claimed that UML should be extended with notations similar to the *population* and *archive* presented in EROOS. To ease the specification to retrieve information about the evolution of the state of an object, we defined a new operator, the @-operator. The @-operator is defined in terms of the #-operator. We finished the work with reflecting on a *guideline* when to use objects and when to use events in building high-level models.

So far the use of the #- and @-operators have only been applied to academic problems. We still need to experiment with these operators in the scope of larger software systems. We expect that the operators will especially be useful when history or evolution of values is of importance.

I was always looking outside myself for strength and confidence but it comes from within.
It was there all the time.
– Anna Freud

Chapter 4

A Framework for Executing Cross-Model Transformations Based on Pluggable Metamodels

Chapter Summary

Different metamodels mostly have common structural constructs and associated functionality: a framework offering constructs to build *hierarchical composition structures* is developed to avoid the need repeat this work over and over again. Next to these constructs, the framework offers a metamodel-independent transformation approach. The knowledge of how to transform concrete metamodel elements is decoupled from the *managing* algorithm. Developers of a transformation provide *strategies* to transform concrete model elements, while the framework is responsible for tasks as execution order, managing cross-model consistency, model validity,...

4.1 Preamble

The research of this chapter started beginning of 2006 in the context of the IWT-funded[1] project "*AspectLab*". Our part of the research is related to the section "Model Transformations".

The rise of Model-Driven Architecture (MDA) [82] triggered the development of tools supporting transformations between models. Unfortunately, most of those tools failed to surpass the stage of generating skeletons of code. The core of the code still had to be worked out after the transformation. Some research to improve the transformation results started from formal models: transformations are defined in mathematical way [139]. Other research used an operational approach based on patterns and frameworks. This work used the operational approach.

The rise of MDA also encouraged the attention for metamodeling and metaprogramming. The best known example is the metamodel of UML, describing UML using its own constructs. A metamodel is an extremely useful starting point to build tools to support software development [135]. The framework developed in this work is situated at the metamodel level and defines a language-independent transformation approach.

With regard to the software development process, this research is situated within and between all activities. In MDA terminology, Platform Independent Models (PIM) are first transformed to lower level Platform Independent Models and then to Platform Specific Models. The modeling language used for the source model and target model can be the same, but can also be different. The base idea was to develop a metamodel-independent transformation tool that supports transformations between models designed in different modeling languages. The implementation of the framework is worked out with a mix of Java 1.4 [4] and Java 5 [8] and therefore doesn't (fully) use the possibilities of techniques like generics [7], lambda expressions [26], defender methods [26], ...

The result of this research is a prototype implementation that was presented in July 2007 at Second International Conference on Software and Data Technologies (ICSOFT) [73]. The research regarding this chapter stopped end of July 2007. The prototype proved the value of the followed approach, but further research and development is needed to investigate on more specialized aspects of transformations.

[1] Agency for Innovation by Science and Technology

4.2 Introduction

In the previous two chapters, we introduced some new concepts to facilitate the development of complex software systems. However, the design of complex enterprise applications also imposes new challenges on tools supporting model-driven software design. Such complex designs are typically created using different modeling languages, where each language focuses on a specific aspect of the enterprise application. Aligning software with business requirements, for instance, is typically done using workflow modeling languages such as the Business Process Execution Language (BPEL) [5]. The technical realization of each identified BPEL process is then modelled using UML interaction diagrams, whereas the back-end database persisting the state of that BPEL process is designed using Entity-Relationship (ER) or Relational (RDB) database models. The heterogeneity of these modeling languages imposes new challenges on modeling tools. For example, Platform Independent Models crafted in earlier stages of the design process must be transformable to models written in different modeling languages, thus creating a need for *cross-model transformations* [82]. Also, general design decisions to be respected by the whole project must be communicated to models written in different modeling languages, as such creating a need for *cross-model communication* in order to guarantee *cross-model consistency* [116].

Current tool support can hardly cope with the increased complexity introduced by heterogeneous modeling languages as the majority of modeling tools rely on *hardwired, vendor-specific metamodels.* By hardwiring their metamodels, such tools disable transformations to other modeling languages. Furthermore, by relying on *proprietary* metamodels, transformation tools obstruct metamodel *reuse* in other modeling tools because those competing tools rely on ad hoc metamodels that are in turn proprietary and hence incompatible.

Being convinced that a lack of support for cross-model transformations and vendor lock-in are serious limitations of today's modeling tools, we have designed and partially implemented a *transformation framework* with support for *pluggable metamodels.* By making metamodels interchangeable, this tool allows modellers to introduce a metamodel for their own modeling language and it provides a basis for executing horizontal, *cross-model transformations.*

Overview

In this chapter, we focus on the *design* of *Pluto*, a framework providing reusable concepts for (1) building concrete metamodels and (2) for transforming concrete models between different modeling languages. The remainder of this text is structured as follows. Section 4.3 elaborates on current tool support for model transformations and discusses a number of limitations, leading to a list of design goals, as described in section 4.4. Section 4.5 introduces our framework, Pluto, and section 4.6 shows how Pluto eases the implementation of new metamodels. Section 4.7 shows how Pluto models can be transformed to target models written in other modeling languages. Finally, section 4.8 discusses related work and section 4.9 concludes.

4.3 Motivation

Tools for designing and transforming models do not fully support the functionality required for developing complex software systems [67, 88, 145]. One notable MDA implementation is the EMF (Eclipse Metamodel Framework) tool [20, 130]. This tool is widely used by research projects and provides a good means of quickly supporting a metamodel. However, EMF does not scale up for large models, does not provide any workgroup support, and is generally not used as a basis for industrial tools. EMF does not fully comply with the MOF standard [28]. Other tools like AndroMDA [17] and Enterprise Architect [29] seem to have the same problems. We have identified a number of disadvantages that have led to the design goals enumerated in section 4.4:

- **Limited Applicability.** Transformation tools are often shipped containing a *hardwired metamodel* of a *single modeling language* so as to (vertically) transform models *within that same language*. Although complex models may be realized using such tools, it is impossible to design models in any other modeling language than the one supported. This forces modellers to use a different tool for each modeling language they are willing to use. Threatened by the risk to scatter their designs over incompatible tools, developers are tempted to stick with a single, general modeling language such as UML, even though specialized modeling languages are often better suited for modeling specific parts of a software system.

- **Limited Interoperability.** Next to being hardwired in modeling tools,

embedded metamodels often contain *proprietary* constructs, for example, to increase the performance of the transformation tool in which they are embedded. Although economically feasible for the tool supplier, who achieves a vendor lock-in, such proprietary constructs are awkward for the *end users* of that tool because it becomes very hard to reuse their designs in other modeling applications. Indeed, the latter will not be able to understand the format of the *proprietary metamodel* in which the original version of the model was defined.

- **Limited Extensibility.** Modeling tools often provide a mechanism to define *extensions* for modeling languages. The Unified Modeling Language, for instance, introduces *UML profiles* and *stereotypes* to allow developers to extend UML with domain-specific modeling concepts [6]. Transformation tools may provide similar extension mechanisms to let developers extend the modeling language on which that tool operates. The *expressive power* of such mechanisms, however, is much weaker than that of a *pluggable metamodel system* because the semantics of metamodel-specific extension mechanisms are irreversibly linked to the semantics of their *base metamodel*, thus obstructing the introduction of new modeling concepts that are incompatible with that base metamodel.

- **Limited Support for Transformations.** Hardcoding a metamodel inside the transformation tool disables transformations *between* modeling languages. This makes it impossible, for instance, to transform a series of BPEL processes into a UML interaction diagram. Indeed, tools used for creating such models will lack the ability to communicate design decisions to each other, thus jeopardizing the consistency of the application being designed.

- **Limited Reusability.** Although specialized tools exist that are able to transform between multiple modeling languages, the set of supported languages is typically predefined and therefore not extensible. Also, the logic for executing such transformations is often too much tailored to the base application, hence obstructing the reuse of transformation algorithms in other tools.

4.4 Design Goals

The main objective of our research is to build a *metamodel-independent* transformation tool that supports transformations between models designed in *dif-*

ferent modeling languages. This gives rise to a number of design goals:

- **Pluggable Metamodel System.** Developers must be able to define their own metamodels and feed them to the transformation tool, as such increasing the applicability of the latter. For example, if our transformation tool is running on a UML metamodel, it must be able to accept an ER metamodel or an RDB metamodel and then allow modellers to design their applications using the UML, ER and RDB modeling languages. Next to adding metamodels of existing modeling languages, developers must also be able to define and insert *custom metamodels* of domain-specific modeling languages or language extensions that they have defined themselves.

- **Cross-model Transformations.** Next to transforming between models that share a common metamodel, our tool must support transformations between *different modeling languages* [103]. This requires our tool to be able to work with multiple metamodels *simultaneously*. When fed with the ER and the UML metamodel, for instance, it must be possible to transform a UML class diagram into an ER schema, and vice versa.

- **Reusability.** Metamodels have a wide-spread applicability beyond the domain of model transformations as they can be used, for instance, in model verifiers or code generators. Therefore, the metamodels on which our tool relies must be *reusable* in these domains without modification: we must avoid polluting metamodels with dependencies on components that are specific to model transformation logic.

- **Obliviousness.** In order to allow the transformation tool and the metamodels to evolve independently, a certain amount of obliviousness is needed in both directions. On one hand, metamodels should be entirely independent of the transformation tool (as noted in "*reusability*"); on the other hand, the transformation tool should not depend on the technical details of a single hardwired metamodel (as noted in "*cross-model transformations*"). Thus, we need to define a *common contract* on which the transformation tool relies when a metamodel is fed to it.

Realization. Given these design goals, we have developed *Pluto*, a framework with reusable concepts for metamodels and model transformations. Section 4.5 outlines this framework. Its two main functions, *metamodel realization* and *model transformation*, are discussed in sections 4.6 and 4.7, respectively.

Traceability. The need for horizontal, cross-model transformations in turn creates a need for *cross-model communication* so as to guarantee the consistency of the design. Therefore, transformations must be *traceable*, meaning that our tool must be able to link target elements to the source element(s) that triggered the creation of the former. This also means that changes to a target model must be communicated back to the source model from which that target model was created.

4.5 The Pluto Framework – Overview

Pluto is a Java framework providing abstract modeling constructs to be reused by concrete metamodels. It is therefore typically layered on top of *concrete* metamodels as shown in figure 4.1. The upper part depicts a set of Pluto

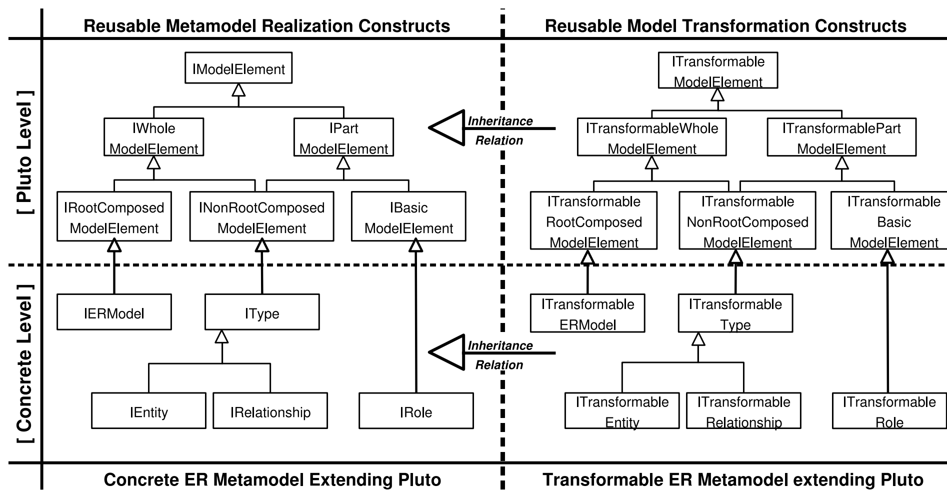


Figure 4.1: Overview of the Pluto framework for metamodeling

constructs providing general functionality for metamodels. The lower part of figure 4.1 shows how interfaces of the ER metamodel are defined as *extensions* of Pluto interfaces. Concrete realizations of these interfaces represent elements that occur in ER models. Such concrete classes implement these ER-specific interfaces by inheriting general functionality from abstract Pluto classes and by filling in ER-specific behaviour where necessary.

It is important to note that Pluto is in itself *not a metamodel of metamodels* and neither is it a *concrete metamodel*. Rather, Pluto should be seen

as an *abstract metamodel* from which concrete metamodels inherit common functionality. Therefore, in the Meta Object Facility layering structure [11], Pluto is situated at the meta-level (M2) rather than the meta-meta-level (M3).

Implementation. The (completely formally specified) implementation of the framework is worked out with a mix of Java 1.4 [4] and Java 5 [8]. Java 5 introduced among others generics [7]. The implementation only partially takes advantage of *generics*. The use of the collections framework has been refactored to use the generic collections framework, but generic or parameterized types are not used. The reason is purely historically, when the first experiments and implementation steps of this research were performed Java 5 was still in beta. Because the main focus of the research was to develop a prototype to prove the design goals (section 4.4) and not to build a fully operational transformation tool, we decided not to refactor the implementation to use generics.

The implementation uses a simplified version of the family of patterns presented in chapter 2. This version did not differentiate between the different kinds of requirements yet. All requirements are checked in a single inspector `canHaveAs`α`(T` α`)`. In this way, the implementation of the framework can be seen as a preliminary validation of the pattern. In the current implementation, this inspector is also responsible for checking correct typing in defining for example the whole-part relation (see section 4.6.1). It is subject to future research to determine how parameterized types can be used to improve the implementation of the framework.

The implementation of the framework consists of 56 classes and 43 interfaces and counts in total 3205 lines of sources code[2]. The mean of the McCabe Cyclomatic Complexity is 1.331 with standard deviation 0.915. The specification of the implementation is also completely (formal and informal) worked out following the rules described in [127].

The remainder of this chapter focuses on two key functions of Pluto.

- **Metamodeling.** Section 4.6 focuses on the *left part* of figure 4.1 and shows how Pluto offers reusable concepts for *composition* and *dependency management*. To illustrate the wide applicability of this transformation framework, we use the Entity Relationship (ER) metamodel and the Relational Database (RDB) metamodel as examples instead of the UML metamodel.

- **Transforming Models.** Section 4.7 reviews the *right part* of figure 4.1

[2]Metrics are calculated with [30].

and shows how Pluto transforms models between different modeling languages. As an example, we show how the ER metamodel can be extended so as to transform models from the ER modeling language to the RDB language, and vice versa.

4.6 Designing Concrete Metamodels as Pluto Extensions

Pluto offers constructs for building *consistent, hierarchical composition structures* (section 4.6.1) and provides a *reusable dependency management system* (section 4.6.2) that can be reused by developers of concrete metamodels. Both constructs provide functionality that is paramount for Pluto's transformation logic (section 4.7).

4.6.1 Reusable Composition Concepts

The existence of nested *whole-part relations* typically causes models to be arranged into *hierarchies*. The `ERModel` instance in figure 4.1, for example, refers to a number of `Relationship` instances, and each `Relationship` in turn contains a number of relationship ends (represented by instances of `Role`). Similar composite structures occur frequently in the majority of available modeling languages. Without having reusable constructions for managing model composition, however, implementors of metamodels have to implement similar structures over and over again. We avoid this repetitive and error-prone work by integrating reusable functionality for managing hierarchies at the level of Pluto. This is achieved by structuring its top level classes according to the *Composite* design pattern [83]. This composite structure is shown in figure 4.2, where instances of `WholeModelElement` refer to instances of `PartModelElement`. The basic version of the Composite pattern is further extended by the introduction of three *subinterfaces* that differentiate between the root node, the intermediate nodes, and the leaves of a model tree:

- `RootComposedModelElement` is used for elements that are not contained in other elements, such as the *root node* of a model. In the ER metamodel (see figure 4.2), for instance, this interface is extended by the `ERModel` interface because ER models cannot be contained in other model elements.

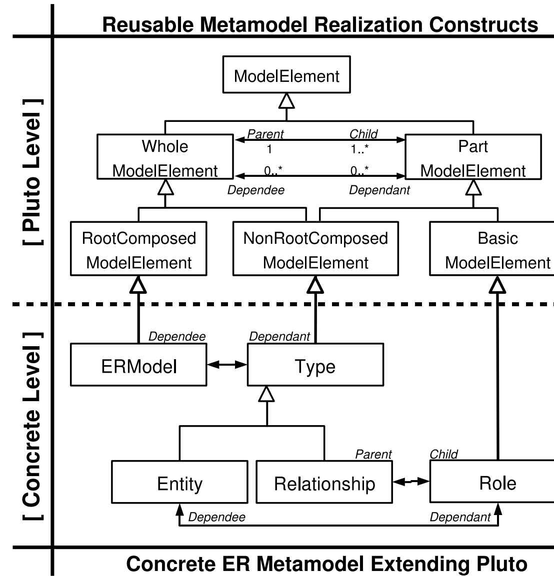


Figure 4.2: Parent/child and dependee/dependant relations in Pluto

- `BasicModelElement` is used to represent those model elements that are contained in other model elements without containing elements themselves. They represent the *leaves* of a model tree. An example of such a basic model element is the relationship end of an `Entity`, as represented by `Role` in figure 4.2.

- `NonRootComposedModelElement` inherits its behaviour from both `WholeModelElement` and `PartModelElement`, meaning that it contains model elements while being contained in another model element. Concrete realizations of this interface represent *internal nodes*. The `Relationship` interface, for instance, is an internal node because it is contained in an `ERModel` (see figure 4.2) while containing a set of `Role` instances.

Evaluation. By augmenting the semantics of model composition, we allow metamodel developers to *reuse* Pluto's consistency and validity checks, as such decreasing the odds for introducing *structural integrity violations* (e.g. circular dependencies or detached model elements) during the construction of a new metamodel.

4.6.2 Reusable Dependency Management

The *whole/part* relation of section 4.6.1 is too general to cover different kinds of dependencies between model elements. Therefore, this section further decomposes that relation into *two specialized dependency relations*. First, the *parent/child* relation is used to specify *interdependencies*. Second, the *dependee/dependant* relation is used to specify *unidirectional dependencies* between different kinds of model elements.
The Parent/Child Relation. This is a hierarchical one-to-many relation used for expressing interdependencies between different kinds of model elements. The following properties characterize the parent/child relation:

- **One-to-Many.** A parent can have multiple children, but a child only has one parent. For example, the realization of the ER metamodel specifies that a `Relationship` (parent) contains multiple relationship ends (i.e. instances of `Role`), but a `Role` (child) can only belong to one `Relationship`.
- **Interdependence.** The parent/child relation is used to express the existence of *bidirectional dependencies* between different kinds of model elements. This means that parents and their children influence each other:
  - *Children depend on their parent.* Children cannot survive the removal of their parent. In the ER metamodel, for instance, this means that instances of `Role` cannot survive the removal of the `Relationship` to which they belong. This is exactly what the ER metamodel must enforce because a role does not make sense when it is not participating in any relation.
  - *Parent depends on children.* In parent/child relations, the parent also depends on its children. A `Relationship`, for instance, is invalid without instances of `Role` being attached to it; such a configuration would specify the existence of a relationship without participants and this is not sensible.
- **Transitive.** Given three model elements α, β, and γ, such that α is a parent of β and β is a parent of γ, then α is an indirect parent (or *ancestor*) of γ.
- **Symmetric.** A model element α is the parent of β iff β is a child of α.

Combined with *transitivity*, we get that α is an *ancestor* of β iff β is a *descendant* of α.

- **Non-reflexive and Acyclic.** The data structure induced by the parent/child relation is *acyclic*. No model element can be its own parent and a child cannot be the parent of one of its ancestors.
- **Tree structure.** The non-reflexive, transitive closure of the parent/child relationship induces an *acyclic tree structure* on a set of model elements.

The Dependee/Dependant Relation. This relation models a hierarchical many-to-many relationship, meaning that a dependant relies on one or more dependees, which in turn have zero or more dependants. The properties of the dependee-dependant relation are summarized below:

- **Many-to-Many.** Other than parent/child interdependencies, where a child only has one parent, a dependant can have *multiple* dependees.
- **Unidirectional Dependency Relation.** Unlike the parent/child relation, where a parent depends on its children, dependees are *independent* of their dependants. In the ER metamodel of figure 4.2, for instance, an `Entity` is a dependee of its `Role` instances (and not a parent) because the ER specification does not require an `Entity` to participate in a `Relationship` to be valid. The reverse, however, remains unchanged: the deletion of an `Entity` is cascaded to its `Relationship` instances in order to ensure model consistency.
- **Transitive.** Similar to the parent/child relation, dependee/dependant is a transitive relation. In figure 4.2, for instance, `ERModel` is a dependee of `Entity`, which is in turn a dependee of `Role`. Therefore, `ERModel` is an *indirect dependee* of `Role` whereas `Role` is an *indirect dependant* of `ERModel`.
- **Non-reflexive, Symmetric, Acyclic.** The dependee/dependant relation has the same mathematical properties as the parent/child relation. By installing a many-to-many dependency relation, however, the non-reflexive, transitive closure of the dependee/dependant induces an acyclic *lattice* structure on a set of related model elements, rather than a *tree*.

Reusable Dependency Management. It is clear that mathematical properties such as *symmetry*, *transitivity* and the abscence of *cyclic dependencies* require rigorous specifications for managing the dependencies between different kinds of model elements in trees and lattices. Without a reusable infrastructure for dependency management, metamodel implementors would have to implement these relations over and over again, thus increasing the possibility of introducing inconsistencies in the metamodel. By integrating dependency management at the level of Pluto, however, concrete metamodels extending this framework inherit (1) fine-grained consistency guarantees and (2) algorithms that automatically manage dependencies based on lifecycle changes of model elements, thus easing the implementation of new metamodels.

4.7 Model Transformations

Next to offering reusable constructs for building metamodels, Pluto incorporates concepts for *transforming* models defined as concrete instances of those metamodels. These concepts are shown in the right part of figure 4.1. The basic idea of model transformations in Pluto is shown in figure 4.3, which shows how transformable behaviour is attached to model elements using the *Decorator pattern* [83]. These classes decorate model elements with transformable

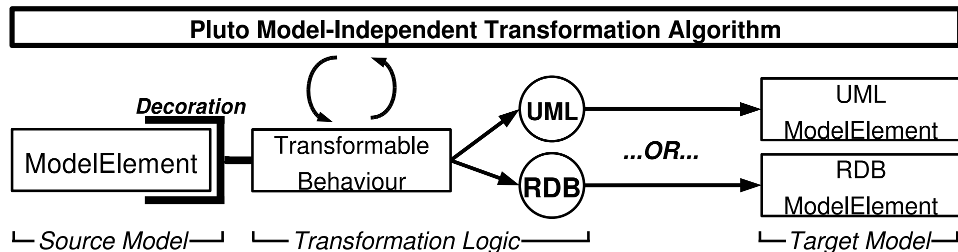


Figure 4.3: Transformation Overview

behaviour by pointing to a *strategy* containing an *execution plan* for creating a target element. Developers willing to transform some source model to a UML class diagram, for instance, attach UML-specific transformation strategies to that decorator, whereas database modellers will select RDB- or ER-specific strategies. This section first explores how transformation strategies are added to metamodels via decorators (section 4.7.1) and then shows how these transformation strategies interact with Pluto's generic transformation algorithm

(section 4.7.2).

4.7.1 Decorating Model Elements with Transformation Strategies

The upper right part of figure 4.1 shows the key interfaces exported by Pluto that attach transformable behaviour to model elements. There is a one-to-one mapping between transformation-specific interfaces (e.g. `TransformableModelElement`) and basic interfaces for metamodels (e.g. `ModelElement`) as shown in figure 4.4. There is also a *default implementation* for each transformation-specific interface to be reused by decorators of concrete metamodels. This is also shown in figure 4.4, where a concrete decorator, `TransformableEntity`, inherits from a Pluto class in order to decorate `Entity`. Such concrete decorators must be created only once for each metamodel element because they are *fully agnostic* about technical transformation details. They only serve as a "bridge" between model elements and their *execution plan*. These execution plans provide mappings between source elements and target elements, so they are encapsulated into *transformation strategies*, shown as *circles* in figure 4.4.

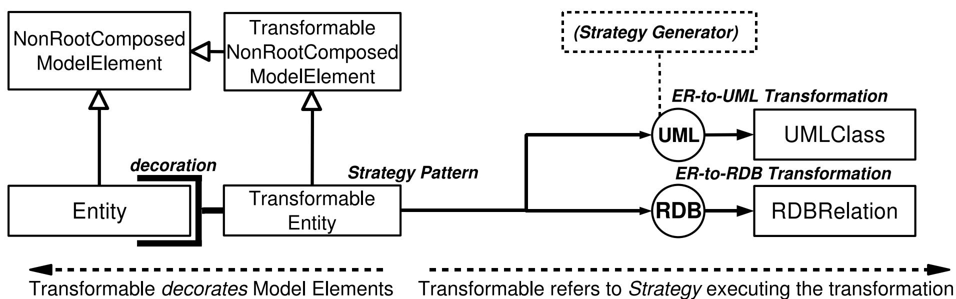


Figure 4.4: Reusable Transformation Concepts

Transformation Strategies. The main responsibility of a strategy is to *encapsulate an execution plan* for the transformation of a source element into the corresponding model construct of a target language. Figure 4.4, for instance, shows two different transformation strategies that can be attached to a `TransformableEntity` decorator. Each strategy contains a different execution plan because they both transform instances of `Entity` to a different target modeling language –UML and RDB, respectively.

Each strategy publishes an operation `transform()`, which is called by

Pluto through the decorator to which the strategy is attached. The execution of such a transformation strategy only has a *local effect*; it transforms a single model element, independent of its surrounding elements. Any dependencies with related model elements are handled by the dependency relations that were integrated in Pluto, as discussed in section 4.6.2. Such localized transformations make the implementation of strategies straightforward, as such making them eligible for *code generation*. Although not a target of this research project, it should be manageable to add an extra layer above Pluto that converts model transformation languages into strategies that can be executed by our transformation algorithm.

4.7.2 Pluto's Generic Algorithm for Transforming Model Elements

The previous section explained how transformation strategies are attached to decorators in order to make model elements transformable. This section shows how Pluto's generic algorithm interacts with these strategies in order to transform model elements. The basic philosophy of this algorithm is to exonerate the developer from taking care of all the technical "middleware" concerns of model transformations. Such tasks include determining an execution order, managing cross-model consistency, ensuring source model validity, etc. Instead, developers only have to provide *strategies* containing *localized execution plans*, which are called by Pluto whenever they are needed during the execution of its transformation algorithm.

This general idea is applied in figure 4.5, where instances of `Entity` are transformed to instances of `Relation`. The vertical arrow indicates the execution of the algorithm and the horizontal arrow shows the transformation of the model element. The transformation algorithm is explained in three steps: (1) *transformation preconditions*, (2) *transformation rules*, and (3) the *transformation protocol*.

Transformation Preconditions. For the transformation algorithm to run correctly, we need to make some assumptions about the source model:

V1 The model is *valid*, meaning that every model element obeys its invariants and that all dependee/dependant relations and parent/child dependencies are wired correctly.

V2 The model is *immutable* during the transformation. Changes made to the source model during the execution of a transformation are not reflected in the target model.

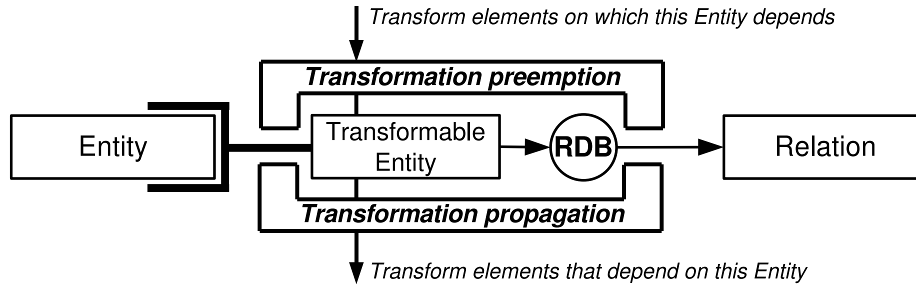


Figure 4.5: Strategies interact with the Pluto transformation algorithm

Transformation Rules. Given that our two validity rules, **V1** and **V2**, are satisfied, we can execute the desired transformation. The generic transformation algorithm of Pluto uses three rules to specify a *transformation order* among a set of transformable model elements:

T1 Model elements can never directly transform other model elements; they can only transform themselves. It is only possible to start the transformation of another model element *indirectly*, as will be explained below.

T2 Elements can only transform themselves *after* their parent and dependees have been transformed. Thus, an element may have to wait for other elements before it can transform itself, which is called *transformation preemption*, as shown in figure 4.5.

T3 After its transformation, a model element indirectly triggers the transformation of its children and dependants. This is called *transformation propagation* (see figure 4.5).

Two-Phase Transformation Protocol. Given our transformation rules, (**T1**–**T3**), by which concrete transformation strategies must abide, every model element can decide whether or not to transform itself based on *local* information, i.e. by determining whether its dependees and parent have been transformed (**T2**). Therefore, the transformation of a model element can be decomposed into two different transformation steps:

CT **Conditional Transformation Request.** These requests are sent by external model elements to the model element that must be transformed.

These requests are termed *conditional* because it is not guaranteed that the transformation will be executed *directly*. Indeed, the request may be *preempted* so as to conform to rule **T2**. Also, a **CT** request will propagate new **CT** requests to all the dependants and children of the model element after it has transformed itself (as required by **T3**). The implementation of the **CT** operation is *independent* of concrete transformations, so we have encapsulated this logic in the Pluto framework.

AT **Actual Transformation.** Unlike the **CT** phase, this second operation does not check preemption constraints and it does not propagate calls to children or dependants. Instead, it activates the execution plan of the strategy that was attached to the transformable model element, causing the target model to be manipulated. Therefore, this operation is private to the model element and it is called by **CT** when all preconditions have been satisfied, i.e. when the parent and the dependees of this model element have been transformed such that preemption is no longer required. This is the only phase that is specific to a model transformation, so these *actual transformations* must be provided by the developers by means of a concrete *transformation strategy* object, as explained in section 4.7.1. The other steps of the algorithm are transformation-independent, so developers can reuse them in concrete transformations without further configuration.

4.7.3 Illustration

Given a set of preconditions, a set of transformation rules, and the dichotomy between *conditional* and *actual* transformations, we now explain the generic execution strategy of our transformation algorithm by a simple example. Assume a conditional transformation request, **CT**, arrives at `ModelElement` in figure 4.6 and assume that the model is *valid* according to the *transformation preconditions* **V1–V2**. This section describes the steps taken by the transformation algorithm of Pluto in order to create a target model for `ModelElement` and its related elements. As noted above, this target model can be written in any other modeling language, depending on the contents of the concrete *transformation strategy* instances that were provided by the implementor of the transformation. This illustration is therefore independent of the chosen target language.
[Phases 1–6] Preemption of ModelElement. Pluto first checks whether `ModelElement` has any parent or dependees that have not executed their trans-

formation, as required by **T2**. After discovering that $\texttt{Parent}_{ME}$ is not transformed, the transformation of `ModelElement` is preempted and a conditional transformation request **CT** is forwarded to $\texttt{Parent}_{ME}$. Pluto repeats the **CT** procedure for that parent and finds that $\texttt{Parent}_{ME}$ can transform immediately without being preempted (according to rule **T2**). Thus, the strategy attached to that parent is executed and a target element is created. Next, according to **T3**, the conditional transformation request must be propagated to all children, causing the call to reach `ModelElement`. Due to the existence of a dependee that still needs to be transformed, however, `ModelElement` is again preempted and a **CT** request is sent to $\texttt{Dependee}_{ME}$. The latter executes its transformation **AT** and propagates a **CT** request to its children.

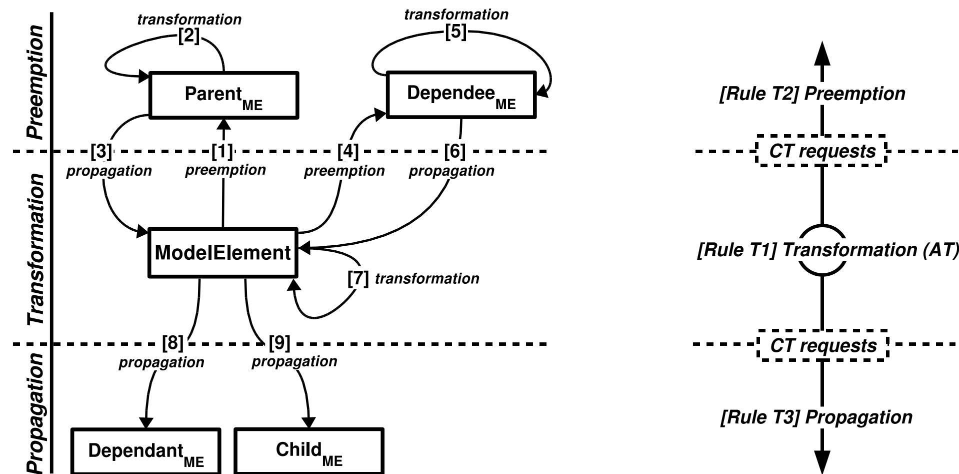


Figure 4.6: Illustration of Pluto's transformation algorithm

[Phase 7] Transformation of ModelElement. Downward propagation of the **CT** request from $\texttt{Dependee}_{ME}$ eventually reaches `ModelElement`, one of the dependants of $\texttt{Dependee}_{ME}$, so Pluto checks whether `ModelElement` can transform itself. All preconditions have been satisfied, since both its parent and its dependee have successfully transformed, so `ModelElement` can execute its **AT** transformation according to **T2**. To do so, Pluto delegates to the *strategy* that was attached to the *decorator* of `ModelElement`. This strategy now manipulates the target model, possibly making use of the elements that were already created by its parent and dependee.

[Phases 8–9] Propagation from ModelElement. After the execution

of `ModelElement` has completed, Pluto propagates a **CT** request to all the children and dependants, causing the transformation algorithm to visit $\texttt{Dependant}_{ME}$ and $\texttt{Child}_{ME}$. These elements can transform themselves without being preempted.
Due to space limitations, we cannot give a more realistic example of this transformation algorithm. Therefore, we refer to our technical report [70] for more information about the technical details of this algorithm in the context of multiple dependees or transitive dependencies between model elements.

4.8 Related Work

In [131] and [140], the applicability of Action Semantics (AS) [109] to model transformations is studied. The authors conclude that AS can be used for transforming between UML models, thus allowing for iterative refinement of UML designs. Another advantage of using AS is that design patterns can be encoded as a sequence of transformation steps, thus allowing to *refactor* designs. One shortcoming is that cross-model transformations are not supported because AS is irreversibly linked to the Unified Modeling Language.

YATL [111] is a language for defining model transformations. It combines declarative concepts for querying the source model with imperative constructs for executing the transformation itself. By relying on the Meta Object Facility [11], YATL provides support for pluggable metamodels, but the transformation tool does not offer reusable concepts for *defining* those metamodels. Furthermore, it is not clear how the transformation language can be applied to concrete instances of such newly introduced metamodels.

MTRANS [112] is a model transformation framework that provides both a development environment and a language to define model transformations. This language is defined as an abstraction above XSLT and, therefore, the transformation architecture of MTRANS is strongly influenced by the XSLT specification. One major drawback of this dependency is that many-to-one transformations are not supported because XSLT inherently relies on one-to-one mappings between source and target elements. Pluto, on the other hand, is independent of any transformation language and transparently manages one-to-many and many-to-one dependencies for model transformations.

UMLX [146] and VMT [124] are graphical transformation languages for MDA, primarily developed in an attempt to increase user-friendliness of model transformation languages. The major advantage of UMLX and VMT is their expressiveness, given their limited amount of graphical modeling constructs.

One problem, however, is that these transformation languages are limited to transforming UML models. As the UMLX compiler is able to compile transformations into Java code, however, it should be possible to attach this generated code to our transformation strategies. Indeed, our model transformation framework fits into a larger framework, Chameleon, [135], which can be used to *transform between programming languages.* Concrete examples of such transformations can be found in the work of [71, 74, 136]. By integrating Pluto's support for language transformations with the VMT compiler, we integrate a visual transformation language with *cross-model transformability,* thus solving the problems of the original VMT proposal, which relies on a hardwired metamodel.

Finally, a large number of transformation languages have been proposed, for example, Converge [133] and the work of Kuznetsov [95]. These languages are typically compiled and executed on a transformation tool that relies on a hardwired metamodel, thus disallowing *pluggable metamodels.* We are investigating how these transformation languages can be compiled to our transformation strategy objects, which is beneficial for both paradigms: (1) the transformation language can be used for cross-model transformations and (2) the developer is freed from having to program strategies.

4.9 Conclusion

Modeling tools often rely on hardwired, proprietary metamodels, as such obstructing *cross-model transformations* and metamodel reuse. This leads to inconsistent designs scattered over a variety of modeling tools. We have implemented Pluto, a framework providing reusable concepts for *metamodeling* and for *model transformations.* Pluto eases the definition of new metamodels by providing reusable concepts for dependency management and model composition. Furthermore, Pluto enables cross-model transformations by deferring model-specific transformation logic to *strategies* containing localized execution plans.

The decoupling between *metamodel-independent* constructs offered by *Pluto* and *model-specific* concepts provided by *developers* decreases the development time of new metamodels and increases their consistency because modellers can focus on metamodel-specific concepts while inheriting common modeling functionality from Pluto.

Have no fear of perfection.
You'll never reach it.
– Salvador Dali

Chapter 5

Conclusion

In this chapter, we summarize the main contributions of this dissertation and discuss possible directions for further research.

5.1 Summary and Contributions

Our research is based on the fundamental observation that *developing software is much more complex than often thought.* The underlying driver is that every software development process involves multiple activities, with each activity using its *own modeling language* or its *own programming language*, in order to build its deliverable and advance the software system to the next development activity.

Doing complex work becomes easier with a powerful tool. This is no different in software development. In this dissertation, we have shown that people who are involved in such a software development process (analysts, design engineers and programmers) *face a much tougher task when their "tools" are less "powerful"*.

Software development is inherently hard, that is termed "essential complexity" by Brooks in his article, No Silver Bullet [60]. Software development is unnecessarily harder when language constructs are less expressive. Lower expressiveness and less abstraction introduce unnecessary and avoidable challenges that is termed "accidental complexity" by Brooks. This dissertation provides guidelines in the form of patterns or language extensions to reduce the accidental complexity in software development.

A Pattern-based Approach for Properties and Associations

In Object-Oriented programming languages, concepts are represented by *objects*. Properties and associations together represent the state of an object. Constraints on those properties and associations determine when an object is in a steady state (meets its invariants). On the one hand, the implementation of those requirements must facilitate redefinitions in subclasses. On the other hand, methods that change the state of an object must guarantee that all involved objects are still in a steady state after the update. To achieve both challenges, the implementation and specification of the requirements is *separated* from the actual changing of the state and is encapsulated in *inspectors*.

Three types of requirements are distinguished, namely *value requirements*, *state requirements* and *transition requirements*. The collaboration of the all methods is defined by a family of patterns. For associations two different mutators, that guarantee all involved objects will be left in a steady state afterwards, are provided.

A disadvantage of patterns, is the presence of (a lot of) *boiler plate code*. Advanced languages constructs can avoid this boilerplate code, therefore some first steps towards the introduction of language constructs are set.

Added Value. A *separation of the concerns* between the development of methods describing requirements and the methods describing the state changes is reached. Three different kinds of requirements are identified to facilitate the (re-)definition of requirements at different positions in the class hierarchy. Each kind of constraint is encapsulated in its own inspector. A family of patterns describes how all methods collaborate. Finally, some ideas are presented to show how the patterns can be replaced by language constructs.

Concepts for Abstracting away Object Reification

Conceptual models introduce accidental complexity when they contain technical aspects in order to describe real-world facts. Such complexity is introduced by enforcing ("locking in") decisions that should have been made in a later activity in the software development process.

The need for technical solutions arises from the absence of appropriate "tools" offered by the modeling language that is being used. UML and OCL lack expressive constructs to reason about event occurrences, even more so when the historical aspect of such occurrences becomes important.

In this dissertation, we have defined a new operator, the #-operator, that allows analysts to treat events as full-fledged elements. By assigning a property, `executionTime`, to events, it becomes possible to model historical event information without the need to introduce irrelevant facts in the conceptual model.

Furthermore, UML also lacks facilities to reason about objects that are still relevant for the software system after they have ceased to exist. UML currently forces the analyst to model the destructor and the consequences of destruction (i.e., the state represented by properties should no longer change) explicitly. We claim that UML should be extended with concepts of *population* (living objects) and *archive* (objects that ceased to exist) as they are offered by EROOS [129].

A conceptual model never describes the whole universe, but is always a description of a subset of real-world facts. We have illustrated that the decision to model a given fact as an object or as an event depends on the selected subset of real-world facts. A guiding principle is defined to assist the analyst in his decision-making: if the lifetime of a fact is of importance then the fact should be modeled as an object. Otherwise, if the fact is instantaneous, the fact should be modeled as an event.

In parallel with our research, OMG has identified a similar need to reason about events. OCL now contains the ^^-operator. [15, Sect. 7.8.1] The semantics of this operator are the same as the #-operator: the result gives all `OCLMessage`s sent. OCL, however, does not assign a property to `OCLMessage` reflecting the time of occurrence.

Added Value. We have lifted events to the same level as objects. To enable the reasoning with events, a new operator, the #-operator, is introduced. The need to distinguish between the different stages of an object, namely before and after destruction, is illustrated and the offered solution in EROOS is proposed as extension for UML. Finally, we have defined a guiding principle to decide whether a fact should be modeled as an object or as an event.

A Framework for Executing Cross-Model Transformations

Generally, multiple languages are used during the development of a software system. Each language is formally defined by a metamodel. These metamodels serve as the basis to define transformations between the different

models. We started from the observation that different metamodels share the same structural constructs and all need to develop the functionality that is coupled with these concepts over and over. By offering these constructs in a framework, this work can be avoided.

Next to offering these constructs, the framework now also has the possibility to offer metamodel-independent transformation strategies. The knowledge of how to transfer concrete metamodel elements is decoupled from the *managing* algorithm, which enables the transformation engineer to concentrate on the real problems. Next to the *Parent/Child* dependency, the framework also needs to offer a second kind of dependency relationship, namely a *Dependee/Dependant* dependency to be able to perform metamodel-independent transformations.

The framework has been validated with a prototype implementation performing a basic transformation between simplified ER-models and relational models.

Added Value. A framework offering reusable concepts for metamodeling eases the task of developing a concrete metamodel. Metamodel independent constructs can be reused over different metamodels. Moreover, these metamodel-independent structural constructs make it possible to define metamodel-independent transformation algorithms which in turn ease the task of developing a concrete transformation.

5.2 Directions for Future Work

For each of the presented contributions, we point out some possible directions for further research.

A Pattern-based Approach for Properties and Associations

The language construct is by definition non-deterministic. The construct needs to be further elaborated to make it possible to close specifications, or in other words to make the specification deterministic. Furthermore, the construction now makes it only possible to define the *false*-part, i.e. values or states that are not allowed can be excluded. However, it must also be possible to define a *true*-part, i.e. values or states that cannot be excluded at any time or at any place. This is not the same as making the definition deterministic.

Functional requirements can demand to change two or more properties at

the same time, in an atomic way. It has to be investigated how such mutators can be build based on the pattern. A simple sequence of setters is certainly not a generic solution as it is possible that changing one of the properties in isolation is not acceptable, while changing all properties together can still result in a steady state.

Construction is in fact an example of changing multiple properties at the same time. Characteristics of the object under construction have no previous state and are not really *changed*. But during the construction, the object may also be attached to one or more other objects. Naturally, these other objects have a previous state. Moreover construction is a special moment in the lifetime of an object. There may be special requirements that apply only that single moment. The process of construction can become even more complex in the context of inheritance and must be investigated thoroughly.

Concepts for Abstracting away Object Reification

Expressions using the #-operator can become rather long. For example the set of occurrences often first has to be reduced, based on values of the arguments, before the actual calculations can be done. The definition of the #-operator can be enriched with syntactic shorthands to improve the usability of the operator. Figure 5.1 sketches this idea.

$$book\#loan(myReader, \geq 3'DAYS)$$
$$\equiv$$
$$book\#loan() \rightarrow select(l \mid l.reader = myReader \; and \; l.duration \geq 3'DAYS)$$

Figure 5.1: Shorthand to reduce the set of occurrences.

The availability of an event can be restricted. Protocols limit the order, time or state in which events can be used. The restricted availability is also know as *nonuniform service availability*. UML uses state diagrams to model nonuniform service availability. It must be investigated how the #-operator relates to state diagrams.

Multiple events can share large parts of their specification. This is often the case when the events describe similar behavior. For example, the events `deposit`, `withdraw` and `transferTo` are related to each other. *Inheritance* is a technique that enables different classes to inherit common components from a base class. Similarly the concept *inheritance* can be used

to describe commonalities between events. For example, an abstract method `transaction` could be introduced to specify the common behavior between the events `deposit`, `withdraw` and `transferTo`. This relation between events is subject of further investigation.

A Framework for Executing Cross-Model Transformations

The framework presented in chapter 4 proved that developing a metamodel can benefit from a framework at the same MOF-level. The prototype implementation has been developed with a subset of the ER-metamodel and relational database metamodel and some transformations translating ER-models into relational models. Larger experiments are needed to further investigate the framework.

The framework is developed with older versions of Java, namely version 1.4 and some elements from version 5.0. Compared to those versions, Java 8 offers lots of new powerful tools to develop software systems. A new version of the framework can probably benefit a lot from generic classes, lambda expressions, and defender methods in interfaces.

Appendices

Appendix A

Property Pattern

```
1 public class Foo {
2
3   public Foo(T α,R δ) throws IllegalArgumentException{
4     registerα(α);
5     registerδ(δ);
6   }
7
8   /********************************************************
9    *                      Property α                      *
10   ********************************************************/
11
12  private T α;
13
14  @Basic
15  public T getα(){
16    return α;
17  }
18
19  protected void registerα(T α)
20                              throws IllegalArgumentException{
21    if (!getClassObject().isProperValueForα(α))
22      throw new IllegalArgumentException();
23    this.α = α;
24  }
25
26  public boolean canHaveAsα(T α){
27    if (!getClassObject().isProperValueForα(α))
28      return false;
29    if (!getClassObject().isProperαδ(α,getδ()))
30      return false;
31    return true;
```

```
32    }
33
34    public boolean canHaveAsNewα(T α){
35      if (!canHaveAsα(α))
36        return false;
37      return ...;
38    }
39
40    public final boolean hasProperα(){
41      return canHaveAsα(getα());
42    }
43
44    public void setα(T α) throws IllegalArgumentException{
45      if (!canHaveAsNewα(α))
46        throw new IllegalArgumentException();
47      registerα(α);
48    }
49
50    /*******************************************************
51     *                    Property δ                        *
52     *******************************************************/
53
54
55    private R δ;
56
57    @Basic
58    public R getδ(){
59      return δ;
60    }
61
62    protected void registerδ(R δ)
63                              throws IllegalArgumentException {
64      if (!getClassObject().isProperValueForδ(δ)
65        throw new IllegalArgumentException();
66      this.δ = δ;
67    }
68
69    public boolean canHaveAsδ(R δ){
70      if (!getClassObject().isProperValueForδ(δ);
71        return false;
72      if (!getClassObject().isProperαδ(getα(),δ))
73        return false;
74      return true;
75    }
76
77    public boolean canHaveAsNewδ(R δ){
78      if (!canHaveAsδ(δ)
```

```
79            return false;
80          return ...;
81        }
82
83        public final boolean hasProperδ(){
84          return canHaveAsδ(getδ())
85        }
86
87        public void setδ(R δ) throws IllegalArgumentException{
88          if (!canHaveAsNewδ(δ))
89            throw new IllegalArgumentException();
90          registerδ(δ);
91        }
92
93        public  COFoo getClassObject(){
94          return COFoo().getInstance();
95        }
96
97        /*********************************************************
98         *                      Class Methods                    *
99         *********************************************************/
100
101
102       public static class COFoo implements ClassObject{
103
104         private static COFoo instance;
105
106         protected COFoo(){}
107
108         public static COFoo getInstance(){
109           if (instance == null)
110             instance = new COFoo();
111           return instance;
112         }
113
114         public boolean isProperValueForα(T α){
115           return ...;
116         }
117
118         public boolean isProperValueForδ(R δ){
119           return ...;
120         }
121
122         public boolean isProperαδ(T α, R δ){
123           if (!isProperValueForα(α))
124             return false;
```

```
125         if (!isProperValueForδ(δ))
126           return false;
127         return ..;
128       }
129     }
130
131 }
```

Listing A.1: The pattern for a properties α and δ

Appendix B

Unidirectional Association Pattern

```
 1 public class Foo {
 2
 3   public Foo(T α) throws IllegalArgumentException{
 4     registerα(α);
 5   }
 6
 7   public  COFoo getClassObject(){
 8     return COFoo.getInstance();
 9   }
10
11   /*******************************************************
12    *                        Property α                    *
13    *******************************************************/
14
15   private T α;
16
17   @Basic
18   public T getα(){
19     return α;
20   }
21
22   protected void registerα(T α)
23                             throws IllegalArgumentException{
24     if (!getClassObject().isProperValueForα(α))
25       throw new IllegalArgumentException();
26     this.α = α;
27   }
28
```

```
29  public boolean canHaveAsα(T α){
30    if (!getClassObject().isProperValueForα(α))
31      return false;
32    if (!getClassObject().isProperαBar(α, getBar())
33      return false;
34    return true;
35  }
36
37  public boolean canHaveAsNewα(T α){
38    if (!canHaveAsα(α))
39      return false;
40    return ..;
41  }
42
43  public final boolean hasProperα(){
44    return canHaveAsα(getα());
45  }
46
47  public void setα(T α) throws IllegalArgumentException{
48    if (!canHaveAsNewα(α))
49      throw new IllegalArgumentException();
50    registerα(α);
51  }
52
53  /*******************************************************
54   *            Unidirectional Association Bar          *
55   *******************************************************/
56   private Bar bar;
57
58   @Basic
59   public Bar getBar(){
60    return bar;
61   }
62
63  protected void registerBar(Bar bar)
64                              throws IllegalArgumentException{
65    if (!getClassObject().isProperValueForBar(bar))
66      throw new IllegalArgumentException();
67    this.bar = bar;
68  }
69
70  public boolean canHaveAsBar(Bar bar){
71    if (!getClassObject().isProperValueForBar(bar))
72      return false;
73    if (!getClassObject().isProperαBar(getα(), bar)
74      return false;
75    return true;
```

```
76  }
77
78  public boolean canHaveAsNewBar(Bar bar){
79    if (!canHaveAsBar(bar))
80      return false;
81    return ..;
82  }
83
84  public final boolean hasProperBar(){
85    return canHaveAsBar(getBar());
86  }
87
88  public void setBar(Bar bar) throws IllegalArgumentException{
89    if (!canHaveAsNewBar(bar))
90      throw new IllegalArgumentException();
91    registerBar(bar);
92  }
93
94  /*******************************************************
95   *                   Class Methods                     *
96   *******************************************************/
97
98  public static class COFoo implements ClassObject{
99
100   private static COFoo instance;
101
102   protected COFoo(){}
103
104   public static COFoo getInstance(){
105     if (instance == null)
106       instance = new COFoo();
107     return instance;
108   }
109
110   public boolean isProperValueForα(T α){
111     return ...;
112   }
113
114   public boolean isProperαBar(T α, Bar bar){
115     if (!isProperValueForα(α))
116       return false;
117     if (!isProperValueForBar(bar))
118       return false;
119     return matchesαBar(α,bar);
120   }
121
```

```
122      protected boolean matchesαBar(T α, Bar bar){
123        return ...;
124      }
125    }
126  }
```

Listing B.1: The pattern for a unidirectional association Foo referring Bar: class Foo

```
1 public class Bar {
2
3   public Bar(S β) throws IllegalArgumentException{
4     registerβ(β);
5   }
6
7   /*******************************************************
8    *                      Property β                     *
9    *******************************************************/
10
11  private S β;
12
13  @Basic
14  public S getβ(){
15    return β;
16  }
17
18  protected void registerα(S β)
19                            throws IllegalArgumentException{
20    if (!getClassObject().isProperValueForα(β))
21      throw new IllegalArgumentException();
22    this.β = β;
23  }
24
25  public boolean canHaveAsβ(S β){
26    if (!getClassObject().isProperValueForβ(β))
27      return false;
28    return true;
29  }
30
31  public boolean canHaveAsNewβ(S β){
32    if (!canHaveAsβ(β))
33      return false;
34    return ...;
35  }
36
37  public final boolean hasProperβ(){
38    return canHaveAsβ(getβ());
39  }
40
41  public void setβ(S β) throws IllegalArgumentException{
42    if (!canHaveAsNewβ(β))
43      throw new IllegalArgumentException();
44    registerβ(β);
45  }
46
47  public  COBar getClassObject(){
```

```
48      return COBar().getInstance();
49    }
50
51    /********************************************************
52     *                    Class Methods                     *
53     ********************************************************/
54
55    public static class COBar implements ClassObject{
56
57      private static COBar instance;
58
59      protected COBar(){}
60
61      public static COBar getInstance(){
62        if (instance == null)
63          instance = new COBar();
64        return instance;
65      }
66
67      public boolean isProperValueForβ(S β){
68        return ...;
69      }
70    }
71  }`
```

Listing B.2: The pattern for a unidirectional association Foo referring Bar: class Bar

Appendix C

Bidirectional Association Pattern

```
 1 public class Foo {
 2
 3   public Foo(T α) throws IllegalArgumentException{
 4     registerα(α);
 5   }
 6
 7   public  COFoo getClassObject(){
 8     return COFoo.getInstance();
 9   }
10
11   /*******************************************************
12    *                    Property α                       *
13    *******************************************************/
14
15   private T α;
16
17   @Basic
18   public T getα(){
19     return α;
20   }
21
22   protected void registerα(T α)
23                             throws IllegalArgumentException{
24     if (!getClassObject().isProperValueForα(α))
25       throw new IllegalArgumentException();
26     this.α = α;
27   }
28
```

```
29   public boolean canHaveAsα(T α){
30     if (!getClassObject().isProperValueForα(α))
31       return false;
32     if (!getClassObject().isProperαBar(α, getBar())
33       return false;
34     if (   hasBar() &&
35          (!getClassObject().isProperαβ(α, getBar().getβ(),
36                                  getBar().getClassObject())  )
37       return false;
38     return true;
39   }
40
41   public boolean canHaveAsNewα(T α){
42     if (!canHaveAsα(α))
43       return false;
44     return ..;
45   }
46
47   public final boolean hasProperα(){
48     return canHaveAsα(getα());
49   }
50
51   public void setα(T α) throws IllegalArgumentException{
52     if (!canHaveAsNewα(α))
53       throw new IllegalArgumentException();
54     registerα(α);
55   }
56
57   /*******************************************************
58    *           Bidirectional Association Bar           *
59    *           (Controlling class)                     *
60    *******************************************************/
61    private Bar bar;
62
63    @Basic
64    public Bar getBar(){
65     return bar;
66    }
67
68    public boolean hasBar(){
69     return getBar() != null;
70    }
71
72   protected void registerBar(Bar bar)
73                        throws IllegalArgumentException{
74     if (!getClassObject().isProperValueForBar(bar))
75       throw new IllegalArgumentException();
```

```
76       this.bar = bar;
77     }
78
79     public boolean canHaveAsBar(Bar bar){
80       if (!getClassObject().isProperValueForBar(bar))
81         return false;
82       if (!getClassObject().isProperαBar(getα(), bar)
83         return false;
84       if (bar != null){
85         if (!getClassObject().isProperαβ(getα(), bar.getβ(),
86                                         bar.getClassObject()))
87           return false;
88       }
89
90       return true;
91     }
92
93     public boolean canHaveAsNewBar(Bar bar){
94       if (!canHaveAsBar(bar))
95         return false;
96       return ..;
97     }
98
99     public final boolean hasProperBar(){
100      return  canHaveAsBar(getBar()) &&
101              ((getBar()==null) || (getBar().getFoo()==this));
102
103    }
104
105  public final boolean canHaveAsNewFooBar(Bar bar){
106    if (!canHaveAsNewBar(bar))
107      return false;
108    if ((bar != null) && !bar.canHaveAsNewFoo(this))
109      return false;
110
111    if (bar != null){
112      if (   bar.hasFoo() &&
113             (!bar.getFoo().canHaveAsNewBar(getBar()))  )
114        return false;
115      if (hasBar() && (!getBar().canHaveAsNewFoo(bar.getFoo())))
116        return false;
117      }
118    return true;
119  }
120
121  public final void setBar(Bar bar)
122                                  throws IllegalArgumentException{
```

```
123   if (!canHaveAsNewFooBar(bar))
124     throw new IllegalArgumentException();
125   Bar oldbarFromThis = getBar();
126   Foo oldFooFromArgument = bar!=null?bar.getFoo():null;
127   registerBar(bar);
128   if (bar != null)
129     bar.registerFoo(this);
130   if (oldFooFromArgument != null)
131     oldFooFromArgument.registerBar(oldBarFromThis);
132   if (oldBarFromThis != null)
133     oldBarFromThis.registerFoo(oldFooFromArgument);
134 }
135
136 public final boolean canHaveAsNewFooBar_2(Bar Bar){
137   if (!canHaveAsNewBar(bar))
138     return false;
139   if ((bar != null) && !bar.canHaveAsNewFoo(this))
140     return false;
141
142   if ((foo!=null) && (foo.hasBar()))
143     if (!foo.getBar().canHaveAsNewFoo(null))
144       return false;
145
146   if (getBar()!=null)
147     if (!getBar().canHaveAsNewFoo(null))
148       return false;
149   return true;
150 }
151
152 public final void setBar_2(Bar bar)
153                             throws IllegalArgumentException{
154   if (!canHaveAsNewFooBar(bar))
155     throw new IllegalArgumentException();
156   Bar oldbarFromThis = getBar();
157   Foo oldFooFromArgument = bar!=null?bar.getFoo():null;
158   registerBar(bar);
159   if (bar != null)
160     bar.registerFoo(this);
161   if (oldFooFromArgument != null)
162     oldFooFromArgument.registerBar(null);
163   if (oldBarFromThis != null)
164     oldBarFromThis.registerFoo(null);
165 }
166   /**********************************************************
167    *                       Class Methods                     *
168    **********************************************************/
169
```

```
170   public static class COFoo implements ClassObject{
171
172     private static COFoo instance;
173
174     protected COFoo(){}
175
176     public static COFoo getInstance(){
177       if (instance == null)
178         instance = new COFoo();
179       return instance;
180     }
181
182     public boolean isProperValueForα(T α){
183       return ...;
184     }
185
186     public final boolean isProperValueForBar_Foo( Bar bar,
187                                                   Foo foo){
188       if  ((bar == null) && (foo == null))
189           return false;
190       return  ( (bar == null) ||
191                 bar.getClassObject().isProperValueForFoo(foo) )
192                &&
193               ( (foo == null) ||
194                 foo.getClassObject().isProperValueForBar(bar) );
195     }
196
197     public boolean isProperValueForBar(Bar bar){
198       return ...;
199     }
200
201     public boolean isProperαBar(T α, Bar bar){
202       if (!isProperValueForα(α))
203         return false;
204       if (!isProperValueForBar(bar))
205         return false;
206       return matchesαBar(α,bar);
207     }
208
209     protected boolean matchesαBar(T α, Bar bar){
210       return ...;
211     }
212
213     public final boolean isProperαβ(T α, S β,
214                                    Bar.COBar classObject){
215       if (!isProperValueForα(α))
```

```
216           return false;
217         if (!classObject.isProperValueForβ(β))
218           return false;
219         return matchesαβ(α,β,classObject) &&
220                 classObject.matchesαβHelper(α,this,β);
221       }
222
223       protected boolean matchesαβ(T α, S β,
224                                       Bar.COBar classObject){
225         return ...;
226       }
227     }
228 }
```

Listing C.1: The pattern for an association Foo_Bar

```
1 public class Bar {
2
3   public Bar(S β) throws IllegalArgumentException{
4     registerβ(β);
5   }
6
7   /*********************************************************
8    *                         Property β                    *
9    *********************************************************/
10
11  private S β;
12
13  @Basic
14  public S getβ(){
15    return β;
16  }
17
18  protected void registerα(S β)
19                            throws IllegalArgumentException{
20    if (!getClassObject().isProperValueForα(β))
21      throw new IllegalArgumentException();
22    this.β = β;
23  }
24
25  public boolean canHaveAsβ(S β){
26    if (!getClassObject().isProperValueForβ(β))
27      return false;
28    if (   hasFoo() &&
29          (!getFoo().getClassObject().
30            isProperαβ(getFoo().getα(), β, getClassObject())) )
31        return false;
32    return true;
33  }
34
35  public boolean canHaveAsNewβ(S β){
36    if (!canHaveAsβ(β))
37      return false;
38    return ...;
39  }
40
41  public final boolean hasProperβ(){
42    return canHaveAsβ(getβ());
43  }
44
45  public void setβ(S β) throws IllegalArgumentException{
46    if (!canHaveAsNewβ(β))
47      throw new IllegalArgumentException();
```

```
48      registerβ(β);
49    }
50
51    public  COBar getClassObject(){
52      return COBar().getInstance();
53    }
54
55    /*******************************************************
56     *              Bidirectional Association Foo           *
57     *******************************************************/
58     private Foo foo;
59
60     @Basic
61     public Foo getFoo(){
62      return foo;
63     }
64
65     public boolean hasFoo(){
66      return getFoo() != null;
67     }
68
69    protected void registerFoo(Foo foo)
70                              throws IllegalArgumentException{
71      assert  foo == null || foo.getBar() == this;
72      assert  foo !=null || !this.hasFoo() ||
73              getFoo().getBar()!=this;
74      if (!getClassObject().isProperValueForFoo(foo))
75        throw new IllegalArgumentException();
76      this.foo = foo;
77    }
78
79    public boolean canHaveAsFoo(Foo foo){
80      if (!getClassObject().isProperValueForFoo(foo))
81        return false;
82      if (foo != null){
83          if (! foo.getClassObject().isProperαβ(foo.getα(),
84                                         getβ(), getClassObject()))
85            return false;
86        }
87      return true;
88    }
89
90    public boolean canHaveAsNewFoo(Foo foo){
91      if (!canHaveAsFoo(foo))
92        return false;
93      return ..;
94    }
```

```
 95
 96   public final boolean hasProperFoo(){
 97     return  canHaveAsFoo(getFoo()) &&
 98              ((getFoo()==null) || (getFoo().getBar()==this));
 99
100   }
101
102 public final boolean canHaveAsNewFooBar(Foo foo){
103   if (foo!=null)
104     return foo.canHaveAsNewFooBar(this);
105   else
106     if (getFoo() != null)
107       return getFoo().canHaveAsNewFooBar(null);
108     else
109       return canHaveAsNewFoo(null);
110 }
111
112 public final void setFoo(Foo foo)
113                               throws IllegalArgumentException{
114   if (!canHaveAsNewFooBar(foo))
115     throw new IllegalArgumentException();
116   if (foo != null)
117     foo.setBar(this);
118   else
119     if (getFoo() != null)
120       getFoo().setBar(null);
121 }
122
123 public final boolean canHaveAsNewFooBar_2(Foo foo){
124   if (foo!=null)
125     return foo.canHaveAsNewFooBar_2(this);
126   else
127     if (getFoo() != null)
128       return getFoo().canHaveAsNewFooBar_2(null);
129     else
130       return canHaveAsNewFoo(null);
131 }
132
133 public final void setFoo_2(Foo foo)
134                               throws IllegalArgumentException{
135   if (!canHaveAsNewFooBar(foo))
136     throw new IllegalArgumentException();
137   if (foo != null)
138     foo.setBar_2(this);
139   else
140     if (getFoo() != null)
141       getFoo().setBar_2(null);
```

```
142 }
143
144   /*******************************************************
145    *                      Class Methods                   *
146    *******************************************************/
147
148   public static class COBar implements ClassObject{
149
150     private static COBar instance;
151
152     protected COBar(){}
153
154     public static COBar getInstance(){
155       if (instance == null)
156         instance = new COBar();
157       return instance;
158     }
159
160     public boolean isProperValueForβ(S β){
161       return ...;
162     }
163
164     public boolean isProperValueForFoo(Foo foo){
165       return ...;
166     }
167
168     protected boolean matchesαβHelper(T α,
169                                       Foo.COFoo classObject,
170                                       S β){
171       return ...;
172     }
173   }
174 }
```

Listing C.2: The pattern for an association Foo_Bar

Bibliography

[1] Object Management Group. http://www.omg.org/. [Online; accessed 01-December-2013].

[2] JSR 296: Swing Application Framework. https://jcp.org/en/jsr/detail?id=296, 1998. [Online; accessed 01-December-2013].

[3] JSR 41: A Simple Assertion Facility. https://www.jcp.org/en/jsr/detail?id=41, 2002. [Online; accessed 01-December-2013].

[4] JSR 59: J2SETM Merlin Release Contents. https://www.jcp.org/en/jsr/detail?id=59, 2002. [Online; accessed 01-December-2013].

[5] BPEL4WS Language Specification. http://www.ibm.com/developerworks, 2003. [Online; accessed 22-july-2007].

[6] OMG UML Specification 1.5. http://www.omg.org/technology/documents/, 2003. [Online; accessed 22-july-2007].

[7] JSR 14: Add Generic Types To The JavaTM Programming Language. https://www.jcp.org/en/jsr/detail?id=14, 2004. [Online; accessed 01-December-2013].

[8] JSR 176: J2SETM 5.0 (Tiger) Release Contents. https://www.jcp.org/en/jsr/detail?id=176, 2004. [Online; accessed 01-December-2013].

[9] JSR 201: Extending the JavaTM Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import. https://www.jcp.org/en/jsr/detail?id=201, 2004. [Online; accessed 01-December-2013].

[10] JSR 221: JDBCTM 4.0 API Specification. https://jcp.org/en/jsr/detail?id=221, 2006. [Online; accessed 01-December-2013].

[11] The Meta Object Facility Core Specification 2.0. http://www.omg.org/technology, 2006. [Online; accessed 22-july-2007].

[12] UML 2.0 OCL specification (OMG final adopted specification). http://www.omg.org/docs/ptc/03-10-14.pdf, 2006. [Online; accessed 01-march-2006].

[13] Unified Modeling Language specification, version 1.4.2). http://www.omg.org/docs/formal/04-07-02.pdf, 2006. [Online; accessed 01-march-2006].

[14] The Promise, the Limits, and the Beauty of Software. https://www.cs.man.ac.uk/aboutus/events/Turing/07-Grady-Booch/, 2007. [Online; accessed 07-October-2013].

[15] Object Constraint Language (OCL) version 2.3.1. http://www.omg.org/spec/OCL/2.3.1, 2012. [Online; accessed 01-december-2013].

[16] Popular Mechanics. http://www.popularmechanics.com/technology/engineering/news/inside-the-future-how-popmech-predicted-the-next-110-years-14831802, 2012. [Online; accessed 07-October-2013].

[17] AndroMDA. http://www.andromda.org/index.html, 2013. [Online; accessed 11-Februari-2013].

[18] C# Programming Guide. http://msdn.microsoft.com/en-us/library/vstudio/67ef8sbd.aspx, 2013. [Online; accessed 22-August-2013].

[19] Contract4J. http://www.polyglotprogramming.com/contract4j, 2013. [Online; accessed 22-August-2013].

[20] Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/?project=emf, 2013. [Online; accessed 11-Februari-2013].

[21] Eclipse Project. http://www.eclipse.org/, 2013. [Online; accessed 22-August-2013].

[22] Eiffel Software. http://www.eiffel.com, 2013. [Online; accessed 22-August-2013].

[23] Enterprise JavaBeans Technology. http://www.oracle.com/technetwork/java/javaee/ejb/index.html, 2013. [Online; accessed 22-August-2013].

[24] Java. http://www.java.com, 2013. [Online; accessed 11-Februari-2013].

[25] Javadoc Tool Home Page. http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html, 2013. [Online; accessed 22-August-2013].

[26] JSR 335: Lambda Expressions for the JavaTM Programming Language. https://www.jcp.org/en/jsr/detail?id=335, 2013. [Online; accessed 01-December-2013].

[27] MATLAB The Language of Technical Computing. http://www.mathworks.nl/products/matlab/index.html, 2013. [Online; accessed 11-Februari-2013].

[28] MDA technology - Model-driven engineering. http://www.modeliosoft.com/en/technologies/mda.html, 2013. [Online; accessed 11-Februari-2013].

[29] MDA Tool for Model Driven Architecture. http://www.sparxsystems.com.au/platforms/mda_tool.html, 2013. [Online; accessed 11-Februari-2013].

[30] Metrics. http://metrics.sourceforge.net/, 2013. [Online; accessed 11-Februari-2013].

[31] Microsoft Research Spec#. http://research.microsoft.com/en-us/projects/specsharp/, 2013. [Online; accessed 22-August-2013].

[32] Object Constraint Language (OCL). http://www.omg.org/spec/OCL, 2013. [Online; accessed 01-December-2013].

[33] OMG's MetaObject Facility. http://www.omg.org/mof/, 2013. [Online; accessed 01-December-2013].

[34] Sather. http://www1.icsi.berkeley.edu/~sather/, 2013. [Online; accessed 22-August-2013].

[35] The Nice Programming Language. http://nice.sourceforge.net/, 2013. [Online; accessed 22-August-2013].

[36] UML Resource Page. http://www.uml.org/, 2013. [Online; accessed 01-December-2013].

[37] Industrial Electronics, IEEE Transactions on. http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=4180444, Issue 2, May 1983. [Online; accessed 07-October-2013].

[38] CHIPS magazine. http://www.doncio.navy.mil/CHIPS/Issue.aspx?ID=39, October-December 2002 Issue. [Online; accessed 03-October-2013].

[39] R. J. Abbott. Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894, Nov. 1983.

[40] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. OUP USA, 1977.

[41] A. Ampatzoglou, S. Charalampidou, and I. Stamelos. Research state of the art on gof design patterns: A mapping study. *Journal of Systems and Software*, 86(7):1945–1964, 2013.

[42] J. Arlow and I. Neustadt. *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2Nd Edition)*. Addison-Wesley Professional, 2005.

[43] K. Arnout. *From Patterns to Components*. PhD thesis, 2004.

[44] S. V. Baelen, J. Lewi, E. Steegmans, H. V. Riel, S. V. Baelen, J. Lewi, E. Steegmans, and H. V. Riel. Eroos: An entity-relationship based object-oriented specification method. In *Technology of Object-Oriented Languages and Systems TOOLS 7*, pages 103–117. Prentice Hall, Hertsfordshire, UK, 1992.

[45] S. Bagui and R. Earp. *Database Design Using Entity-Relationship Diagrams, Second Edition*. Auerbach Publications, Boston, MA, USA, 2nd edition, 2011.

[46] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[47] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001.

[48] P. Bekaert. *Behavioral Semantics for EROOS Conceptual Modeling: Separation of Concerns Through Nondeterminism.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, June 2006. Steegmans, Eric (supervisor).

[49] P. Bekaert, G. Delanote, F. Devos, and E. Steegmans. Specialization/-generalization in object-oriented analysis: strengthening and multiple partitioning. In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems,*, pages 34–43, 2002.

[50] P. Bekaert and E. Steegmans. Non-determinism in conceptual models. In K. Baclawski and H. Kilov, editors, *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics,*, pages 24 – 34, 2001.

[51] P. Bekaert, B. Van Nuffelen, M. Bruynooghe, D. Gilis, and M. Denecker. On the transformation of object-oriented conceptual models to logical theories. In *Lecture Notes in Computer Science,*, pages 152–166. Springer, 2002.

[52] P. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[53] J. Bishop. Language features meet design patterns: raising the abstraction bar. In *Proceedings of the 2nd international workshop on The role of abstraction in software engineering*, ROA '08, pages 1–7, New York, NY, USA, 2008. ACM.

[54] J. Bloch. *Effective Java Programming Language Guide.* Sun Microsystems, Inc., Mountain View, CA, USA, 2001.

[55] J. Bloch. *Effective Java (2nd Edition) (The Java Series).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[56] G. Booch. *Object-oriented Analysis and Design with Applications (2Nd Ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[57] J. Bosch. Design patterns as language constructs. *JOOP*, 11(2):18–32, 1998.

[58] J. Boydens. *Location Transparency and Transactions as First-Class Concepts in Object-Oriented Programming Languages.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, May 2008. Steegmans, Eric (supervisor).

[59] J. Boydens and E. Steegmans. Model Driven Architecture: The next abstraction level in programming. In L. De Backer, editor, *Proceedings of the First European Conference on the Use of Modern Information and Communication Technologies,*, pages 97–104, 2004.

[60] F. P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, Apr. 1987.

[61] M. Broy, I. Krüger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2), 2007.

[62] F. Budinsky, M. Finnie, P. Yu, and J. Vlissides. Automatic code generation from design patterns. *IBM Systems Journal*, pages 151–171, 1996.

[63] M. Campbell-Kelly and W. Aspray. *Computer: a history of the information machine.* Basic Books, Inc., New York, NY, USA, 1996.

[64] C. Chambers, B. Harrison, and J. Vlissides. A debate on language and tool support for design patterns. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 277–289, New York, NY, USA, 2000. ACM.

[65] R. N. Charette. IEEE Spectrum. `http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code`, 2009. [Online; accessed 07-October-2013].

[66] A. Clark. Metaclasses and reflection in smalltalk, 1997.

[67] T. Clark and P.-A. Muller. Exploiting model driven technology: A tale of two startups. *Softw. Syst. Model.*, 11(4):481–493, Oct. 2012.

[68] O.-J. Dahl and K. Nygaard. Simula: An algol-based simulation language. *Commun. ACM*, 9(9):671–678, Sept. 1966.

[69] D. de Champeaux, P. America, D. Coleman, R. Duke, D. Lea, G. T. Leavens, and F. Hayes. Formal techniques for oo software development (panel). In A. Paepcke, editor, *OOPSLA*, pages 166–170. ACM, 1991.

[70] S. De Labey, G. Delanote, K. Vanderkimpen, and E. Steegmans. A framework for executing cross-model transformations based on pluggable metamodels. `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW489.pdf`, 2007. [Online; accessed 03-September-2013].

[71] S. De Labey, M. van Dooren, and E. Steegmans. ServiceJ. A type system extension for programming web service interactions. In L. Zhang, K. Birman, and J. Zhang, editors, *Proceedings of the Fifth IEEE International Conference on Web Services,*, pages 505–512, 2007.

[72] G. Delanote, J. Boydens, and E. Steegmans. A pattern-based approach towards expressive specifications for property concepts. In L. Lavazza, R. Oberhauser, A. Martin, J. Hassine, M. Gebhart, and M. Jäntti, editors, *ThinkMind // ICSEA 2013, The Eighth International Conference on Software Engineering Advances,*, pages 249–257, 2013.

[73] G. Delanote, S. De Labey, K. Vanderkimpen, and E. Steegmans. A framework for executing cross-model transformations based on pluggable metamodels. In *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT07),*, pages 315–325, 2007.

[74] G. Delanote and E. Steegmans. Concepts for abstracting away object reification at the level of platform independent models (PIMs). In R. Machado, J. Fernandes, M. Riebisch, and B. Schätz, editors, *Proceedings of The Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software,*, pages 94–102, 2006.

[75] F. Devos. *Patterns and Anti-Patterns in Object-Oriented Analysis.* PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2004. Steegmans, Eric (supervisor).

[76] F. Devos and E. Steegmans. The Message Paradigm in Object-oriented Analysis. In G. M. and K. C., editors, *ML 2001 - The Unified Modeling Language,*, pages 182–193, 2001.

[77] F. Devos and E. Steegmans. Meta-model patterns in object-oriented analysis. In *14th Information Resource Management Association International Conference,*, 2003.

[78] F. Devos and E. Steegmans. Specifying business rules in object-oriented analysis. *Software and Systems Modeling*, 4(3):297–309, July 2005.

[79] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, Oct. 1972.

[80] T. Dingsøyr, S. Nerur, V. Balijepally, and N. B. Moe. A decade of agile methodologies: Towards explaining agile software development. *J. Syst. Softw.*, 85(6):1213–1221, June 2012.

[81] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[82] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[83] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[84] J. Gil and D. H. Lorenz. Design patterns and language design. *IEEE Computer*, 31(3):118–120, 1998.

[85] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[86] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[87] F. Hayes and D. Coleman. Coherent models for object-oriented analysis. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 171–183, New York, NY, USA, 1991. ACM.

[88] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Scaling-up model-based-development for large heterogeneous systems with compositional modeling. In *Software Engineering Research and Practice*, pages 172–176, 2009.

[89] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[90] R. Hyde. *The Art of Assembly Language.* No Starch Press, San Francisco, CA, USA, 2003.

[91] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[92] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[93] M. Karaorman, U. Hölzle, and J. L. Bruno. jcontractor: A reflective java library to support design by contract. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 175–196, London, UK, UK, 1999. Springer-Verlag.

[94] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[95] M. Kuznetsov. Automated model transformation in mda. In *In Colloquium on Database and Information Systems*, 2005.

[96] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, Upper Saddle River, NJ, 1998.

[97] G. T. Leavens and Y. Cheon. Design by contract with jml. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`, 2006. [Online; accessed 22-August-2013].

[98] S. B. Lippman, J. Lajoie, and B. E. Moo. *C++ Primer.* Addison-Wesley Professional, 5th edition, 2012.

[99] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[100] C. Mann. Why software is so bad...and what's being done to fix it. MIT Technology Rev., vol. 105, pp. 33-38, 2002.

[101] R. C. Martin. More c++ gems. chapter The Open-closed Principle, pages 97–112. Cambridge University Press, New York, NY, USA, 2000.

[102] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[103] T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformation. In *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005.

[104] B. Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[105] B. Meyer. Lecture 7: Patterns, observer, mvc. `http://se.inf.ethz.ch/old/teaching/ss2007/0050/slides/07_softarch_patterns_observer_3up.pdf`, 2007. [Online; accessed 09-september-2013].

[106] R. Mitchell, J. McKim, and B. Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[107] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[108] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[109] Object Management Group. ptc/02-09-02: UML 1.5 – Action Semantics, 2002.

[110] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[111] O. Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.

[112] M. Peltier, J. Bezivin, and G. Guillaume. Mtrans: A general framework based on xslt for model transformations. WTUML01, Proceedings of the Workshop on Transformations in UML, 2001.

[113] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, June 2002.

[114] W. Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[115] B. Randall. The 1968/69 nato software engineering reports. dagstuhl-seminar 9635: History of software engineering. Schloss Dagstuhl, Germany, August 26–30, 1996.

[116] A. Rensink. Subjects, models, languages, transformations. In *Dagstuhl Seminar Proceedings (04101)*, pages 1–13, 2005.

[117] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON*. IEEE Press, August 1970. Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338.

[118] A. L. Rubinger and B. Burke. *Enterprise JavaBeans 3.1 (6. ed.)*. O'Reilly, 2010.

[119] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[120] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[121] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Boston, MA, 2 edition, 2005.

[122] J. Said. *Pattern-based approach for object-oriented software design*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Sept. 2003. Steegmans, Eric (supervisor).

[123] B. Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, Sept. 2003.

[124] S. Sendall, G. Perrouin, N. Guelfi, and O. Biberstein. O.: Supporting model-tomodel transformations: The vmt approach. Technical report, University of Twente, 2003.

[125] J. Siegel. Developing in OMGs Model-Driven Architecture. http://www.cin.ufpe.br/~redis/mda/01-12-01.pdf, 2013. [Online; accessed 03-September-2013].

[126] I. Sommerville. *Software Engineering 9.* Pearson Education, 2011.

[127] E. Steegmans. *Object Oriented Programming with Java.* Acco, 2011.

[128] E. Steegmans, P. Bekaert, F. Devos, G. Delanote, N. Smeets, M. van Dooren, and J. Boydens. Black & White Testing: Bridging Black Box Testing and White Box Testing. In P. Sterck, editor, *Software Testing: Beheers Optimaal de Risico's van IT in Uw Business,*, pages 1–12, 2004.

[129] E. Steegmans, J. Lewi, S. De Backer, J. Dockx, B. Swennen, and S. Van Baelen. Object oriented software development with EROOS: The analysis phase - reference manual version 1.1. EROOS Reference Manual 1.1, Department of Computer Science, K.U.Leuven, Sept. 1996.

[130] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[131] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel. Using uml action semantics for model execution and transformation. *Inf. Syst.*, 27(6):445–457, Sept. 2002.

[132] The Institute of Electrical and Eletronics Engineers. Ieee standard glossary of software engineering terminology. IEEE Standard, September 1990.

[133] L. Tratt and T. Clark. Model transformations in Converge, October 2003. Workshop in Software Model Engineering (WiSME) 2003.

[134] S. Van Baelen. *A constraint-centric approach for object-oriented conceptual modelling.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Science, May 2007. Steegmans, Eric (supervisor).

[135] M. van Dooren. *Abstractions for improving, creating, and reusing object-oriented programming languages.* PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2007. Steegmans, Eric (supervisor).

[136] M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. *ACM SIGPLAN notices*, 40(10):455–471, October 2005.

[137] M. van Dooren and E. Steegmans. A higher abstraction level using first-class inheritance relations. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 425–449. Springer, 2007.

[138] E. Van Gestel. *Moose:a framework uniting data base modelling, object-orientation and formal specifications, engineeering style*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Mar. 1994. Lewi, Joannes (supervisor).

[139] P. Van Gorp. *Model-driven Development of Model Transformations*. PhD thesis, Dept. of Mathematics and Computer Science, University of Antwerp, 2008.

[140] D. Varró and A. Pataricza. UML Action Semantics for model transformation systems. *Periodica Politechnica*, 2003. In press.

[141] J. Vlissides. *Pattern hatching: design patterns applied*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1998.

[142] Y. Wand and R. Weber. On the ontological expressiveness of information systems analysis and design grammars. *Information Systems Journal*, 3(4):217–237, 1993.

[143] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[144] E. W.Dijkstra. De software crisis, ontstaan en hardnekkigheid. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD829.html`, 1982. [Online; accessed 01-December-2013].

[145] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In A. Moreira, B. Schtz, J. Gray, A. Vallecillo, and P. J. Clarke, editors, *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[146] E. D. Willink. E.d.willink umlx: A graphical transformation language for mda umlx: A graphical transformation language for mda, 2003.

[147] K. Yue. What does it mean to say that a specification is complete. *Proceedings of the Fourth International Workshop on Software Specification and Design*, 1987.

List Of Publications

- P. Bekaert, G. Delanote, F. Devos, and E. Steegmans. Specialization/-generalization in object-oriented analysis: strengthening and multiple partitioning. In J.-M. Bruel and Z. Bellahsene, editors, Advances in Object-Oriented Information Systems, pages 34-43, 2002.
- E. Steegmans, P. Bekaert, F. Devos, G. Delanote, N. Smeets, M. van Dooren, and J. Boydens. Black & White Testing: Bridging Black Box Testing and White Box Testing. In P. Sterck, editor, Software Testing: Beheers Optimaal de Risicos van IT in Uw Business, pages 1-12, 2004.
- G. Delanote and E. Steegmans. Concepts for abstracting away object reification at the level of platform independent models (PIMs). In R. Machado, J. Fernandes, M. Riebisch, and B. Schtz, editors, Proceedings of The Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pages 94-102, 2006.
- S. De Labey, G. Delanote, K. Vanderkimpen, and E. Steegmans. A framework for executing cross-model transformations based on pluggable metamodels. CW Reports CW489, Department of Computer Science, KU Leuven, 2007.
- G. Delanote, S. De Labey, K. Vanderkimpen, and E. Steegmans. A framework for executing cross-model transformations based on pluggable metamodels. In Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT07), pages 315-325, 2007.
- G. Delanote, J. Boydens, and E. Steegmans. A pattern-based approach towards expressive specifications for property concepts. In L. Lavazza, R. Oberhauser, A. Martin, J. Hassine, M. Gebhart, and M. Jäntti, editors, ThinkMind // ICSEA 2013, The Eighth International Conference on Software Engineering Advances, pages 249-257, 2013.

Curriculum Vitae

Geert Delanote was born on December 22th, 1975 in Poperinge (Belgium). He received a Bachelor degree in computer science (Kandidaat in de Informatica) and a Master's degree in computer science (Licentiaat in de Informatica) from the Katholieke Universiteit Leuven (KU Leuven) in Belgium. He graduated cum laude in September 2000 with the thesis "Ontwikkeling van een interpretator en simulator voor EROOS specificaties", supervised by Prof. dr. ir. Eric Steegmans. He started working as a Ph.D. student at the Software Development Methodology research group at the Department of Computer Science at the KU Leuven in October 2000.

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
IMINDS - DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
http://distrinet.cs.kuleuven.be