DISS. ETH NO. 20910

A Seamless Framework for Object-Oriented Persistence in Presence of Class Schema Evolution

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
MARCO PICCIONI
Laurea degree in Mathematics, Università degli Studi di Roma La Sapienza, Italia
Master in Economics, Università Commerciale L. Bocconi, Italia

born
July 7th, 1965

citizen of
Italy

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Harald Gall, co-examiner
Prof. Dr. Martin Robillard, co-examiner

2012

ACKNOWLEDGEMENTS

First and foremost I thank Bertrand Meyer, for hiring me as a research assistant and for giving me the opportunity to do a PhD under his supervision at ETH Zurich. His support and guidance made the work presented in this thesis possible, and had a decisive influence on my formation as a researcher in the past years.

I also thank the co-examiners of this thesis, Harald Gall and Martin Robillard, who kindly made time in their busy schedules to co-referee this work.

Among the present and past members and visitors of the Chair of Software Engineering at ETH Zurich, and without following any particular order, I would like to thank: Manuel Oriol, for helping me out in taking the first steps in research and for the fruitful cooperation that followed; Carlo Alberto Furia and Sebastian Nanz, who kindly agreed to review the thesis draft; Nadia Polikarpova, for being always willing to discuss anything at any time; Bernd Schoeller, for teaching me the "direct approach" to things; Andreas Leitner, for teaching me what "Forschung" means; Cristiano "Coach" Calcagno, for bringing fractional permissions to the masses; Ilinca Ciupa, for being always supportive and willing to help; Michela Pedroni, for being there both as a colleague and as a friend; Stephan Van Staden and Julian Tschannen for the fruitful discussions; Jason Yi Wei for all the help; Marco "Taco" Trudel, for answering all sort of questions and helping me out in so many ways; Benjamin Morandi, for being a nice colleague and office-mate; Claudia Günthart for the patience and the good spirits. And then Martin "Asador" Nordio, Scott "Native Speaker" West, Christian "The Good German" Estler, Andrey "Randy" Rusakov, Max Yu Pei, Lisa Ling Liu, Mischael Schill, Alexey Kolesnichenko, Cristina Pereira, Marie-Helene Nienaltowski, Piotr Nienaltowski, Till Bay, Wolfgang Schwedler, Mei Tang, Volkan Arslan, and Yann Müller (may he rest in peace).

I also thank the students who worked with me during various semester, diploma, and master projects: Matthias Loeu, Peizhu Li, Adrian Hel-

festein, Luc Hansen, Ruihua Jin, Teseo Schneider, Pascal Dufour, Adrian Schmidmeister, Alexandra Hochuli, and Roman Schmocker.

Special thanks to all the participants to the API front-end usability study: Alexander Kogtenkov, Arno Fiva, Michael Steindorfer, Adam Duracz, Tilmann Zäschke, Severin Münger, Adriana Ispas, Daniel Moser, Mattia Gollub, Andrey Rusakov, Valentin Wüstholz, Philipp Gamper, Ilinca Ciupa, Andreas Leitner, Samuel Bryner, Philipp Reist, Michela Pedroni, Florian Köhl, Jiwon Shin, Miriam Tschanen, Jascha Grübel, Nicholas Pleschko, Mischael Schill, Daniel Schweizer, Christian Locher, and Oliver Probst.

I would also like to thank Denise Spicher from the Student Administration Office, for putting up with all my questions and always being efficient with a smile.

In the Global Information Systems Group, particular thanks to Moira Norrie for the useful comments on the PhD research plan, and to Stefania Leone and Tilmann Zätschke for having discussed my TSE paper draft providing valuable feedback.

Within the Eiffel community I wish to thank Emmanuel Stapf, Alexander Kogtenkov, Jocelyn Fiat, Javier Hector, and Thomas Beale for the help and good feedback.

My former colleagues at Sistemi Informativi supported this Swiss idea of mine from the start: Fiorella Lancia, Cristina Alberti, Silvia Brogi, Laura Conti, Giuseppe Di Raimondo, Alessia Fazzi, Daniela Maciariello, Arianna Malato, Rossana Meaggia and Gianluca Rizzo have my thanks as well.

My family has been always supporting me: Rita, Roberto, Fabio, Guido, Nicole, Valerio, Francesco, and Riccardo are all in my heart.

To Franca, a very special person, go my very special thanks.

I finally wish to thank Cecilia, “my sun and stars”, life companion and main supporter, for always being there for me. Without her love and caring I simply could not have first started and then survived this exciting adventure.

CONTENTS

1 Overview **1**
- 1.1 Main results . . . 2
- 1.2 A case of object-oriented schema evolution . . . 2
- 1.3 Structure . . . 7

2 Related Work **9**
- 2.1 The class schema approach . . . 9
- 2.2 The multi-version approach . . . 12
- 2.3 Schema evolution in relational databases . . . 15
- 2.4 Other approaches to schema evolution . . . 16
- 2.5 Summary of contributions with respect to previous work . . 18

3 Evolving Persistent Applications: a Model **21**
- 3.1 Syntax for classes . . . 22
- 3.2 Schema modification operators . . . 23
  - 3.2.1 The attribute not changed schema modification operator . . . 24
  - 3.2.2 The attribute added schema modification operator . 24
  - 3.2.3 The attribute renamed schema modification operator 25
  - 3.2.4 The attribute type changed schema modification operator . . . 25
  - 3.2.5 The attribute removed schema modification operator 25
  - 3.2.6 The Attribute made attached schema modification operator . . . 26
- 3.3 Completeness . . . 27
- 3.4 Concrete transformation syntax and object transformers . . . 27
- 3.5 Heuristics for class schema evolution . . . 29
- 3.6 A measure of robustness to evolution for persistent object-oriented applications . . . 31

4 How Software Evolution Affects Persistence: Empirical Evidence 33
4.1 Persistence-affecting changes in Java 35
4.1.1 Analyzing *java.util* 35
4.1.2 The persistence evolution robustness of `java.util` 39
4.1.3 Analyzing the Apache Tomcat code base 40
4.2 Persistence-affecting changes in Eiffel 41
4.2.1 Eiffel libraries changes 41
4.2.2 Analyzing the EiffelStudio code base 44
4.3 Evaluating the persistence model on the Java and Eiffel repositories 45
4.4 The Evolution of class invariants in connection with persistence 47
4.4.1 An exploratory study on class invariant evolution . . 48
4.4.2 More results on class invariant evolution 53
4.5 Threats to validity 58

5 Tools and Libraries for Evolving Persistent Apps 61
5.1 IDE support 62
5.1.1 Schema evolution IDE support 62
5.1.2 Serialization support: an object browser GUI 67
5.2 Code generation 69
5.3 The persistence library implementation 71
5.3.1 Front-end API 72
5.3.2 The retrieval algorithm 76

6 Evaluating the Design of the Persistence API 79
6.1 Evaluation guidelines 79
6.2 Empirical answers to API questions: example results from previous studies 81
6.3 Research questions 83
6.4 Participants 83
6.5 Study setup 86
6.6 Data collection protocol 90
6.7 Data analysis: API usability tokens elicitation 91
6.7.1 API usability tokens for class *REPOSITORY* 92
6.7.2 API usability tokens for class *QUERY* 93
6.7.3 API usability tokens for class *CRUD_EXECUTOR* 95
6.7.4 API usability tokens for class *CRITERION* 96
6.8 Data Analysis: final questionnaire 98
6.9 Lessons learned 104
6.10 Threats to validity 105

7 Conclusions and Future Work 107
7.1 Tackling the limits of existing approaches 107
7.1.1 Multi-version model 107
7.1.2 Invariant-safe evolution 108
7.1.3 Release-time evolution handling 108
7.2 Conclusions 109
7.3 Future work 110

A Eiffel and Design by Contract 113
A.0.1 Types 113
A.0.2 Information hiding 114
A.0.3 Code organization 114
A.0.4 Terminology 114
A.0.5 Design by Contract 115
A.0.6 Void Safety 116

B The Persistence Library's Backend implementation 117
B.1 The ORM layer: from *REPOSITORY* to *BACKEND* 117
B.1.1 Collection handling 121
B.1.2 Handling object references 123
B.2 The ORM layer: from *BACKEND* to the database 126
B.3 Automatically generating the database schema 128
B.4 Framework support for transactions and errors 130
B.5 Framework extension points 132
B.5.1 Supporting an additional relational database 132
B.5.2 Supporting additional ORM mappings 133
B.5.3 Supporting non-relational databases 135
B.5.4 Cross-implementation extensions 136

C Graphs 137

Abstract

Many object-oriented software systems rely on persistent data. Such systems also evolve over time. These two equally valid needs clash: a system may attempt to retrieve data whose description in the program has changed. A typical scenario involves an object retrieval failure because of an evolution in the corresponding class. Another, more critical one occurs when a retrieval algorithm accepts objects whose class invariants are not valid anymore. This may happen because in most programming languages class invariants are not made explicit in the code. While in the first scenario a runtime failure will typically point out the issue, the second scenario may lead to invalidating the consistency of the whole system.

The research and software development community has mainly been trying to tackle the first scenario above by providing software developers with the possibility of writing transformation functions to adapt old objects to the new classes. This thesis contributes to this effort by devising a software framework that provides a solution to both scenarios described above.

The framework provides a concise model for persistence-affecting class schema modifications, and an infrastructure keeping track of all class versions and system releases, recognizing the class schema modifications statically, that is before they cause retrieval errors. The framework then further supports developers by using heuristics to generate transformation functions. At retrieval time, a version-aware retrieval algorithm leverages the information previously collected, and together with a class invariant validation step handles object retrieval in a safer way. The algorithm is part of a persistence library seamlessly supporting multiple persistence backends.

To establish the relevance of the problem under study and to validate the choice of class schema modifications among the many possible, this thesis presents the results of four empirical studies performed on existing software libraries and applications, written in both Java and Eiffel. These studies suggest that the problem of a semantically consistent class schema

evolution exists, and that the chosen subset of schema modification operators is viable.

This thesis also suggests a measure for the robustness of evolving object-oriented applications that persist their objects. The measure is meant to offer a simple way to evaluate current and future software projects with respect to their ability to retrieve previously stored objects. The proposed measure is computed and evaluated over an existing Java library.

An additional contribution consists in a study of a phenomenon not analyzed before: the evolution of class invariants and its possible effects on object retrieval. The analysis of eight software projects led to the conclusion that while developers do write class invariants, such activities as adding and removing attributes do not lead, as one might expect, to stronger or weaker invariants in term of number of clauses. This suggests that class invariant evolution constitutes one more potential risk to the consistency of the retrieval operations, because for example the newly added attributes' values could be stored and then retrieved without being guarded by a corresponding class invariant clause.

The usability of the framework front-end API is assessed by performing an exploratory study involving 25 object-oriented developers that had never seen the API before. The study results show that while the API can be used for commonplace database access tasks without resorting to external documentation and specific database knowledge, it also has some usability issues that need to be addressed.

In conclusion, this thesis provides a seamless and invariant-safe solution to the problem of class schema evolution of object-oriented software, encompassing the time in which a system is released, the time in which objects are stored, and the time in which object are retrieved.

RIASSUNTO

Molti sistemi software orientati agli oggetti fanno affidamento su dati persistenti. Tali sistemi evolvono anche nel tempo. Questi due aspetti, entrambi legittimi, confliggono, perché un sistema potrebbe leggere dei dati la cui descrizione nel programma è cambiata. Uno scenario tipico vede la lettura di un oggetto fallire per via di una evoluzione nella classe corrispondente. Uno scenario piú critico si ha quando un algoritmo di lettura dati accetta degli oggetti i cui invarianti di classe non sono piú validi. Ciò può accadere perch'e in molti linguaggi di programmazione gli invarianti di classe non sono resi espliciti nel codice. Mentre nel primo scenario un errore a tempo di esecuzione evidenzierà il problema, il secondo scenario potrebbe causare una mancanza di consistenza per l'intero sistema.

La comunità dei ricercatori e degli sviluppatori software ha tentato di risolvere principalmente il primo scenario menzionato sopra dando agli sviluppatori la possibilità di scrivere funzioni di trasformazione con lo scopo di adattare i vecchi oggetti alle nuove classi. Questa tesi contribuisce a questo tentativo proponendo un framework software che fornisce una soluzione per entrambi gli scenari sopra menzionati.

Il framework propone un conciso modello per i cambiamenti nella struttura delle classi che sono rilevanti rispetto alla persistenza, ed una infrastruttura che tiene conto di tutte le versioni di una classe e di tutti i rilasci di un software, identificando i cambiamenti nella struttura delle classi staticamente, cioè prima che causino errori di lettura. Il framework poi supporta ulteriormente gli sviluppatori usando delle euristiche per generare funzioni di trasformazione. Quando è il momento di leggere i dati, un algoritmo che sa come gestire le versioni di una classe utilizza le informazioni raccolte in precedenza, e con uno step di verifica dell'invariante gestisce la lettura degli oggetti in un modo piú sicuro. L'algoritmo fa parte di una libreria di persistenza che supporta senza soluzione di continuità molti back-end.

Per stabilire la rilevanza del problema sotto studio e per validare la scelta dei cambiamenti nella struttura delle classi tra i molti possibili, que-

sta tesi presenta i risultati di quattro studi empirici condotti su applicazioni e librerie software esistenti, scritte sia in Java che in Eiffel. Tali studi suggeriscono che il problema della consistenza nella semantica dell'evoluzione della struttura delle classi esiste, e che il sottoinsieme scelto di operatori dei cambiamenti nella struttura e' in grado di funzionare.

Questa tesi suggerisce anche una misura per la robustezza di applicazioni orientate agli oggetti che evolvono e salvano gli oggetti stessi. La misura intende offrire un modo semplice per valutare progetti software presenti e futuri rispetto alla loro capacità di leggere oggetti salvati in precedenza. La misura proposta è calcolata e valutata su di una libreria Java esistente.

Un contributo aggiuntivo consiste nello studio di un fenomeno non analizzato in precedenza: l'evoluzione di invarianti di classe e i suoi possibili effetti sulla lettura degli oggetti. L'analisi di otto progetti software ha portato alle seguenti conclusioni: gli sviluppatori di fatto scrivono gli invarianti di classe, ed attività come aggiungere o rimuovere attributi non portano, come potremmo aspettarci, ad invarianti piú forti o piú deboli in termini di numero di clausole. Ciò suggerisce che l'evoluzione degli invarianti di classe costituisce un risco potenziale aggiuntivo per la consistenza delle operazioni di lettura, perché per esempio i valori dei nuovi attributi potrebbero essere salvati e poi letti senza essere verificati dalle corrispondenti clausole dell'invariante di classe.

L'usabilità della front-end API del framework è stato valutato tramite uno studio esplorativo che ha coinvolto 25 sviluppatori orientati agli oggetti che non avevano mai visto l'API prima. I risultati dello studio mostrano che mentre l'API può essere usata per effettuare delle comuni operazioni di accesso ad una base di dati senza utilizzare documentazione esterna o presumere specifiche conoscenze di basi di dati, presenta anche dei problemi di usabilità che devono essere affrontati.

In conclusione, questa tesi propone una soluzione al problema della evoluzione della struttura delle classi nel software orientato agli oggetti che è uniforme e sicura rispetto agli invarianti id classe, e che va dal momento in cui un sistema viene rilasciato al momento in cui gli oggetti vengono letti, passando per quando gli oggetti vengono salvati.

LIST OF FIGURES

3.1 Syntax for class definitions of Eiffel programs. 22
3.2 Attribute not changed schema modification operator. 24
3.3 Attribute added schema modification operator. 24
3.4 Attribute renamed schema modification operator. 25
3.5 Attribute type changed schema modification operator. 25
3.6 Attribute removed schema modification operator. 26
3.7 Attribute made attached schema modification operator. . . . 26
3.8 A concrete syntax for transformers. 28
3.9 Object transformers. 29

4.1 Single SMOs vs. all changes and vs. persistence-affecting ones. 37

5.1 EiffelStudio IDE tool integration. 64
5.2 Version selection pop-up. 65
5.3 Schema evolution handler report. 65
5.4 Selecting a release and a class for filtering. 66
5.5 Filtering the attributes of a class. 66
5.6 Starting the object browsing facility integrated into Eiffel-Studio. 68
5.7 Object visualization GUI. 68
5.8 Object browsing facility integrated into EiffelStudio. 69
5.9 Object browsing facility integrated into EiffelStudio. 69
5.10 The main front-end classes and their relations. 72
5.11 The `OBJECT_QUERY [G]` and `CRITERION` class hierarchy structures. 73

B.1 `OBJECT_GRAPH_PART` hierarchy 118
B.2 The Object-to-Relational API important classes. 119
B.3 Inserting and retrieving objects using *RELATIONAL_REPOSITORY*. 120

B.4 An ER-model where a RELATIONAL_COLLECTION_PART with 1:M mapping can be used. 122
B.5 An ER-model where a RELATIONAL_COLLECTION_PART with M:N mapping can be used. 122
B.6 An ER-model where an OBJECT_COLLECTION_PART can be used. 122
B.7 An object graph in main memory. 123
B.8 The previous object graph at depth 1 (grey nodes are not considered). 125
B.9 From BACKEND to the databases. 127
B.10 The ER-Model of the generic database layout. 129
B.11 Partial error class hierarchy. 132
B.12 ER-Model for persons and items. 133

C.1 AutoTest project: weakening and strengthening of class invariant when adding or removing attributes. 138
C.2 EiffelBase project: weakening and strengthening of class invariant when adding or removing attributes. 139
C.3 Eiffel Program Analysis project: weakening and strengthening of class invariant when adding or removing attributes. . 140
C.4 Gobo Kernel project: weakening and strengthening of class invariant when adding or removing attributes. 141
C.5 Gobo Structure project: weakening and strengthening of class invariant when adding or removing attributes. 142
C.6 Gobo Time project: weakening and strengthening of class invariant when adding or removing attributes. 143
C.7 Gobo Utility project: weakening and strengthening of class invariant when adding or removing attributes. 144
C.8 Gobo XML project: weakening and strengthening of class invariant when adding or removing attributes. 145

List of Tables

4.1 Changes found across 5 versions of `java.util` 36
4.2 Persistence-affecting changes across 5 versions of `java.util` 37
4.3 Changes found in 5 versions of `java.util`, by class. 38
4.4 Persistence evolution-robustness (PER) of `ArrayList`. . . . 39
4.5 P-evolution-robustness (PER) of `java.util`. 40
4.6 SMOs found in Tomcat's repository. 41
4.7 SMOs found across 5 versions of Eiffel. 43
4.8 SMOs found across 5 Eiffel versions, per class. 44
4.9 SMOs found in EiffelStudio's repository. 45
4.10 Changes in class invariant clauses found across 5 versions of `EiffelBase`. The last small revision number is justified by an intervening repository migration. 49
4.11 Changes in class invariant clauses found across 5 versions of `EiffelBase`, by class. 50
4.12 Changes in class invariants across consecutive revisions of `EiffelBase`. 51
4.13 Type of clauses occurring in class invariants across 5 versions of `EiffelBase`, by class. 52
4.14 List of Eiffel projects used in the study (Age is in weeks). . . 54
4.15 Invariant changes when adding attributes, by project. 56
4.16 Invariant changes when removing attributes, by project. . . 57
4.17 Attribute change analysis across all Eiffel projects. 58

6.1 Participant pool information. 85
6.2 API usability tokens (AUT) for repository-related issues. . . 93
6.3 API usability tokens (AUT) for query-related issues. 95
6.4 API usability tokens (AUT) for crud_executor-related issues. 97
6.5 API usability tokens (AUT) for criterion-related issues. . . . 98
6.6 Answers to the final questionnaire. 102

6.7 Answers to the questionnaire: experienced group. 103
6.8 Answers to the questionnaire: group with less O-O experience. 103

List of Listings

1.1 BANK_ACCOUNT, version 1. 3
1.2 BANK_ACCOUNT, version 2. 4
1.3 Schema evolution handler for class BANK_ACCOUNT. 6
5.1 Domain class PERSON 73
5.2 Querying a MySQL database 74
5.3 Algorithm for object retrieval. 76
6.1 Class PERSON, whose objects are to be stored. 87
6.2 Class USABILITY_TEST, the experiment starting point. 88
6.3 Constructor of class PREDEFINED_CRITERION. 97
A.1 Object test. 116
B.1 Domain class ITEM 133
B.2 The collection handler for LINKED_LIST 134
B.3 Domain class PERSON extended by the ORM 135

CHAPTER 1

OVERVIEW

Between the time when an object-oriented program writes objects to persistent storage and when another execution retrieves them, the program may have evolved; in particular, classes describing these objects may have changed. This need to fit old objects into new classes is the problem of schema evolution for object-oriented software. While many solutions have been suggested and some are currently used, none has been generally accepted.

The techniques used to cope with the problem in practice rely heavily on manual effort by the developers, who must understand and examine previous class versions and provide conversion code. This approach is not only tedious, but a threat to software reliability, as it usually relies on tolerant retrieval algorithms making questionable decisions about the key issue: how to avoid accepting semantically inconsistent objects into the retrieving system.

The purpose of the present work is to lay a solid foundation for a general, stable solution to the problem of object-oriented class schema evolution.

To achieve its purpose this thesis proposes a conceptual framework modeling class schema evolution of persistent object-oriented applications, and realizes it with a tool integrated into an integrated development environment (IDE), and a persistence library offering seamless access to different kinds of persistence stores.

1.1 Main results

This thesis relies on the concept of *seamless persistence* for evolving object-oriented software. The seamlessness is intended with respect to two dimensions: *time* and *persistence media*.

With respect to time, this thesis proposes a framework that aims at supporting developers starting from when they release new class versions to when they store and retrieve objects in and from persistent stores. In particular, the thesis describes the principles behind invariant-safe class schema evolution.

With respect to persistence media, this thesis aims at providing a framework front-end application programming interface (API) that can uniformly access different kinds of persistent stores, therefore minimizing the prerequisite knowledge about the underlying database technology.

The contributions of this thesis include the following:

- A formal model for atomic changes in a class schema, resulting from object-oriented theory and experience.
- A schema evolution tool integrated into an IDE and implementing the formal model.
- A persistence library integrating the proposed schema evolution approach and featuring seamless access to different kinds of persistence stores.
- A measure of robustness for the evolution of persistent applications, devised to understand to what extent they are able to successfully retrieve previously stored objects.
- Four empirical studies to assess the relevance of class schema evolution as a software engineering problem. The studies cover both Java and Eiffel, and for each language analyze both software libraries and realistic applications.
- A further study to investigate the evolution of the specific form of code-integrated specification that is relevant to persistence: class invariants.

1.2 A case of object-oriented schema evolution

This section presents a motivating example and shows how the persistence framework described in Chapter 5 copes with it. This thesis will use

the Eiffel notation when expressing source code, because it provides embedded support for class invariants, which constitute a cornerstone of the whole approach. The aspects of Eiffel relevant to this thesis are illustrated in Appendix A.

Assume a software system stores objects of type *BANK_ACCOUNT* (see Figure 1.1).

```
class
  BANK_ACCOUNT

create make

feature  -- Initialization

  make
      -- Create a bank account with an initial deposit.
    do
      tot_deposits := 1
    end

feature  -- Status report

  balance: INTEGER
      -- Account balance.
     do
      Result := tot_deposits - tot_withdrawals
     end

  info: INTEGER
      -- Some numeric information.

feature -- Basic operations

  deposit (sum: INTEGER)
      -- Add 'sum' to account.
    require
      sum_positive: sum > 0
    do
      tot_deposits := tot_deposits + sum
    ensure
      balance_correct: balance = old balance + sum
    end
```

```
  withdraw (sum: INTEGER)
      -- Retrieve 'sum' from account.
    require
      sum_positive: sum > 0
      has_sufficient_funds: sum < balance
    do
      tot_withdrawals := tot_withdrawals + sum
    ensure
      balance_correct: balance = old balance - sum
    end

feature {NONE} -- Implementation

  tot_deposits: INTEGER
      -- Total amount deposited.

  tot_withdrawals: INTEGER
      -- Total amount withdrawn.

invariant
  valid_account: tot_deposits > tot_withdrawals
  info > 0
end
```

Listing 1.1: *BANK_ACCOUNT*, version 1.

The balance is computed on-demand from the total amount of deposits and withdrawals, and there is an explicit class invariant capturing the intended semantics of the bank account. This invariant is checked after the invocation of the constructor `make` of a bank account object, and also before and after the invocation of any other routine in the class.

Version 1 then evolves into version 2 (Listing 1.2), in which the query `balance` becomes an attribute updated every time a deposit or withdrawal takes place, and the attribute `info` becomes a string. While in languages with a C-derived syntax like Java we would need to modify client code to accommodate the fact that a method becomes an attribute, here clients are not affected, because in Eiffel both attributes and argumentless functions can be accessed in the same way from outside the class, providing uniform access to the class itself. Note also how the invariant is now expressed in terms of the new attribute `balance`.

```
class
  BANK_ACCOUNT
```

```
create make

feature  -- Initialization

  make
      -- Create a bank account with an initial deposit.
    do
      balance := 1
    end

feature  -- Status report

  balance: INTEGER
      -- Account balance.

  info: STRING
      -- Some numeric information, expressed as a string.

feature -- Basic operations

  deposit (sum: INTEGER)
      -- Add 'sum' to account.
    require
      sum_positive: sum > 0
    do
      balance := balance + sum
    ensure
      balance_correct: balance = old balance + sum
    end

  withdraw (sum: INTEGER)
      -- Retrieve 'sum' from account.
    require
      sum_positive: sum > 0
      has_sufficient_funds: sum < balance
    do
      balance := balance - sum
    ensure
      balance_correct: balance = old balance - sum
    end

invariant
  valid_account: balance > 0
```

```
end
```

Listing 1.2: *BANK_ACCOUNT*, version 2.

What happens when trying to retrieve an object stored with version 1 of class *BANK_ACCOUNT* into an object of version 2 depends on the retrieval system, which may trigger a failure or act leniently by silently accepting objects having the attributes *balance* and *info* initialized to the default. In case of the example this is certainly wrong. To understand why, imagine developers forgetting to code an appropriate transformation function assigning the difference between *tot_deposits* and *tot_withdrawals* stored in version 1 to the newly created *balance* attribute. Then *balance* will incorrectly be zero for all retrieved objects. The same happens if the integer value of attribute *info* is not appropriately converted into a string. Luckily, in the case of the *balance* attribute the new class invariant will be violated triggering an exception, because the value of *balance* is now zero and not positive as the invariant prescribes.

The framework presented in this thesis aims at supporting developers in evolving object-oriented applications. It integrates seamlessly into an IDE, keeping track of all class versions and system releases. When a coherent set of classes is released, developers are supported in writing transformation functions to migrate from one version to another. Depending on the specific evolutionary changes automatically detected, the framework support can vary from a function template containing some standard initializations to a full-blown function body generation, where no additional developer intervention is needed. Listing 1.3 shows the code produced by the framework for the transformation function in case of the previous example.

```
class
  BANK_ACCOUNT_SCHEMA_EVOLUTION_HANDLER

feature  -- Transformation functions

  v1_to_v2: HASH_TABLE [TUPLE [LIST [STRING], FUNCTION [ANY
      , TUPLE [LIST [ANY]], ANY]], STRING]
      -- Conversion function from version 1 to version 2.
    local
      tmp: SCHEMA_EVOLUTION_DEFAULT_CONVERSION_FUNCTIONS
    do
      -- auto-generated code and comments
      create temp
      create Result.make_default
      -- New attribute of type INTEGER
```

```
      -- What is the default value?
      -- Please also check the class invariant!
      Result.force (tmp.variable_constant (0), "balance")
      -- Convert from INTEGER to STRING
      Result.force (tmp.variable_change_type ("info", agent
         tmp.to_string (?)), "info")
      -- auto-generated code and comments
   end
end
```

Listing 1.3: Schema evolution handler for class `BANK_ACCOUNT`.

In case of the `balance` attribute, the tool recognizes that two attributes have disappeared and a new one has been added, but obviously needs the developer's help to figure out how to initialize it. On the bright side, a default value initialization will raise an exception because it violates the version 2 class invariant, so there is no risk of accepting inconsistent objects into the system. For the`info` attribute, the framework recognizes the type change and provides a full code generation of the transformation function body. Finally, the framework includes a retrieval algorithm that is aware of the presence (or absence) of transformation functions between any two versions of each class, triggering retrieval failures if the required transformation function is not present, or if the retrieved object does not satisfy the new version class invariant. This guarantees an invariant-safe class schema evolution.

1.3 Structure

Chapter 2 describes related approaches to class schema evolution. The rest of the related work is cited where appropriate.

Chapter 3 defines a conceptual framework for persistent evolving object-oriented applications by presenting a formal model representing attribute changes through a set of transformation functions, and suggesting the definition of a measure for the robustness of evolving object-oriented applications that persist their objects.

Empirical studies are essential to understand what kind of schema evolution actually happens in practice; without empirical evidence, proposed solutions to the schema evolution problem are bound to be off the mark. Chapter 4 presents the results of four empirical studies performed on existing software libraries and applications, written in both Java and Eiffel, and compute the proposed measure of persistent software evolution robustness for an existing and widely used Java library. One more study

analyzes the evolution of class invariants in eight software projects and investigates if and how their evolution influences the ability to retrieve previously stored objects.

Chapter 5 describes the framework implementation intended to support developers through the entire schema evolution process, from class release time to retrieval time. The framework implementation is described according to its two main components. The first component is a tool integrated into the EiffelStudio IDE and including version handling and code template generation for transformation functions. The second component is a persistence library, independent from the tool and supporting an object-oriented API to access multiple persistence stores and a retrieval algorithm preventing the acceptance of inconsistent objects into the system.

Chapter 6 presents the empirical evaluation of the framework front-end API, and finally Chapter 7 draws conclusions on the work done and suggests future work.

CHAPTER 2

RELATED WORK

The issues arising from schema evolution are widely acknowledged. They affect object-oriented databases, relational databases and all programming languages providing a serialization facility.

This chapter presents most of the related work. The rest is referred to where appropriate in the remainder of the thesis.

The related work is described using different sections for clarity. Section 2.1 describes the most common approach to object-oriented persistence, the one that focuses mainly on the current class schema and the retrieved one, therefore not explicitly supporting class versioning. Section 2.2 describes the persistence solutions offering explicit support for multiple versions, and contrasts them with the framework proposed by this thesis. Section 2.3 discusses the topic of schema evolution in relational databases. This is of interest because many object-oriented applications use relational databases, that also have evolving schemas and have been dealing with the related issues for quite some time. Section 2.4 describes other interesting approaches to class schema evolution that do not fall into the previous sections. Finally, Section 2.5 summarizes the contributions of this thesis with respect to the prior art.

2.1 The class schema approach

Given an object we need to retrieve, the most widespread approach compares the object's class schema available in the retrieving system with the class schema of the stored object. If a mismatch occurs, developers typically implement a transformation function to adapt the retrieved object to the newly created one. This approach is an attempt to provide a solution

that is practical and relatively easy to implement as opposed to implementing explicit support for multiple versions (see Section 2.2).

Serialization is a specific mechanism used to save relevant object information in a file or send it remotely over a network for later retrieval. As compactness might be of the essence, a binary encoding of the data is commonplace, but a textual format like XML or JSON may be used as well.

In Java a class can enable future serialization of its instances by implementing the *Serializable* interface, or its descendant *Externalizable* [3, 80, 68, 102]. The difference between *Serializable* and *Externalizable* is that while the former provides a default serialization mechanism with a predefined format, the latter gives to the class complete control over format and contents of the stream of objects and its supertypes. Developers can provide custom deserialization methods like *Serializable.readObject* and *Externalizable.readExternal* to help establishing the — typically implicit — class invariant. Providing custom deserialization methods becomes particularly important when objects have non-trivial class invariants that would not be satisfied by an automatic attribute initialization to the default values. An important constraint is that every serializable class has an automatically generated and unique identifier associated with it, stored in the *serialVersionUID* attribute and calculated using a complex algorithm mimicking closely the class schema (and including attributes information, method signatures and other class details). If developers accept the default generated value for the *serialVersionUID*, they implicitly allow the serialized form to become an encoding of the physical representation of the object graph rooted at the object itself. Therefore the deserialization process becomes very sensitive to most of the class and attribute modifications, and sometimes even to different compiler implementations. Therefore, accepting the default value for the *serialVersionUID* attribute can lead, in time, to unexpected difficulties in retrieving objects previously stored [14].

The JavaBeans framework offers an interesting solution for serialization. The main abstractions, *XMLEncoder* and *XMLDecoder*, suggest that the file format used is XML-based, making it possible to use XML transformations to handle class schema evolution. *XMLEncoder*, apart from cloning the object graph as expected, records the necessary steps to perform the clone, acting therefore more as a code generator than as a standard serializer. This means that the XML documents created are basically programs that can then be interpreted by the *XMLDecoder* against a fixed set of semantics, very much in an automatic fashion. The framework relies by default on the existence of accessors — setter and getter methods for attributes. The attributes that have accessors are called properties. When the properties are not sufficient to express a class's persistence needs, either

because there are some relevant attributes that do not have corresponding accessors, or because one does not want to store all the properties, it is possible to use persistence delegates (method objects passed as parameters to other methods) during the writing process. As all the object customization (including the creation information) is produced by the *XMLEncoder*, invoking a *readObject* method is not needed, and the *XMLDecoder* has a relatively simple structure: it is just an executor of the instructions coded by the *XMLEncoder* in reverse order [94].

The .NET framework offers three technologies to serialize objects, each suggested for specific scenarios. Data Contract Serialization (DCS) is used for general persistence, web services and JSON serialization; XML Serialization is used for a full control over the XML serialized format; Runtime Serialization (either in binary format or compliant to the Simple Object Access Protocol (SOAP)) is used for .NET remoting [42]. DCS requires developers that want persistent objects to apply the *DataContractAttribute* to the class and the *DataMemberAttribute* to the fields and properties. Various serialization callbacks can be invoked at serialization and deserialization time. The serialized stream is very sensitive to changes to the data members structure, so even changing their order may lead to a deserialization failure. The interface *IExtensibleDataObject* can be implemented to store any unknown data to the current version coming from a future version, which helps dealing with some of the possible deserialization issues. Runtime serialization requires the class to apply at the very least the *SerializableAttribute*, and offers more control by implementing the *ISerializable* interface to provide a special serialization constructor.

Python offers a serialization API similar to the ones seen above [64]. The action of serializing an object is called "pickling", while the action of deserializing is called "unpickling". Two formats are available: a default printable ASCII format, and a more efficient binary format. Callbacks are available to adapt the retrieved objects to the new format. Python's pickling mechanism constitutes the backbone of the ZOPE application server [40]. An interesting fact about Python's serialization is that the serialization protocol is made explicit via the *protocol* parameter. This helps detecting in advance possible deserialization failures due to serialization protocol evolution. To be more precise, while protocol backwards compatibility is guaranteed, it is not possible to read an object stored using a newer protocol when using an older protocol.

Eiffel's serialization mechanism offers a solution in which all conflicts are resolved in one class. Custom deserialization behavior is available by inheriting from a *MISMATCH_CORRECTOR* class and redefining the callback *correct_mismatch* to re-establish the class invariant. As a difference

with the solutions above, it is easier to detect class invariant violations because the native support for Design by Contract [90] will trigger a class invariant check when the *correct_mismatch* has terminated execution.

In our view, the serialization mechanisms of object-oriented languages are converging towards a full-fledged solution such as an object oriented database management system (OODBMS), because they have increasingly smaller memory footprints and the ability to selectively read data, while in the case of serialization mechanisms the data have to be read as a whole. To assess such a solution, we examine now some object-oriented databases.

The db4o object oriented database [104, 38] offers an advantage under the point of view of API usability, because it allows objects to be stored as they are, without polluting them with persistence code. This means that it is not necessary to mark the objects in any special way to allow them to be persisted. It is sufficient to pass them to the method *set* in class *ObjectContainer*. With respect to schema evolution handling, if developers need a custom behavior to establish the retrieving object class invariant, they have two possibilities: choose to use reflectively invoked methods in the object class or register listeners to specific *ObjectContainer* events outside the object class. As seen for Java and .NET, there is no explicit support for class invariant violation.

The Orion object-oriented database [9] introduces a model recognizing the importance of invariants to validate conversions, and deals with a broad set of code changes, mostly involving both single and multiple inheritance. The model proposed in Chapter 3 is simpler with respect to the set of code changes, because it considers the object's flattened form. It is also in one case more general, because it allows switching to any type when detecting an attribute type change, while Orion only allows switching to an ancestor type. The proposed framework also offers a limited support for "attribute renamed", while Orion does not take any action in this case. Finally, in Orion invariants are mostly related to keeping the system consistent with respect to inheritance, while the proposed framework uses class invariants to validate the conversions semantically.

2.2 The multi-version approach

Storing all versions of a class makes it easier to provide more specific schema evolution support. ENCORE [112] is an early example enforcing a serialized form limited to the attributes in the class interface, and proposing an ad hoc constraint language for inter-property constraints.

A general framework for class schema evolution that has been applied

to an OODB was proposed by Lautemann [84]. Two important differences with the framework described in this thesis are that it required a schema designer role conceptually distinct from the application developer role, and envisioned a completely transparent schema evolution that it has proven to be impossible to achieve in all scenarios.

A more recent example of a multi-version OODB is Versant Object Database (VOD) [39]. It consists of two main storage areas: a dictionary, which carries the class definitions for objects in the database, and the actual database, containing the serialized objects forms. The dictionary is necessary to understand the structure of the objects stored in the database. The delicate issue consists in modifying the dictionary after having stored some objects. The dictionary keeps versions for every class, so when a class schema modification is detected, old objects are lazily converted if possible. In the case of an attribute added to the latest version of a class, the automatic, lazy object conversion initializes it to the default at retrieval, a procedure that may be risky as seen in Section 1.2. When automatic conversions are not possible, VOD provides a tool to perform server-side manual class schema evolution.

In Objective C, to support encoding and decoding of instances a class must adopt the NSCoding protocol and implement its methods. The protocol declares two methods that are invoked on the objects being encoded or decoded. During encoding or decoding, a coder object invokes methods that allow the object being encoded to substitute a replacement class or instance for itself. Objective C also uses "keyed archiving", in which every value encoded is given an arbitrary string key and there is no automatic versioning. This allows more flexibility when fixing a possible mismatch, and also allows for inserting custom version information. An interesting technique that tries to cope with forward compatibility is "fallback handling", useful when one of a set of possible keys for a value is encoded. The set of supported keys may evolve over time, with newer keys being preferred in future versions of your class. Fallback handling defines a fundamental key that must be readable forever, but is used only when no other recognized keys are present. Future versions can then write a value using both a new key and the fallback key. Older versions of the class will not see the new key, but can still read the value with the fallback key [76].

To convert a stored object of a certain version into an object of the current version, CLOSQL uses update or backdate routines [96]. An unfortunate requirement is that a database administrator is needed every time a class is created, to specify which update or backdate routines have to be executed. As in our approach, the update and backdate routines for a certain version are kept all together, in this case in an "update method". CLOSQL

supports "linear versioning": a new version can only be generated from the latest version. In contrast to CLOSQL, the framework discussed in this thesis handles transformation functions also from older versions with respect to the last one.

GemStone is a computationally complete Smalltalk extension providing data definition, data manipulation and query facilities to persist objects across executions [67]. Each class in GemStone has an associated class history containing all the previous versions. Instead of providing automatic support for class versioning like our framework does, GemStone allows developers to decide when to define a new version and when to override ad hoc migration methods in the destination class.

The Oracle relational database, starting from version 11g release 2, has introduced "editions" to offer better support for online application upgrade [103]. The concept is more restrictive with respect to what the authors call releases, because it is limited to "editionable" object types, which do not include, for example, tables and java classes.

When a certain class evolves over time, it may be considered a different type, and named differently, or it may be considered the same type, keeping the same name but providing other means of taking into account the different inner structure and semantics. While the second option is commonplace, the first one has been previously explored [106]. Every new version of a class is given a new name including version information. Versions can therefore be considered full citizens of the type system. To cope with this scenario, Eiffel provides `converters` [92]. Converters are a language mechanism intended to cover those cases in which inheritance is not appropriate, for example to convert string or numeric type implementations across different systems. The main idea is that two types can either conform, via inheritance, or convert to each other, but not conform and convert at the same time. The mechanism (integrated into the runtime) takes care of automatically invoking appropriate transformation functions placed into the class itself when operations like assignments or argument passing are performed. A downside of this approach is the lack of scalability, because every type should provide converters for each previous version, therefore clogging the corresponding class code with a potentially high number of transformation functions.

Another example of making different, parallel versions explicit can be found in UpgradeJ [12], though focused neither on object persistence nor on enforcing semantical consistency.

The automated detection of code changes has been extensively explored in literature, both with respect to existing libraries and software configuration systems, for example by Dig et al. [49]. D'Ambros et. al [45]

propose general techniques for analyzing software repositories for code changes: one in particular, hotspot analysis, is interesting because it focuses on entities that change frequently and therefore can be critical for the evolution of a software system. The proposed idea is to devise heuristics and warning mechanisms similarly to the work presented in this thesis for class attributes changes. A specific algorithm for extracting fine-grained source code changes in Java code called *change distiller* was proposed by Fluri et al. [62]. The algorithm operates on abstract syntax trees in different revisions, detects changes using similarity measures and produces an edit script to transform one tree into the other. The changes are classified according to a taxonomy of 35 change types, of which two refer to attribute declaration (renaming and type change) and two to class-body changes (attribute addition and removal) [60]. In comparison to the sophisticated and broader in scope change distilling algorithm, the algorithm used in this thesis provides one new attribute change type ("Attribute made attached") and is focused on detecting only of the changes that matter to persistence with certainty, providing warnings when a perfect match is not possible.

One interesting approach proposes that schema evolution can be solved by devising bi-directional transformations [43, 73]. Though acknowledging the advantages of generating the inverse transformations (mostly) for free from a bi-directional domain specific language, it is the author's belief that using it would be an excessive burden on the developer. The idea is that it is easier to work with just one programming language, and that the class invariant checks and the transformation functions created early can help to improve the software system robustness with respect to its capacity to evolve and retrieve previously stored objects.

2.3 Schema evolution in relational databases

Schema evolution issues are also relevant for relational databases [46]. The inference rules derived in Section 3 are related to those implemented in many schema matching tools available [108]. In particular, TransScm and Tess [95, 86] view a schema as a set of types and provide two-version transformations following specific rules. Human intervention is required to provide new rules or to select match candidates. Apart from the different transformation functions semantics and implementation, the framework proposed in this thesis keeps track of all versions of a class schema, and limits the modification operations to those for which the data surveys

in Chapter 4 show potential application.

The PRISM workbench [41, 7] presents some analogies with what is presented in the following chapters, namely the use of a GUI-based tool to facilitate schema evolution, and a certain degree of automation in the process. The main difference to be noticed is that in PRISM the database administrator has to input manually the Schema Modification Operators (SMO), while in our case they are detected automatically by the system at release time. An additional difference is that PRISM's SMOs, though more numerous than the ones selected for our framework, do not include "column type change", that would be an analog for our "attribute type changed".

If we consider a class under a persistence point of view, what we are left with is a set of attributes with their respective name and types (primitive types or other class types), and a class invariant, establishing a semantic relation among the attributes.

In a relational database table there is not a direct counterpart for a class invariant, but there are the following kinds of state constraints instead:

- Domain constraints, specifying a certain range of values for attributes.
- Tuple constraints, specifying attribute comparison.
- Relation constraints: uniqueness with a primary key (PK), functional dependency and aggregate.
- Database constraints, involving more relations, for example referential integrity constraints.

Grefen and Apers analyze the issues caused by semantic errors in relational databases [69] and underline that referential integrity constraints are particularly important when the a relational database is used to contain data coming from an object domain. Example of relational database engines including support for state constraints include IBM DB2, Oracle, Ingres, and Postgres.

2.4 Other approaches to schema evolution

Orthogonal Persistence Java (OPJ) is an extension of the Java Language Specification adding "orthogonal persistence" capabilities to the Java platform [4, 6]. The three principles defining orthogonal persistence are:

- Type Orthogonality: persistence is available for all data, irrespectively of type.
- Persistence by Reachability: the lifetime of all objects is determined by reachability from a designated set of root objects, the so-called persistent roots.
- Persistence Independence: it is indistinguishable whether code is operating on short-lived or long-lived data.

The well-known prototype implementation for OPJ, PJama [5, 8, 50] is an extension of the Java Virtual machine together with a persistent store, in which the state of an executing application is kept. The system state is check-pointed atomically and periodically to be able to recover from exceptions and crashes. PJama provides an approach to schema evolution that involves persisting both objects and classes. To perform conversions between classes, PJama developers use a small API and a standalone, command-line utility, so the whole process is not automated and requires human intervention. In case the changes are validated, objects are typically converted eagerly.

Several authors suggest to assist metamodel evolution by stepwise adaptation in a similar fashion with respect to what is presented in this thesis, using a transformational approach and a classification of modifications [119, 70, 72, 20]. Analyzing the AST in metamodels and applying similarity metrics to detect changes has been done by Falleri et al. [59]. All the cited works about metamodels are of interest because class schemas are a particular kind of metamodels.

The automatic generation of transformation functions through type transformers in C programs has been described by Neamtiu et al. [97]. It focuses on updating structs whose layout might evolve. However, it is not per se linked to object-orientation and it does not benefit from having a model or an integration into an IDE.

The formal object model proposed by the ODMG standard [18] encompasses more definitions than the framework described in this thesis, but it is missing some modifications on attributes that have seen to be relevant in practice, like “attribute renamed” and “attribute type changed [48].

BKNR is the Common Lisp Web Application environment based on transaction logging and supporting immutable binary large objects and Common Lisp Object System (CLOS) persistence [47]. All operations which change the persistent image of a system are explicitly written to a transaction log file. The persistence mechanism supports a snapshot API

which allows the persistent object system to write all currently active objects to a sequential file, after having tagged them with IDs. Persistent objects IDs are written to the transaction log. Schema evolution happens while BKNR is reading a snapshot in the sequential file. For each class in the snapshot, there is a layout record containing the class name and the slot (attribute) names. When reading a snapshot file, the object system compares the class layout in the snapshot against the class definitions of the running code. Inconsistencies detected are signaled, and restarts are used to select schema evolution actions [75]. While this works in the simplest cases, specific conversion methods can be defined for more complex cases. The strength of this approach is that one can easily load snapshots from a production system using new code, without any additional required conversion step. This means that changes are detected early, before they become too complex, and there can be an incremental refinement of the schema evolution process conceptually similar to the one promoted by the present thesis.

A more generalized approach tackles the issues connected to software evolution by using the semantic web. Würsch et al. propose SEON, a pyramid of ontologies for software evolution devising a unifying taxonomy for software evolution analysis and services [120]. One of the SEON's ontologies refers to fine-grained source changes and it is based on the change distiller meta-model mentioned in Section 2.2. It seems natural to contribute to this effort by extending the ontology mentioned above with the concept of attribute attachment discussed in Section 3.2.6.

2.5 Summary of contributions with respect to previous work

This thesis presents a framework that contrasts in different ways with the approaches presented in the previous sections. Section 2.1 describes the most common approach to object-oriented persistence, the one that focuses mainly on the current class schema and the retrieved one, therefore not explicitly supporting class versioning. This pragmatic approach is acceptable for short-lived applications, or for applications that have to maintain small amounts of relatively stable data. It becomes problematic when applied to long-lived applications that undergo significant changes in their lifetime and need to persist their objects. This happens because the code to handle all the possible schemas has to live inside the only code-bloated transformation function available, because the retrieval issues are

only acknowledged when they happen, and because there is no particular support—during the retrieval operations—to ensure that the retrieving class' invariant holds. The approaches described in Section 2.2 provide support for multiple versions, and therefore provide a solution to the code bloating issue mentioned above, but do not solve neither the issues due to late acknowledgement of the retrieval conflicts, nor the issues due to disregard for class invariants. Relational databases are of interest because they also have evolving schemas and have been dealing with the related issues for quite some time. Section 2.3 shows that they suffer from similar issues with respect to the ones mentioned about the systems supporting multiple versions, in particular with respect to providing timely support when conflicts arise and with respect to enforcing state constraints. The various approaches described in Section 2.4 provide some interesting ideas, but don't seem to offer a full-fledged solution immediately applicable to nowadays software systems and including IDE support and invariant-safety.

That said, a list of the contributions of this thesis with respect to the prior art is summarized below:

- A framework supporting multiple versions and multiple transformation functions (one for each pair of versions) which are kept separate from the domain class they are associated with.

- An IDE-integrated tool to support developers with the schema evolution-related issues at system release time.

- A retrieval algorithm ensuring an invariant-safe object retrieval.

CHAPTER 3

EVOLVING PERSISTENT APPLICATIONS: A MODEL

Object-oriented applications evolve over time. This means that classes may change their structure, in terms of routines and attributes. A persistence mechanism needs to be aware of changes in the class structure, and in particular of those concerning the attributes, because their names and types are typically stored together with their values, and so it may happen that the stored class structure does not match the current one during retrieval. This thesis presents a model of updates that particularly emphasizes the generation of transformation functions — unlike previous models [17, 65]. This chapter presents a conceptual framework modeling the changes that may occur in class attributes. The purpose of the model is to capture the most relevant and common attribute changes and the possible transitions between them in a way that it is possible to implement a framework using this information.

Section 3.1 introduces a simplified syntax for Eiffel classes, which is then used in Section 3.2 to define a set of *schema modification operators* (SMOs) modeling atomic attribute changes. Section 3.3 shows that the SMOs can be composed to produce class transformations, and that such transformations are complete with respect to attribute modifications. Section 3.4 illustrates that the generation of object transformers from a class transformation can be expressed as transformation functions. Section 3.5 illustrates how the SMOs are detected, and describes the heuristics used to generate object transformers. This is important to make the model implementable.

Finally, Section 3.6 defines a measure for the robustness of object-oriented applications that persist their objects. The measure will be com-

puted in Chapter 4.1.2 for a widely used Java library to know more about its expected behavior with respect to its class schema evolution.

3.1 Syntax for classes

Figure 3.1 presents a simplified syntax of class definitions in Eiffel programs. The notation represents a class as set of attributes. This makes sense in terms of persistence because when storing an object of a class we typically store its attributes' names and values. The definition includes a notation for void safety. A program is void safe if it has no void dereferencing errors. A void (or null) dereferencing error happens when the operation `x.f()` fails because the target object `x` denotes a void reference at execution time [93]. Since release 6.1 (2007) the Eiffel language has incorporated the *void safety* mechanism into the compiler, requiring developers to indicate, for every attribute, whether it can accept void values (keyword `detachable`) or not (keyword `attached`); see Appendix A. The definition omits the declaration both of routines and constraints on generic parameters, as they are typically not included in the persisted form. It also does not explicitly consider inheritance, either single or multiple, because it assumes access to the flattened class schema, which includes all the attributes from ancestor classes. This is a valid assumption, because in practice the persistence mechanism flattens each object as it stores it.

$C, D \in Names$			*class names*
$N, M \in GNames$			*names for generic parameters*
$name \in names$			*names for attributes*
att	$::=$	$name : type$	*attribute definition*
	$\mid$	$name : N$	*attribute of generic type*
$type$	$::=$	C	*class type*
	$\mid$	$type[type]$	*generic derivation*
	$\mid$	**attached** C	*void-safe type*
	$\mid$	**detachable** C	*non void-safe type*
$class$	$::=$	**Class** C **feature** $att_1, \ldots, att_n$ **end**	*class*
	$\mid$	**Class** $C[N_1, \ldots, N_i]$ **feature** $att_1, \ldots, att_n$ **end**	*generic class*

Figure 3.1: Syntax for class definitions of Eiffel programs.

3.2 Schema modification operators

Adapting the definition from Curino et al. [41, 7] to an object-oriented context, the model is based on a set of *schema modification operators* (SMO). Each SMO is a change to a class schema, along with the semantic changes possibly occurring to all the instances conforming to that schema. More formally, an SMO is a function

$$A : class \mapsto class$$

modifying at most one attribute. The proposed model defines six SMOs:

- Attribute not changed.
- Attribute added.
- Attribute renamed.
- Attribute type changed.
- Attribute removed.
- Attribute made attached.

The choice is motivated by two reasons: firstly, when an application stores objects it mainly stores class attributes and their values, so the SMOs are all attribute modifications. Secondly, in our experience, supported by the empirical data presented in Chapter 4, the first five SMOs are the changes most frequently performed. We included the "Attribute made attached" SMO because we believe that it will become a relevant attribute modification in the near future. This intuition is based on the fact that in the statically typed programming languages community there seem to be an increasing interest and effort into providing more static guarantees that certain categories of errors will not happen at runtime. With respect to this, the elimination of the possibility that reference types are used at runtime without having an object attached (therefore raising null pointer exceptions) is certainly an important topic. The set of chosen SMOs implies the following changes found In Chapter 4: "Attribute becoming a constant" (relevant because constants are not serialized), is considered as "Attribute removed"; "Constant becoming an attribute" is similarly considered as "Attribute added". In addition, class (static) attributes are not considered because they are typically not serialized.

In the following sections the six SMOs are defined.

3.2.1 The attribute not changed schema modification operator

Figure 3.2 defines the *attribute not changed* SMO. In this case what matters is not what is shown in the figure, but what is not shown instead. The reason why we introduced this SMO is to take into account that an attribute not changing its name and type across versions can still change its semantics. An example could be a numeric attribute expressing a measure in different units, or a string attribute expressing a date in different formats.

$$\frac{att_i \in class_0 \qquad class_0 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_n\ \textbf{end}}{A_{noChange(att_i)}(class_0) = class_0}$$

Figure 3.2: Attribute not changed schema modification operator.

3.2.2 The attribute added schema modification operator

Figure 3.3 defines the *attribute added* SMO. The rule clarifies that we consider an attribute as added if its name is different from any other attribute name in the existing class version, where none of the previous attributes has been removed. This is the most common attribute modification, both according to our experience and the empirical data presented in Chapter 4. The most likely reason for its popularity is that adding attributes and methods is the preferred evolutionary activity, because it does not break backward compatibility. In this context the issues arising from persistence are an unwelcome side effect. If an attribute added in a new class version needs a specific initialization, it is suddenly not sufficiently to initialize it to its default, because the class invariant can be put at risk as seen in Section 1.2.

$$\frac{att \notin \{att_1, \ldots, att_n\}}{A_{new(att)}(\textbf{Class}\ C \ldots \textbf{feature}\ att_1, \ldots, att_n\ \textbf{end}\ ) = \textbf{Class}\ C \ldots \textbf{feature}\ att_1, \ldots, att_n, att\ \textbf{end}}$$

Figure 3.3: Attribute added schema modification operator.

3.2.3 The attribute renamed schema modification operator

Figure 3.4 defines the *attribute renamed* SMO. In particular, the first line of the rule specifies that the old and the new version of the attribute have different names but same (possibly generic) type, and the third line of the rule specifies that the new version of the attribute has a different name with respect to any of the other attributes of the new class.

$$\frac{\begin{array}{c}(att_i = name : N \wedge att'_i = name' : N) \vee (att_i = name : type \wedge att'_i = name' : type)\\ att_i \in class_0\\ att'_i \notin \{att_0, \ldots, att_{i-1}, att_{i+1}, \ldots, att_n\}\\ class_0 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att_i, att_{i+1}, \ldots, att_n\ \textbf{end}\\ class_1 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att'_i, att_{i+1}, \ldots, att_n\ \textbf{end}\end{array}}{A_{nameChange(name,name')}(class_0) = class_1}$$

Figure 3.4: Attribute renamed schema modification operator.

3.2.4 The attribute type changed schema modification operator

Figure 3.5 defines the *attribute type changed* SMO. In particular, the first line of the rule specifies that the old and the new version of the attribute have different types but the same name.

$$\frac{\begin{array}{c}type \neq type' \wedge att_i = name : type \wedge att'_i = name : type'\\ att_i \in class_0\\ class_0 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att_i, att_{i+1}, \ldots, att_n\ \textbf{end}\\ class_1 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att'_i, att_{i+1}, \ldots, att_n\ \textbf{end}\end{array}}{A_{typeChange(name:type,name:type')}(class_0) = class_1}$$

Figure 3.5: Attribute type changed schema modification operator.

3.2.5 The attribute removed schema modification operator

Figure 3.6 defines the *attribute removed* SMO. Together with the attribute added SMO, the attribute removed SMO plays an important role in proving that the SMOs can be composed to produce class transformations, and that such transformations are complete with respect to attribute modifications (see Section 3.3).

$$\frac{\begin{array}{c} att_i \in class_0 \\ class_0 = \textbf{Class } C \ldots \textbf{feature } att_0, \ldots, att_{i-1}, att_i, att_{i+1}, \ldots, att_n \textbf{ end} \\ class_1 = \textbf{Class } C \ldots \textbf{feature } att_0, \ldots, att_{i-1}, att_{i+1}, \ldots, att_n \textbf{ end} \end{array}}{A_{removeAttribute(name)}(class_0) = class_1}$$

Figure 3.6: Attribute removed schema modification operator.

3.2.6 *The Attribute made attached schema modification operator*

Attribute made attached, defined in Figure 3.7, covers the case of an attribute that was allowed to be void (null) in a certain version and becoming *attached* (guaranteed non-void) in a subsequent version. This scenario implies that the attribute needs to be explicitly instantiated at retrieval time.

$$\frac{\begin{array}{c} att_i = name : C \wedge att_i' = name : \textbf{attached } C \\ att_i \in class_0 \\ class_0 = \textbf{Class } C \ldots \textbf{feature } att_0, \ldots, att_{i-1}, att_i, att_{i+1}, \ldots, att_n \textbf{ end} \\ class_1 = \textbf{Class } C \ldots \textbf{feature } att_0, \ldots, att_{i-1}, att_i', att_{i+1}, \ldots, att_n \textbf{ end} \end{array}}{A_{addAttach(att_i)}(class_0) = class_1}$$

Figure 3.7: Attribute made attached schema modification operator.

As described in Appendix A, Eiffel provides a mechanism that can statically detect such erroneous use of references and relies on the two keywords `attached` and `detachable`. Other programming languages provide support for non-null types.

C# provides *nullable types*; though useful when working with null values in databases, they are restricted to value types (like numeric types and *structs*).

Scala tackles the issue by providing the *Option* class, that must deal with the *None* value, and does not solve the issue at compile time [98].

Java offers the *Option* generic type as well, useful to avoid a *NullPointerException* at runtime but at the cost of a conditional instruction. Again the compiler does not check for null references statically. Java's arguably best IDE, IntelliJ IDEA, provides the *@Nullable* and *@NotNull* annotations for statically detecting *NullPointerExceptions* [79]. These have been proposed for inclusion in the standard JVM, but the issue is still pending at the time of writing. There are also static analysis tools solving the problem of null pointer references, like the FindBugs tool developed by University of Maryland, providing compile-time checks for Java [74, 99, 100].

The type system of Spec# distinguishes between *non-null* types and *possibly-null* types, and its checker does not allow possibly-null types to be dereferenced [10].

Finally, the NICE programming language provides language annotations equivalent to those for Eiffel [15], and guarantees no references pointing to null at runtime.

3.3 Completeness

The given set of SMOs is *complete* in the following sense: any modification to the attributes of a class can be described by some *class transformation*, where a class transformation $T_{A_1,\ldots,A_N}$ going from one version of a class to another is described as a composition of SMOs $A_1, \ldots, A_N$ ($N \geq 1$):

$$T_{A_1,\ldots,A_N} : class \mapsto class$$

such that:

$$T_{A_1,\ldots,A_N}(class_0) = (A_n \circ \ldots \circ A_1)(class_0)$$

The proof of this property is as follows: suppose we have two classes C1 and C2 with n attributes and m attributes respectively, and assume the generic parameter list is preserved between C1 and C2. Hence there always exists a class transformation from C1 to C2 that first deletes all the n attributes from C1 and then adds all the m attributes to C2. Thus, class transformations are complete with respect to attribute modifications.

While there is always a decomposition, using a straightforward algorithm might not produce the best results. Devising a heuristic is therefore essential to use this model as the basis of an implementation. Section 3.5 describes the heuristics in detail.

3.4 Concrete transformation syntax and object transformers

Figure 3.8 presents a possible concrete syntax for object transformers. An object transformer is a function accepting as an argument an object of a previous version of a class, and returning an object of a new version of the same class. Its body may contain attribute assignments, invocations of other object transformers or occasionally no instructions (no-op). The new definitions introduced are `oldc` referring to the value of an expression `c`

before it is evaluated, and **Result** referring to the return value of the object transformer.

$otname \in OTNames$			*object transformers names*
e	$::=$	$e.e \mid oldc \mid name$	*expressions*
$instruction$	$::=$	$Result.name := e \mid$ *no-op* $\mid otname(e, e)$	*instructions*
$instructions$	$::=$	$instruction \mid instruction; instructions$	*list of instructions*
ot	$::=$	$otname(oldc : oldC) : C\ instructions$	*object transformer*

Figure 3.8: A concrete syntax for transformers.

The generation of object transformers from a class transformation can be expressed as transformation functions as shown in Figure 3.9. The function $\mathcal{G}[\![\cdot]\!]$ takes a class transformation and generates the associated object transformer `ot`. Generating an object transformer that creates an instance of the new class from a stored instance of the old class means generating its name and the actual instructions in its body. Function $\mathcal{S}[\![\cdot]\!]$ takes a class transformation as an argument and generates the actual instructions forming the object transformer code. After the definition of $\mathcal{S}[\![\cdot]\!]$ we illustrate in Figure 3.9 all the possible transformation functions that can be used as arguments, that is, the SMOs. While the SMOs allow a straightforward representation of the static transformations of a class, this is not sufficient to generate the object transformers. The main reason is that there is a need for an explicit default initialization of new attributes. To define such initializations, programmers must provide default values for every affected type. This is expressed by the $\mathcal{I}[\![\cdot]\!]$ input token, evaluating to the next value in a list of inputs. It is worth noticing that the “Attribute type changed” SMO will be processed completely only if there is a specific object transformer that knows how to convert the two specific types. This is the case, for example, of strings versus integers. In the case in which no such specific object transformer is available, we use the output generator $\mathcal{W}[\![\cdot]\!]$ (for warning) to signal the inability to generate a translation or a removal operation and evaluating to `no-op`. FInally, the last line in Figure 3.9 states that the transformations are processed one after the other.

$$
\begin{array}{l}
\mathcal{G}[\![\cdot]\!] : (class \mapsto class) \mapsto ot \\
\mathcal{G}[\![T_{A_1,\ldots,A_n}]\!] = otname(oldc : oldC) : C\ \mathcal{S}[\![T_{A_1,\ldots,A_N}]\!]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S}[\![\cdot]\!] : (class \mapsto class) \mapsto instructions & & \\
\mathcal{S}[\![A_{noChange(att_i)}]\!] & = & Result.name_i\textbf{:=}oldc.name_i \\
\mathcal{S}[\![A_{new(att)}]\!] & = & Result.name\textbf{:=}\mathcal{I}[\![\cdot]\!] \\
 & & \textit{where att=name:...} \\
\mathcal{S}[\![A_{nameChange(name:type,name':type)}]\!] & = & Result.name'\textbf{:=}oldc.name \\
\mathcal{S}[\![A_{typeChange(name:type,name:type')}]\!] & = & Result.name\textbf{:=}oldc.name \\
 & & if\ \textit{type' is assignable to type} \\
 & = & Result.name\textbf{:=}otname(oldc.name) \\
 & & \textit{if there is an object transformer otname} \\
 & & \textit{transforming an object of type type} \\
 & & \textit{into an object of type type'} \\
 & = & \mathcal{W}[\![name_0]\!]\ otherwise \\
\mathcal{S}[\![A_{removeAttribute(name_0)}]\!] & = & \mathcal{W}[\![name_0]\!] \\
\mathcal{S}[\![A_{addAttach(att_i)}]\!] & = & Result.name_i\textbf{:=}oldc.name_i \\
\mathcal{S}[\![T_{A_1,\ldots,A_n}]\!] & = & \mathcal{S}[\![A_1]\!];\mathcal{S}[\![T_{A_2,\ldots,A_N}]\!]
\end{array}
$$

Figure 3.9: Object transformers.

3.5 Heuristics for class schema evolution

While we have considered the extraction of transformations based on actual changes made by programmers using the IDE, these do not always guarantee an improvement in handling the transformations with respect to what is suggested in Section 3.4. This is because developers do not have to use the IDE's refactoring facilities, or they can simply write their program using a different IDE, or even no IDE at all. This is why the current solution relies on statically comparing the abstract syntax trees (ASTs) to detect SMOs. While this guarantees that a comparison between existing versions can always be performed, static comparisons imply that in certain situations, like the "Attribute renamed" SMO described in item 5 below, the outcome of the tool can only suggest, rather than guarantee, the possibility that the specific SMO has been applied.

To SMO detection algorithm (see also Section 5.2) first iterates through the new class attributes, searching for a match with attributes in the old class, and then iterates through the old class attributes, searching for a match with attributes in the new class. The second iteration will make it

possible to find all the attributes that were removed. After detecting the SMOs, the subsequent step is to create appropriate heuristics. Among the following techniques, 1, 2, 3, and 4 can be inferred with certainty; the others are only heuristics.

1. An attribute that does not change name and declared type generates an "Attribute not changed" SMO. A possible change in the attribute semantics will be taken care by the retrieving object's class invariant. The framework is always able to detect this SMO correctly, and provides a template for a transformation function and a warning about this possibility.

2. An attribute that does not change name but changes type between two versions generates an "Attribute type changed" SMO. The framework is always able to detect this SMO correctly, and depending on the type involved, to provide a complete code generation (as in the example in Section 1). A change in the attribute semantics is taken into account by the new type and by the retrieving object class invariant.

3. An attribute of the new version that does not appear in the old version generates an "Attribute added" SMO. The framework always detects this SMO correctly, initializes the attribute to its default and suggests checking the new class invariant with respect to the old one to avoid retrieval failures.

4. An attribute *att* removed from the old version generates an "Attribute removed" SMO if in the new version there is no new attribute *att′* having the same type as *att*. The framework always detects this SMO correctly. Once again, the class invariant of the retrieving object will take care of validating the new version semantics.

5. An attribute *att* removed from the old version is a candidate for generating an "Attribute renamed" SMO if in the new version there is a new attribute *att′* having the same type as *att*. By only comparing the two classes' ASTs it is not possible to determine if a rename actually occurred, because the new attribute could be semantically unrelated to the old one. In this case we would have a class transformation composed by an "Attribute removed" followed by an "Attribute added". The framework detects that this SMO might have occurred and generates a transformation function template warning about the possibility of a rename. In this case the retrieving object class invariant is essential in establishing the correct object semantics. It would

be possible to obtain a result worthy of higher confidence by checking that the variables are used by the same clients in the same context. As this approach requires a global analysis of the code, we considered it was too time-consuming and limited ourselves to analyzing the ASTs statically. An intermediate approach would be to limit the code analysis to the class routines using the attribute candidate to a rename. This would certainly help to better detect attribute renames with respect to the AST only solution and it will therefore be considered for future work on the tool (see Section 7.3).

6. An attribute of the old version that has been augmented with the keyword *attached* in the new version generates an "Attribute made attached" SMO. The idea is to make sure the attribute is attached by explicitly creating an object in the transformation function body, attaching it to the attribute and coding a postcondition stating that the attribute should be non-void. Here both the transformation function postcondition and the retrieving object class invariant are crucial to validate possible semantics changes.

3.6 A measure of robustness to evolution for persistent object-oriented applications

It can be helpful to measure a persistent object-oriented software statically with respect to its robustness to software evolution.

Consider a class C with *m* versions, and a binary relation T over it defined as "there exists a transformation function between versions v_i and v_j" ($i, j = 1, \ldots, m, i \neq j$). We define a measure of class persistence evolution robustness (PER) for C as the ratio of the modulus of the transitive closure τ^+ of the binary relation T to the total number of transformation functions. As there are `m(m-1)` possible pairs of versions, each pair representing a transformation function from a version v_i to a version v_j, we can then write:

$$PER(C) = \frac{|\tau^+(T)|}{m(m-1)} \tag{3.1}$$

The transitive closure τ^+ is needed to reflect the fact that the existence of transformation functions between v_i and v_{i+1} and between v_{i+1} and v_{i+2} implies the one between v_i and v_{i+2} ($i = 1, \ldots, m-2$).

The PER measure for a class C ranges therefore between 0 and 1, where 1 means that all the transformation functions between any two of the class'

versions exist. In this sense the class is considered "robust" with respect to schema evolution.

As an example of an application of definition 3.1, consider the two versions of class *BANK_ACCOUNT* introduced in Section 1.2, and assume these are the only two existing versions of the class. If someone writes only the forward transformation from version 1 to version 2, the PER of class *BANK_ACCOUNT* evaluates to 0.5: one existing transformation out of a possible two (one in each direction).

As another example, consider a class with 5 versions and 4 existing transformation functions, such as those from version v_i to version v_{i+1} ($i = 1, \ldots, 4$). Because of the transitive closure, there will be 6 more transformation functions (v_1 to v_3, v_1 to v_4, v_1 to v_5, v_2 to v_4, v_2 to v_5, v_3 to v_5) that can be implied by the first 4. For example v_1 to v_2 and v_2 to v_3 imply v_1 to v_3, and so on. In total there are 10 transformation functions out of 20, giving a PER of 0.5 again.

The PER measure of a class with `m` versions can be easily restricted to a single version v_i by computing the ratio of the existing transformation functions between v_i and any other version to the total number of transformation functions that in this case is `2(m-1)`. In the case of the second example above, the class PER for version v_i ($i = 1, \ldots, m$) would be `(m-1)/(2(m-1))`, so 0.5 once more.

The PER measure can also be extended to a system release, by computing the average of the PERs of the classes in the release, though in this case we get a measure that provides only a general indication.

To be able to compute the PER measure it is not necessary to adopt the framework proposed in this thesis. Section 4.1.2 shows how we compute PER for a software library that does not even implement the multi-version approach: the `java.util` package and its classes. This can be useful to evaluate the risk of experiencing retrieval failures before deciding to radically change the approach to class schema evolution in an existing software system.

CHAPTER 4

HOW SOFTWARE EVOLUTION AFFECTS PERSISTENCE: EMPIRICAL EVIDENCE

Empirical studies are essential to understand what kind of schema evolution actually happens in practice; without empirical evidence, proposed solutions to the schema evolution problem are bound to be off the mark. To assess the conceptual framework presented in Chapter 3 we therefore need to see if the evolution of classes whose objects are likely to be persisted actually happens. We formulate the following research questions:

1. Are the schema modification operators suggested in Chapter 3 actually found in realistic code bases (both software libraries and applications)?

2. Besides Eiffel—the language used for the framework implementation—do the results apply also to a more widespread language like Java?

3. Can the measure of robustness for the evolution of persistent applications defined in Section 3.6 be used to understand to what extent we are able to successfully retrieve previously stored objects in the package *java.util*?

This chapter analyzes the code of four software projects in two programming languages, Java and Eiffel, focusing on persistence-affecting changes. The projects are the following:

- The *java.util* data structure and utilities library (Section 4.1.1).

- The Apache Tomcat web server (Section 4.1.3).
- The EiffelBase data structure and utilities library (Section 4.2.1).
- The EiffelStudio IDE (Section 4.2.2).

For each language, the choice made is to analyze one set of classes from a widely used data structure and utility library, and one other from an established software project. The complete set of data is available online [87].

Section 4.1.2 describes the computation of the measure of robustness for the evolution of persistent applications defined in Section 3.6 on the *java.util* package, with the purpose of understanding to what extent we are able in practice to successfully retrieve previously stored objects.

Section 4.3 discusses the evaluation of the persistence framework described in chapter 3.

Finally, Section 4.4 describes two empirical studies investigating the evolution of the specific form of code-integrated specification that is relevant to persistence: class invariants. Class invariants describe relationships between attribute values. As attribute types and their values are stored by persistence mechanisms, there is a connection between class invariants and persistence. Furthermore, class invariants are also part of a class' documentation. Other forms of documentation—for example, comments—evolve together with code [61], APIs [82], or tests [121]. It seems natural then to investigate what happens to objects that need to be retrieved when their invariants evolve.

The research questions of interest target languages supporting Design by Contract, where programmers can formalize class invariants. In this context, we study the changes introduced by programmers to class invariants over multiple revisions, with the goal of answering the following questions:

4. Are the changes to class invariants frequent or infrequent?
5. If they are frequent:
   (a) Do they more often become stronger or weaker over time?
   (b) Are their typical changes compatible with the capacity to retrieve previously stored objects, or do they hinder it?
   (c) On a typical revision, do class invariants become stronger, weaker or stay the same when we add or remove class attributes?

4.1 Persistence-affecting changes in Java

The Java studies that played a major role in answering the first two research questions mentioned at the beginning of the chapter analyze two source code repositories, the *java.util* data structure and utilities library and the Apache Tomcat web server. The studies are intended to determine if class schema evolution actually happens in existing Java software projects. Section 4.1.2 provides an answer to the third research question by showing that the measure of robustness for the evolution of persistent applications defined in Section 3.6 can be computed, and provides useful indications for existing projects like the *java.util* package.

Earlier studies have already analyzed Java source code repositories looking for changes. Advani et al. [1] evaluated changes in fifteen open source Java systems and showed that "Rename field", "Move field", "Rename method", and "Move method", account for approximately 66% of the total identified changes. We notice that, in particular, "Rename field" and "Move field" alone account for 32% of the total. This result suggests that the evolution of fields does actually happen in realistic systems. Furthermore, given that changes in fields may influence the ability of a system to retrieve them later, and given that class schema evolution involves fields in 32% of the cases, it seems relevant to investigate the matter further. The cited study [1] calculates typical attribute changes on all classes of the considered systems. There is therefore no evidence that classes whose object are specifically tagged to be persistent would exhibit the same characteristics. The following section 4.1.1 tries to improve [1], and focuses on classes intended to produce objects that will be persisted. The classes, all from the Java package `java.util`, are checked to see if they evolve in a similar manner to non-persistent ones. Java code is an apt choice for this study because it is widely used and because classes that might have persistent instances are easily recognizable, as they are marked as `Serializable`.

4.1.1 *Analyzing* java.util

The *java.util* package contains classes modeling collections, dates, currencies, and locales. The classes taken into account here are all the 22 classes directly implementing the `Serializable` interface. The analysis extends across five versions of the language: 1.2.2, 1.3.1, 1.4.2, 5.0, 6.0. These versions can be considered significant steps in the evolution of the Java language, bearing important changes in both the APIs and their implementations [36]. The study considers 22 types of changes, seven of which are directly relevant to the serialization process and involve attributes (fields ac-

cording to the previous study [1]). While "Attribute added", "Attribute removed", "Attribute renamed", and "Attribute type changed" are straightforward, we consider "Attribute initialization value changed" because it may influence the class invariant. In addition, "Attribute becoming a constant" is relevant because constants are not serialized, and therefore it will be interpreted as "Attribute removed". In a similar fashion, "Constant becoming an attribute" will be interpreted as "Attribute added". Furthermore, class (static) attributes are not considered because they are typically not serialized. In line with what was discovered previously [1], Table 4.1 shows that the persistence-affecting changes constitute of 18% of the total (115 out of 649). are shown in Table 4.1

	attribute added	attribute removed	attribute renamed	attribute type changed	attribute value changed	attribute to constant	constant to attribute	non persistence-affecting	all
1.2.2 → 1.3.1	4	0	0	0	1	0	0	11	16
1.3.1 → 1.4.2	10	13	1	1	0	0	1	99	125
1.4.2 → 5.0	29	6	0	13	2	4	0	270	324
5.0 → 6.0	14	9	4	1	0	2	0	154	184
All	57	28	5	15	3	6	1	534	649
%	9	4	1	2	1	1	0	82	100

Table 4.1: Changes found across 5 versions of `java.util`.

Table 4.2 focuses on persistence-affecting changes, and shows that "Attribute added", "Attribute removed", and "Attribute type changed" together constitute 87% of all persistence-affecting changes (100 out of 115).

Figure 4.1 synthesizes Table 4.1 and Table 4.2 in a bar graph showing a comparison between the weight of each SMO with respect to all the changes and the weight of each SMO with respect to the persistence-affecting changes.

Table 4.3 shows the distribution of changes across different classes and across all five versions of the `java.util` package. The data suggest that persistence-affecting changes are sufficiently widespread among classes.

	attribute added	attribute removed	attribute renamed	attribute type changed	attribute value changed	attribute to constant	constant to attribute	all
1.2.2 → 1.3.1	4	0	0	0	1	0	0	5
1.3.1 → 1.4.2	10	13	1	1	0	0	1	26
1.4.2 → 5.0	29	6	0	13	2	4	0	54
5.0 → 6.0	14	9	4	1	0	2	0	30
All	57	28	5	15	3	6	1	115
%	50	24	4	13	3	5	1	100

Table 4.2: Persistence-affecting changes across 5 versions of `java.util`.

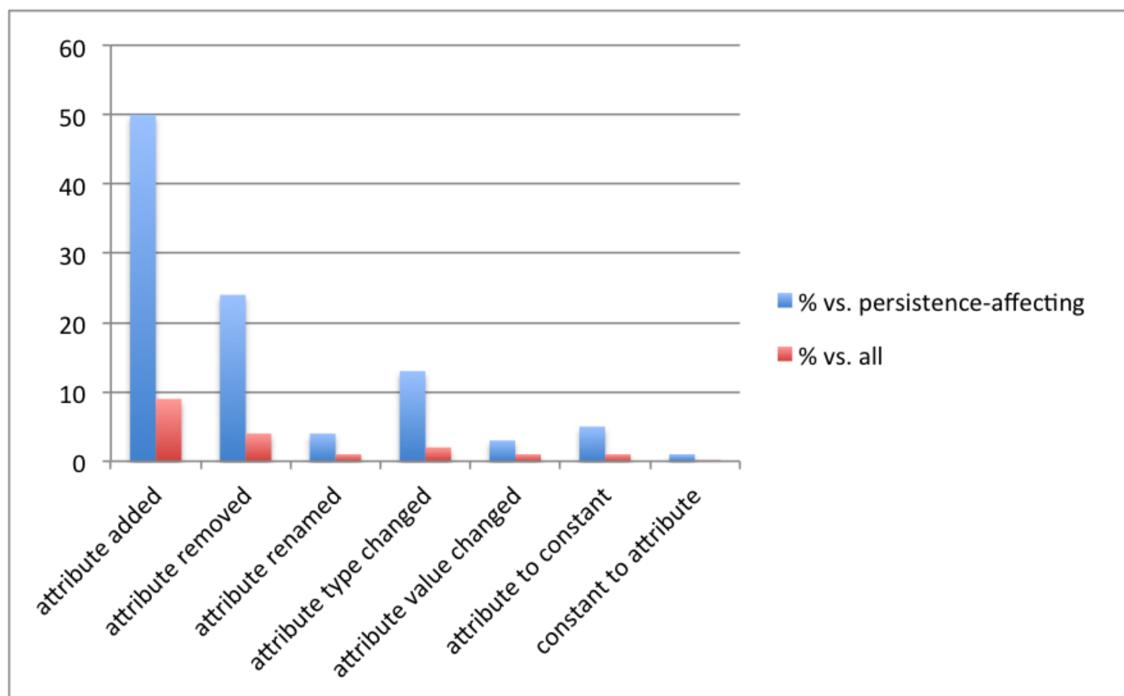

Figure 4.1: Single SMOs vs. all changes and vs. persistence-affecting ones.

To summarize, the study shows that classes whose instances might

Class	Changes	Persistence-affecting	%
ArrayList	18	3	17
BitSet	42	8	19
Calendar	52	24	46
Currency	3	2	67
Date	22	9	41
EnumMap	1	0	0
EnumSet	1	0	0
EventObject	1	1	100
HashMap	101	11	11
HashSet	9	2	22
HashTable	41	5	12
IdentityHashMap	20	2	10
LinkedHashSet	5	1	20
LinkedList	50	1	2
Locale	34	16	47
PriorityQueue	17	0	0
Random	13	7	54
TimeZone	28	7	25
TreeMap	122	13	11
TreeSet	39	3	8
UUID	0	0	0
Vector	30	0	0
All	649	115	

Table 4.3: Changes found in 5 versions of `java.util`, by class.

be serialized do change over time. Moreover, 18% of these changes are persistence-affecting, that is, they directly impact the capability of classes to deserialize instances of their previous versions. This might be aggravated by the high number of stored objects that may need to be updated.

As a telling example, in class *java.util.Calendar*, version 1.2, there are two version-related attributes, *currentSerialVersion* and *serialVersionOnStream*, in addition to the usual *serialVersionUID*. As explained in the code comments, these were used to solve the problem of keeping track of the different object versions, and had default values suggested. With the framework proposed by this thesis and its integrated version support, creating these two attributes would have proved unnecessary and the transformation function would have not been cluttering the class itself.

4.1.2 *The persistence evolution robustness of* `java.util`

In spite of the fact that Java implements a class-schema approach (as seen in Section 2.1), it is possible to compute the persistence evolution robustness (PER) measure defined in Section 3.6 for the *java.util* package. Table 4.4 shows the calculation for class *ArrayList*. Between versions 1.2.2 and 1.3.1 there are no changes, so we can assume that both the forward and the backwards transformations exist and work correctly. From version 1.2.2 to 1.4.2 there is an attribute added, for which only the forward transformation exists. The reverse transformation is not handled, and so deserialization of objects of version 1.4.2 from version 1.2.2 does not work, because removing an attribute is an incompatible change according to the Java Serialization Specification [102]. Proceeding in a similar way, and taking into account that an attribute type changed is also an incompatible change, we end up with a total of 4 existing transformations for *ArrayList*, out of 20 possible across the 5 considered versions. The resulting PER value is therefore 0.2. Table 4.5 shows the value of PER for every class in the *java.util*, and for the package itself.

The data suggest that computing the PER measure for a class can be a simple way to quantify how likely it is that the class will throw retrieval errors. Therefore the measure can be easily used to assess the "persistence evolution robustness" of a software class and compare it to the measures computed for other classes.

Version interval	Forward transformation	Backward transformation	Attribute changes
1.2.2 → 1.3.1	1	1	no changes
1.2.2 → 1.4.2	1	0	1 added
1.2.2 → 1.5.0	0	0	1 added, 1 type changed
1.2.2 → 6.0	0	0	1 added, 2 type changed
1.3.1 → 1.4.2	1	0	1 added
1.3.1 → 1.5.0	0	0	1 added, 1 type changed
1.3.1 → 6.0	0	0	1 added, 2 type changed
1.4.2 → 5.0	0	0	1 type changed
1.4.2 → 6.0	0	0	2 type changed
5.0 → 6.0	0	0	1 type changed
All	3	1	PER: 0.2

Table 4.4: Persistence evolution-robustness (PER) of `ArrayList`.

Class	Class PER
ArrayList	0.2
BitSet	0.4
Calendar	0.25
Currency	0.7
Date	0.3
EnumMap	1
EnumSet	1
EventObject	0.7
HashMap	0.1
HashSet	0.3
HashTable	0.3
IdentityHashMap	0.4
LinkedHashSet	0.4
LinkedList	0.4
Locale	0.2
PriorityQueue	1
Random	0.15
TimeZone	0.3
TreeMap	0.2
TreeSet	0.35
UUID	1
Vector	1
Package PER	0.48

Table 4.5: P-evolution-robustness (PER) of `java.util`.

4.1.3 *Analyzing the Apache Tomcat code base*

The Apache Tomcat Java open source project was first released in 1999. It implements a web server and includes Catalina as a servlet container, Coyote as HTTP connector and Jasper as JSP (Java Server Pages) engine.

Tomcat is an example of a well-known and widely used application. It is interesting in the context of this thesis because a web server typically needs to store information about shared values that need to survive the stateless HTTP protocol interaction.

This section analyzes the evolution of The Tomcat's classes implementing the *Serializable* interface, and takes into account the earliest and the latest releases available in the open source repository at the time of writing (3.0 from 2006 and 7.0 from 2012 respectively) [63]. Table 4.6 shows the classes in Tomcat's code base containing at least one schema modification operator (SMO). This happens in 12 out of the 27 classes analyzed. Consistently with what we observed in the case of `java.util`, "Attribute added", "Attribute removed", and "Attribute type changed" together constitute 93% of the persistence-affecting changes (compared to 87% in `java.util`). The data therefore confirm that the kind of SMOs found in the case of the `java.util` library classes were also found in a realistic and widely used software application. One additional observation is that there were not significantly more changes than it might have been

possible to expect from an application as opposed to a library project. This might be because every mature software project puts an extremely high value in stability and backward compatibility, and tends to limit the evolutionary changes to the bare minimum.

	attribute added	attribute removed	attribute renamed	attribute type changed	all
ValueReference	0	2	0	0	2
ApplicationParameter	0	0	0	1	1
FilterDef	1	0	0	0	1
FilterMap	2	0	2	2	6
MultipartDef	0	0	0	3	3
NamingResources	2	0	0	0	2
SecurityCollection	2	0	0	0	2
ServletDef	4	0	0	0	4
CsrPreventionFilter	4	0	0	0	4
AbstractReplicatedMap	2	0	0	0	2
FeatureInfo	2	1	0	0	3
DeltaSession	0	1	0	0	1
All	19	4	2	6	31
%	61	13	7	19	100

Table 4.6: SMOs found in Tomcat's repository.

4.2 Persistence-affecting changes in Eiffel

Though less widespread than Java, Eiffel provides language-integrated support for class invariants, and more than twenty years of source code in its repositories. These features make it a valuable asset for research in the field of persistence. This section describes the changes found in the EiffelBase standard data structures and utility library 4.2.1 and in the EiffelStudio IDE code base 4.2.2.

4.2.1 *Eiffel libraries changes*

The study analyzes 22 classes from EiffelBase, the standard data structures and utility library. Most of them are direct counterparts of the classes previously analyzed in Java. In the five cases in which it is not possible to find a direct counterpart, the study considers classes whose instances are likely to be serialized, such as strings and data structures. The analysis

spans five releases, but the period is wider in the Java study, ranging from 1993 to 2010 versus 1999 to 2009. The difference in period is due to different availability of data in the software repositories. The two programming languages are taken into consideration during a “maturity” period, after an initial stabilization period of four years.

The list of relevant changes for Eiffel persistence includes more types of changes: “Inheritance from one class added”, “Inheritance from one class removed”, “Class name changed”, and “Void safety clause added”. With respect to the Java study, this study does not consider ”Attribute initialization value changed” because in Eiffel assigning a value to an attribute in the declaration means making it a constant, which is typically not serialized. Furthermore, no instances of ”Attribute renamed”, ”Attribute becoming a constant” or ”Constant becoming an attribute” were found. Interestingly, however, there is a consistent number of inheritance-related changes, of which there was no trace in Java. This may be due to the fact that Eiffel allows multiple inheritance, making it easier to add and remove functionality by using inheritance. As seen in Section 3.1, the framework models inheritance by considering the flattened class schema. Therefore, the ”Inheritance added” and ”Inheritance removed” changes are subsumed into the other already analyzed ones. Finally, there are also some changes to attributes meant to enforce void-safety, a matter discussed in section 4.2.1.1. The kinds of changes found, together with their occurrences and across considered versions, are summarized in Table 4.7. Note that “Attribute added” and “Attribute removed” are still relevant, while “Inheritance added” and “Inheritance removed” together constitute 54% of all persistence-affecting changes.

Table 4.8 shows the distribution of changes in different classes and across all the considered five versions of Eiffel. As in the Java study, the data suggest that persistence-affecting changes are sufficiently widespread among classes.

The analysis unveiled an issue that could be easily solved by the framework presented in this thesis. In class *HASH_TABLE*, considered from revision 13752 to revision 47039, there was a change in the types of three attributes. This made it impossible to read previously serialized objects. The solution, implemented in a later version, was to code a transformation function using an ad hoc new attribute, `hash_table_version_57`. If the library maintainers had used the framework proposed by this thesis, there would have been no need to add any new attribute, and the transformation function would not have cluttered the class.

	attribute added	attribute removed	attribute type changed	inheritance added	inheritance removed	class name changed	void safety added	all
1993 → 2002	8	2	0	8	14	0	0	32
2002 → 2006	2	0	3	4	0	0	0	9
2006 → 2008	4	1	0	3	0	1	0	9
2008 → 2010	0	0	0	0	0	0	4	4
All	14	3	3	15	14	1	4	54

Table 4.7: SMOs found across 5 versions of Eiffel.

4.2.1.1 Void safety-related changes.

A program is void safe if it has no void dereferencing errors. A void (or null) dereferencing error happens when the operation `x.f()` fails because the target object `x` denotes a void reference at execution time [93]. Since release 6.1 (2007) the Eiffel language has incorporated the *void safety* mechanism into the compiler, requiring developers to indicate, for every attribute, whether it can accept void values with the keyword `detachable` or with the keyword `attached` (or no keyword as this is the default); see Appendix A. In the case of class `DATE_TIME_PARSER`, we discovered four changes related to void safety, in which the keyword `detachable` was added to four attributes. This change (from `attached` to `detachable`) is harmless under the point of view of software evolution, because an `attached` attribute, which is required to be non-void, will always be stored and so will comply with the weaker requirement of being `detachable` at retrieval time. Though it did not occur in our findings, the symmetric change, from `detachable` to `attached`, can in principle be dangerous. A `detachable` attribute stored as void will be considered as `attached` by the retrieving system. This will raise an exception at runtime, both in case of retrieval from storage and in any other case involving assignment at runtime, such as a call to an external module assigning void to an `attached` attribute. This finding, while related to the Eiffel language, can become relevant for the future evolution of languages like Java that do

Class	Persistence-affecting changes
Arrayed_list	2
Bit_ref	0
Date_time_parser	6
Array	1
Date	2
String_8	6
Tuple	3
Action_sequence	2
Array_2	0
Arrayed_set	0
Hash_table	12
Linked_tree	1
Linked_set	2
Linked_list	0
I18n_locale_manager	4
Priority_queue	0
Random	3
Time	2
Binary_search_tree	3
Binary_search_tree_set	4
Uuid	1
Arrayed_stack	0
All	54

Table 4.8: SMOs found across 5 Eiffel versions, per class.

not currently incorporate this feature.

4.2.2 *Analyzing the EiffelStudio code base*

EiffelStudio is an example of an Integrated Development Environment (IDE), a very common type of application that is likely to have persistence needs. EiffelStudio stores information about user preferences, and also data concerning automatically generated tests. The EiffelStudio IDE was initially called EiffelBench and was released in 1990. It was renamed to EiffelStudio in 2001 and was open-sourced in 2006. The study analyzes the evolution of persistent classes of the main EiffelStudio application code and of its automated testing facility called "AutoTest". We take into account the earliest and the latest releases available at the time of writing:

release 5.4 from 2004[1] and release 6.8 from 2011 [115].

Table 4.9 shows the classes in which at least one SMO was found. This occurred in 12 out of the 20 classes analyzed. The data again confirm what was found in the case of the Eiffel library classes, and are also consistent with what was found in the Java repositories.

	attribute added	attribute removed	attribute renamed	attribute type changed	inheritance added	class name changed	void safety added	all
Sd_config_data	8	0	1	0	0	0	0	9
Search_table	2	0	0	0	0	0	1	3
Sd_inner_container_data	0	0	0	2	0	0	4	6
Ev_split_area	1	0	0	0	0	0	2	3
Ev_split_area_i	0	1	0	0	0	0	3	4
Ev_widget_i	1	0	0	0	0	0	3	4
Sd_place_holder_zone	1	0	0	0	0	0	0	1
Sd_notebook_upper	4	3	0	2	0	0	0	9
Epa_test_case_info	0	0	0	0	0	1	0	1
Profile_information	0	0	0	1	0	0	0	1
Profile_set	0	0	0	36	0	0	0	36
Test_session_record	0	0	0	0	1	0	0	1
All	17	4	1	41	1	1	13	78

Table 4.9: SMOs found in EiffelStudio's repository.

4.3 Evaluating the persistence model on the Java and Eiffel repositories

Analyzing the history of classes along different revisions provides useful insights on the schema evolution process. The data analysis presented

[1] When EiffelStudio code was open sourced in 2006, the code from previous years was made available as well.

in the previous sections shows that the schema of persistent classes may evolve significantly both in software libraries and in applications.

To evaluate the formal model presented in Chapter 3, we check if it recognizes the 278 persistence-affecting SMOs (counting both Eiffel and Java SMOs) found in the analysis. The results are that the framework is able to identify the SMOs in 93% of all cases (259 out of 278). The unrecognized SMOs are "Void safety added" and "Class name changed", which is interpreted as a new class introduced into the system. "Attribute renamed" is counted as recognized because the framework, though not generating the full body of the transformation function, is aware of the different possibilities and generates a comment providing guidance to developers. It should also be observed that in all of the four code repositories analyzed all the instances of "Attribute renamed" SMOs were all true renamings, not to be interpreted as "Attribute removed" plus "Attribute added".

While the formal model scored well with respect to the four code bases analyzed, we don't know if its default implementation would be sufficient to deal with the majority of the scenarios. One practical problem occurs when developers have to deal with a custom storable form. This means that developers may want to choose which attributes to include among those which will be persisted. This is a realistic scenario for which the framework presented in this thesis does offer support in the IDE-integrated implementation, by means of a filtering option that also allows the choice of default values for retrieval (see Section 5.1).

Another possible issue is that we did not ascertain, in the case of the Java studies, whether the analyzed classes generated objects that were actually persisted. We believe we can safely assume so based on the following two observations: most of the classes are very often serialized data structures and utilities, like linked lists and arrays, and all the considered classes implement the *Serializable* interface, implying that their objects are meant—by the library designers—to be serialized in the first place.

The results above suggest that the model devised is sufficiently detailed to take into account a realistic set of SMOs in two different object-oriented programming languages.

In Chapter 3 we showed that the transformations functions are complete with respect to attribute modifications. Moreover, though no static guarantees are provided, we can state that given any two versions, and given a transformation function between the two, the tool ensures that the function pre- and post-conditions, and the retrieving class invariant, will hold at runtime provided assertion checks are enabled.

The importance of class invariants to avoid that potentially unsafe objects are accepted into the system has been mentioned already. Unfortu-

nately, class invariants are not yet widely used among developers outside of Eiffel, and other mainstream languages do not offer support for them. An obvious question is then: what can Java developers do to emulate the framework's mechanism enforcing class invariants? While analyzing the Java libraries, we discovered a possible solution in class *BitSet*, during the transition from version 5.0 to 6.0. The class authors introduced a method *checkInvariants()*, with the idea of enforcing some class-wide properties. This method is then invoked from every public method in the class, emulating the Eiffel invariant checking mechanism. Therefore the absence of an embedded Design By Contract language support (also for preconditions and postconditions) makes things more complicated for Java developers, but the example of class *BitSet* shows that emulating the invariant mechanism is possible.

In the worst case, developers can still bypass, actively or passively, all the checks mentioned. In case of an attribute added, for example, they could ignore the warning, accept the (possibly wrong) default provided, and in general not code an appropriate transformation function or class invariant. They could even code the right invariant but disable runtime checking of assertions. This means that while the framework supports developers in handling the class schema evolution of object-oriented software applications, it does not guarantee a completely safe class schema evolution. To improve this, the measure for evolution robustness of persistent applications (PER) may help in assessing how much trust we can put in every single class of a code release with respect to its robustness to schema evolution.

4.4 The Evolution of class invariants in connection with persistence

This section presents two novel studies on class invariant evolution. Our first study (4.4.1) focuses on understanding if class invariants actually evolve by analyzing (by hand) classes from a software data structure and utility library. As evidence of invariant evolution is found, the study further investigates if the invariants themselves become stronger or weaker over time. This is relevant for persistence because it may influence the application's capacity to retrieve previously stored objects. The complete set of data is available online [105].

Our second study [55] (whose lead author was our colleague C. Estler) investigates the evolution of class invariants more extensively and

in a semi-automatic fashion (using a tool). The study, presented in Section 4.4.2, suggests that though class invariants indeed evolve, their strength is more likely not to change in presence of added or removed attributes. This constitutes a potential threat for an application's capacity to retrieve objects, because developers may add attributes without correspondingly strengthening their class invariants, leading to accepting inconsistent objects into the system.

4.4.1 An exploratory study on class invariant evolution

To answer research question 4, a study analyzes 44 classes in the EiffelBase library, containing various data structures and utility classes. The selected subset includes commonly used classes and their complete inheritance hierarchy. The library itself contains 309 classes at the time of writing, so the sample amounts to 14% of the total. Each class is considered across 5 major revisions, spanning from 1993 to 2012. More complete data will be presented in Section 4.4.2.

A class invariant is a boolean expression, consisting in a label and a number of boolean clauses connected by logical operators. With this description, the class invariant changes considered are:

- Clause added.
- Clause removed.
- Clause changed.
- Label changed.

The change "clause added" is considered because an invariant can change its strength. It becomes stronger (more difficult to satisfy) when one or more conditions are added using an *and*; and it becomes weaker (easier to satisfy) when one or more conditions are added using an *or*.

For the change "clause removed" similar considerations apply: depending on the removed clause, the invariant can become stronger or weaker.

A clause is considered as "changed" when any of its code changed significantly across versions. This was established by manual inspection. Cosmetic changes like formatting changes (e.g. added spaces) and feature renaming that did not change the semantics were not considered. As an example, consider class `FINITE` between revisions 8120 and 31933, where clause `empty = (count = 0)` becomes `is_empty = (count = 0)`.

Revision interval	Clause added	Clause removed	Clause changed	Label changed	All
0 → 387	40	0	0	0	40
387 → 8120	28	2	1	1	32
8120 → 31933	13	8	12	0	33
31933 → 66985	1	5	1	1	8
66985 → 548	0	10	3	0	13
All	82	25	17	2	126
%	65	20	13	2	100

Table 4.10: Changes in class invariant clauses found across 5 versions of `EiffelBase`. The last small revision number is justified by an intervening repository migration.

Furthermore, as invariant labels are used to document informally the semantics of a clause and to provide information in case of an invariant violation, label changes were considered as well.

Table 4.10 shows the total number of class invariant clause changes across the five considered revision intervals, without taking inheritance into account, meaning that each class has been analyzed in isolation. To justify the apparent irregularity in the revision numbers we considered, the frequency of recording the changes increased considerably after the first period. The data shed some light on the first research question above, suggesting that class invariants do in fact evolve. Adding clauses is the prevalent activity, mostly concentrated in the initial development phases, and then becoming less and less frequent. Removing clauses seems to have the opposite trend, while changing labels does not appear to be particularly frequent.

Table 4.11 shows the distribution of invariant changes across different classes and across the five considered revision intervals. The data suggest that invariant evolution can vary significantly across classes, *HASH_TABLE* and *LINEAR* being extreme examples, having respectively 37 and just one change.

To answer research question 5.(a), Table 4.12 shows a summary of the evolution of class invariants in terms of relative strength, across the five considered revision intervals. The numbers are computed by considering how class invariants evolve across two consecutive revisions of each class, and then aggregated across all classes and revisions. The possibilities for invariant evolution that are taken into consideration across pairs of consecutive revisions are:

- Invariant strengthened.

Class name	Clause added	Clause removed	Clause modified	Label modified	All
Active	2	0	1	0	3
Any	2	0	0	0	2
Array	4	0	0	1	5
Array2	1	0	0	0	1
Arrayed_list	3	1	0	0	4
Bag	0	0	0	0	0
Bilinear	3	1	0	0	4
Bounded	2	0	0	0	2
Box	0	0	0	0	0
Chain	8	1	2	0	11
Collection	0	0	0	0	0
Comparable	0	0	0	0	0
Container	0	0	0	0	0
Cursor_structure	0	0	0	0	0
Dispenser	2	0	2	0	4
Dynamic_chain	1	1	0	0	2
Dynamic_list	0	0	0	0	0
File	3	0	1	0	4
Finite	2	1	1	0	4
Hash_Table	22	12	3	0	37
IO_Medium	0	0	0	0	0
Indexable	1	1	0	0	2
Iterator	1	0	0	0	1
Linear	1	0	0	0	1
Linear_iterator	3	2	0	0	5
Linked_List	5	0	2	0	7
List	2	0	0	0	2
Numeric	4	4	0	0	8
Part_comparable	0	0	0	0	0
Resizable	1	0	0	0	1
Sequence	0	0	0	0	0
Stack	0	0	0	0	0
Table	0	0	0	0	0
Traversable	1	0	1	0	2
Tree	8	1	4	1	14
Unbounded	0	0	0	0	0
All	82	25	17	2	126
%	65	20	13	2	100

Table 4.11: Changes in class invariant clauses found across 5 versions of `EiffelBase`, by class.

- Invariant weakened.
- Invariant undergoing a complex change, making it too difficult,

Stronger invariant	Weaker invariant	Complex invariant	Unmodified invariant	Total
59	25	7	89	180
33%	14%	4%	49%	100%

Table 4.12: Changes in class invariants across consecutive revisions of `EiffelBase`.

or impossible to analyze. By complex change we mean a change that leads to a non-comparable invariant (e.g. neither stronger nor weaker). A simple example is `x > 0` that evolves into `y > 0`, where `x` and `y` are unrelated attributes. These two invariants are clearly not comparable. The reason for such an invariant change could be a significant class implementation change.

If none of the above three cases happens between two consecutive revisions, the invariant is counted as unmodified.

Inheritance was considered as well: if class `B` inherits from `A` and there is a clause added to the invariant of `A`, the invariants of both `A` and `B` are strengthened as a result.

The data suggest that most of the time invariants do not change across versions. When they do, they are mostly strengthened; only in a few cases were the changes complex to assess. The conclusion: invariant strengthening, a potential issue for object retrieval, does happen. An example of an invariant too complex to assess was encountered in class `HASH_TABLE` between revisions 31933 and 66985: there the invariant changes amounted to one clause added, five clauses removed and 1 clause changed.

To answer research question 5.(b), in 50% of the cases class invariant evolution is not an issue for object retrieval. However, it may be an issue in 33% of the cases, because a strengthening of a class invariant between consecutive versions v_1 and v_2 makes it more difficult to retrieve objects stored in v_1 format from within objects in v_2 format, because there are more clauses to satisfy. As an example, in revision 387 of class `ARRAY` two clauses were added: `consistent_size: count = upper - lower + 1` and `non_negative_size: count >= 0`. Retrieving an object of type `ARRAY` of a version prior to 387 becomes therefore potentially more difficult because it might violate one of the two clauses above.

Table 4.13 shows what kind of clauses the class invariants are made of, for each class and across the five considered revision intervals.

It is interesting to notice that most of the invariant clauses analyzed (58%) are very simple, involving relational operators applied to numeric

Class name	x relational op y	$p/\neg p$	$predicate$	composite	$p \Rightarrow q$	All
Active	0	0	0	1	1	2
Any	0	2	0	0	0	2
Array	3	1	0	0	0	4
Array2	1	0	0	0	0	1
Arrayed_list	1	1	0	0	1	3
Bag	0	0	0	0	0	0
Bilinear	0	0	0	2	0	2
Bounded	2	0	0	0	0	2
Box	0	0	0	0	0	0
Chain	3	0	0	4	1	8
Collection	0	0	0	0	0	0
Comparable	0	0	0	0	0	0
Container	0	0	0	0	0	0
Cursor_structure	0	0	0	0	0	0
Dispenser	0	0	2	0	0	2
Dynamic_chain	0	1	0	0	0	1
Dynamic_list	0	0	0	0	0	0
File	2	1	0	0	0	3
Finite	2	0	0	0	0	2
Hash_Table	20	0	1	1	0	22
IO_Medium	0	0	0	0	0	0
Indexable	1	0	0	0	0	1
Iterator	1	0	0	0	0	1
Linear	0	0	0	0	1	1
Linear_iterator	3	0	0	0	0	3
Linked_List	0	1	0	4	0	5
List	2	0	0	0	0	2
Numeric	0	0	4	0	0	4
Part_comparable	0	0	0	0	0	0
Resizable	1	0	0	0	0	1
Sequence	0	0	0	0	0	0
Stack	0	0	0	0	0	0
Table	0	0	0	0	0	0
Traversable	0	0	0	0	1	1
Tree	5	0	0	2	1	8
Unbounded	0	0	0	0	0	0
All	47	7	7	14	6	81
%	58	9	9	17	7	100

Table 4.13: Type of clauses occurring in class invariants across 5 versions of `EiffelBase`, by class.

operands. Simple boolean expressions, predicates involving more complex computations, and implications amount altogether to 25% of the invariant clauses analyzed, while in 17% of the cases a combination of the

previously described items was found.

4.4.2 More results on class invariant evolution

The more recent study of our group [55] further investigates contract evolution in existing contracted code bases, performing an extensive empirical study of 15 open source software projects in Eiffel and C#. The study analyzes full-fledged contracts: preconditions, postconditions, and class invariants; this section restricts itself to discussing the results on class invariant strength that directly relate to persistence. The study presented in the Section 4.4.1 and the one presented in this section have a significant difference. In the previous study the data were collected entirely by hand and covered 5 revisions, whereas in the present study the data were collected mostly automatically and covered all the revisions. Both approaches are valuable: collecting data manually made a more precise analysis possible, because the invariants were inspected one by one, and further data about which kind of clauses made up the invariants were also collected and analyzed. Collecting data using a tool provided mainly the advantage of being able to process many more data from different projects. This allowed more confidence in the final results.

The Eiffel projects analyzed are the following:

- AutoTest, a contract-based random testing tool.
- EiffelProgramAnalysis, a utility library for analyzing programs.
- EiffelBase, a general-purpose data structures library.
- Gobo Kernel, a library for compilers interoperability.
- Gobo Structure, a data structure library.
- Gobo Time, a date and time library.
- Gobo Utility, a library to support design patterns.
- Gobo XML, a XML Library supporting XSL and XPath.

In Table 4.14 we report, for each Eiffel project, the total number of revisions, the life span (in weeks), the size in lines of code at the latest revision, and the number of developers involved.

The raw measures produced include, for each revision:

- The number of classes.

#	PROJECT	LANG	# REV	AGE	# LOC	# DEV
1	AutoTest	Eiffel	306	195	65'625	13
2	Eiffel Program Analysis	Eiffel	208	114	40'750	8
3	EiffelBase	Eiffel	1342	1006	61'922	45
4	Gobo Kernel	Eiffel	671	747	53'316	8
5	Gobo Structure	Eiffel	282	716	21'941	6
6	Gobo Time	Eiffel	120	524	10'840	6
7	Gobo Utility	Eiffel	215	716	6'131	7
8	Gobo XML	Eiffel	922	285	163'552	6
	Total		4'066	4'303	424'077	99

Table 4.14: List of Eiffel projects used in the study (Age is in weeks).

- The number of classes with invariants.
- The average number of invariant clauses per class.
- The number of classes modified compared to the previous revision.
- An estimate of whether a class invariant becoming stronger or weaker in time.

For invariant strength, the data show that for each project there is often a certain level of effort in writing class invariants to which developers commit early in the project's life and which remains fairly stable over time. In the first revisions it is common to have more varied behavior, corresponding to the system design being defined. However, the average strength of invariants (in terms of number of invariant clauses) typically reaches a plateau in the mature phases. This result refines what found in the study of Section 4.4.1.

This section tries to provide an answer to research question 5.(c), that is: do class invariants become stronger, weaker or stay the same when we add or remove class attributes?

Measuring precisely the strength of the invariant in an automatic fashion is very difficult, because it requires detailed knowledge of the semantics of a class and it may even require establishing undecidable properties. In our study, we use the number of invariant clauses (elements *and*ed, normally on separate lines) as a proxy for invariant strength. In particular, if we add a clause to a class invariant without changing its other clauses, we certainly have a strengthening (and, conversely, a weakening when we remove a clause). Of course there are scenarios in which just counting the clauses may measure strength incorrectly, for example when some clauses

are modified while no other clause is removed or added. The data analyzed [56], however, suggest that changes to existing specifications are infrequent events in a project's life if compared to changes to code; therefore, counting the number of clauses approximates strength to a good degree in practice.

Appendix C shows what happens to class invariant strength when attributes are added or removed, project by project and across all revisions. We summarize here some observations that are interesting for persistence:

- In AutoTest, revision 150 added 17 attributes while there is no detectable increment in the number of clauses of the corresponding invariants. This makes it potentially more difficult to retrieve object of this revision from objects of subsequent revisions, because there are not new invariant clauses describing constraints on the new attributes. Hence, the class invariant will likely not help with detecting inconsistent objects as they are retrieved.

- In Eiffel Program Analysis there were many attributes added and removed across revisions, but invariants stayed the same. The scenario is similar to the one described for the AutoTest project.

- In Gobo Structure, revision 170, there are 17 added attributes while there is no detectable increment in the number of clauses of the corresponding invariants. The scenario is similar to the one observed for the Autotest project.

- In Gobo Time, revision 64, there are 3 removed attributes with the invariant that does not change strength in term of number of clauses. This scenario does not pose a threat with respect to retrieving objects in revision 64 from objects in later revisions, but it does pose potential threats with respect to retrieving objects in revision 64 from objects in previous revisions, because the scenario would be similar to the ones above for added attributes.

- In Gobo XML there are quite a lot of changes, among which several revisions in which up to 12 attributes were added (e.g. in revision 550), but as the class invariants were strengthened as a consequence, it seems safe to assume that issues during object retrieval are not likely to happen.

Tables 4.15 and 4.16 describe the invariant changes when attributes are respectively added and removed. Each table refers to all the projects and is divided in three sections, each corresponding to an invariant behavior

(labeled as “same”, “strong”, and “weak”). For each measure we show minimum, median, maximum, standard deviation and sum across all revisions.

An observation valid for both tables is that the median is always zero, which means that on average (given that the median as the most representative measure of central tendency for asymmetric distributions) the invariants do not change when an attribute is added or removed.

The standard deviation does not vary wildly across the projects and is always relatively small. Still, it is interesting to observe what happens to the standard deviation when adding attributes and the invariant stays the same across revisions. It is possible to divide the projects in two groups, in each of which the standard deviation has the same order of magnitude. As shown in the leftmost part of Table 4.15, the standard deviation for projects like EiffelBase, Eiffel Program Analysis, Gobo Kernel, Gobo Time, Gobo Utility and Gobo XML is an order of magnitude smaller than for AutoTest and Gobo Structure. This means that AutoTest and Gobo Structure have more variability when attributes are added and the class invariant does not change across revisions.

Looking at the sum, it is interesting to notice that Gobo XML shows some significant invariant-related activities. Gobo XML has 184 classes which, for some pair of consecutive revisions, attributes are added and the invariant does not change (in terms of number of clauses), 94 classes in which attributes are added and the invariant becomes stronger, and 65 classes in which attributes are removed and the invariant does not change. This is confirmed by visual inspection.

	+A SAME					+A STRONG					+A WEAK				
Project	**m**	μ	**M**	σ	sum	**m**	μ	**M**	σ	sum	**m**	μ	**M**	σ	sum
AutoTest	0	0	17	1.29	117	0	0	1	0.06	1	0	0	1	0.06	1
EiffelBase	0	0	3	0.21	41	0	0	3	0.1	7	0	0	1	0.04	2
EiffelProgramAnalysis	0	0	3	0.5	44	0	0	1	0.07	1	0	0	0	0	0
GoboKernel	0	0	2	0.1	5	0	0	0	0	0	0	0	1	0.04	1
GoboStructure	0	0	17	1.06	34	0	0	2	0.22	12	0	0	1	0.06	1
GoboTime	0	0	1	0.13	2	0	0	0	0	0	0	0	0	0	0
GoboUtility	0	0	1	0.12	3	0	0	1	0.07	1	0	0	0	0	0
GoboXML	0	0	8	0.74	184	0	0	12	0.59	94	0	0	1	0.03	1

Table 4.15: Invariant changes when adding attributes, by project.

To be able to assess more precisely how class invariants change in strength over time we used a Wilcoxon signed-rank test to determine what happens when attributes are added to a class: does the class invariant be-

	-A SAME					-A STRONG					-A WEAK				
Project	**m**	μ	**M**	σ	sum	**m**	μ	**M**	σ	sum	**m**	μ	**M**	σ	sum
AutoTest	0	0	2	0.21	12	0	0	0	0	0	0	0	1	0.1	3
EiffelBase	0	0	14	0.41	31	0	0	1	0.05	3	0	0	2	0.06	3
EiffelProgramAnalysis	0	0	3	0.35	17	0	0	0	0	0	0	0	0	0	0
GoboKernel	0	0	1	0.04	1	0	0	0	0	0	0	0	1	0.04	1
GoboStructure	0	0	2	0.16	5	0	0	2	0.12	2	0	0	2	0.18	7
GoboTime	0	0	3	0.27	3	0	0	0	0	0	0	0	1	0.09	1
GoboUtility	0	0	0	0	0	0	0	0	0	0	0	0	1	0.07	1
GoboXML	0	0	5	0.36	65	0	0	1	0.05	2	0	0	5	0.27	33

Table 4.16: Invariant changes when removing attributes, by project.

come stronger or weaker, or does it stay the same? There is evidence that the invariant's strength not changing is more likely than the alternatives ($V = 0$, $p = 0.01$ in both tests, see the first two columns in Table 4.17). This is problematic for persistence because it suggests that developers often add new attributes to a class without correspondingly adding new invariant clauses for describing constraints on the new attributes. Hence, the class invariant will likely not help with detecting inconsistent objects as they are retrieved. Our measure of strength as the number of clauses is quite fitting in this situation, as new clauses will describe properties of new attributes that were not there previously.

We detected a similar effect—invariant not changing—when looking at revisions where attributes are removed (see the last two columns in Table 4.17). In this case, however, the negative consequences are limited to the arguably less common case of backward retrieval, where we want to restore an object stored with a newer version using an older version of its class.

Finally, we detected no difference between the non-flattened version (not taking inheritance into account) and the flattened version (taking inheritance into account) of the considered classes, so the tables refer to the flattened versions.

	$+$A inv weak vs same	$+$A inv strong vs same	$-$A inv weak vs same	$-$A inv strong vs same
V	0	0	3	0
p	$1.41 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$	$1.41 \cdot 10^{-1}$	$2.23 \cdot 10^{-2}$
$\Delta(\mu)$	$-2.3 \cdot 10$	$-1.65 \cdot 10$	-4	-7

Table 4.17: Attribute change analysis across all Eiffel projects.

4.5 Threats to validity

While the analysis of the results of the first four studies in this chapter (Sections 4.1.1, 4.1.3, 4.2.1, and 4.2.2) shows that the framework model and its implementation address in principle most practical challenges, the validation based on empirical data is still limited.

The current SMO detection and code generation could be dependent on the specific programming languages chosen (Java and Eiffel), as no similar investigation has been conducted on other programming languages' code bases.

The study on class invariant evolution in Section 4.4.1 compares the state of the invariant at each revision interval, therefore without taking into account the fact that clauses might have been added and then removed again (or vice versa) between two consecutive releases.

The raw measures taken in the study of Section 4.4.2 on class invariant evolution solve the issue with the previous study because they take into account all the revisions. There are, however, two potential threats to construct validity. First, using the number of clauses as a proxy for the strength of a specification may produce imprecise measures. While it has been clarified that the imprecision is an acceptable trade-off in most cases, it remains the fact that an exact computation of strength, while not feasible in general, might have been possible to compute is some cases. Second, the flattening introduced to study the effect of inheritance poses a potential threat to internal validity. This was fully addressed by analyzing all projects twice: in non-flattened and flattened version, and establish-

ing with high statistical confidence that there were not significant differences. A potential threat for all Eiffel-related studies is that classes whose instances are meant to be stored are not marked in any particular way. While this provides more flexibility for developers, it can also lead to considering classes whose objects were not meant to be stored. We addressed this threat by selecting, were possible, classes that have *Serializable* counterparts in Java.

The part of the study included in this thesis is restricted to class invariants in Eiffel. While other notations for contracts (e.g., JML) are similar to the considered ones, they were not analyzed. This may limit the generalizability of our findings. In contrast, the restriction to open-source projects does not pose a serious threat to external validity in the study, because several of the projects are mainly maintained by professional programmers.

Assuming that the idea and the current implementation are meaningful, general willingness to adopt the framework remains to be proved. In particular, there is no usability study trying to investigate developers' understanding and acceptance of such an IDE-integrated solution. There is no usability study for the IDE part of the framework because priority was given to validating the framework persistence API, which is arguably the first step towards adopting the whole approach in existing, realistic software projects.

CHAPTER 5

TOOLS AND LIBRARIES FOR EVOLVING PERSISTENT APPLICATIONS

The framework implementation, which consists of an IDE-integrated tool and an independently usable software persistence library, demonstrates how the model presented in Chapter 3 can be applied in practice.

While the approach described in this thesis can be applied to any object-oriented programming language providing support for storing and retrieving objects, using Eiffel as an implementation language and EiffelStudio as an IDE is motivated by the integrated support for *Design by Contract*, and in particular for class invariants [34]. Class invariants do in fact play an important role in complementing the framework's code generation support, because they provide a form of runtime validation for the object retrieval. This runtime validation is performed by the persistence library's retrieval algorithm, presented is Section 5.3.2.

The framework implementation as described in this thesis is integrated in *Eve*, the ETH Zurich Chair of Software Engineering's research branch of the EiffelStudio source code repository. The IDE integration was developed with the contribution of Schneider [111], while the persistence framework's library was designed and developed with the contribution of R. Schmocker [110], B. Meyer and the whole Chair of Software Engineering at ETH Zurich that participated to the design and code reviews.

In the main wiki page [57] there are instructions to compile the latest Eve sources for different platforms (Windows, Linux, Mac OS X), or alternatively use the latest available stable release. At the bottom there are links to the various sub-project directories containing the sources for the

IDE tool (named “Escher”), the object browser (named “Ebbro”), and the persistence library (named “Abel”).

Section 5.1 presents the IDE support offered by the framework. In particular, it describes the developers' interactions with the EiffelStudio GUI to create a software project release, the schema evolution handlers containing the transformation functions and the filters for the attributes. It then illustrates the framework's object browsing capabilities. Section 5.2 describes the framework code generation features. Section 5.3.1 describes the front-end API, validated in the experimental study presented in Chapter 6. Section 5.3.2 describes the retrieval algorithm. The rest of the framework implementation is described in Appendix B. More precisely, the appendix describes in detail the implementation of the Object-Relational Mapping (ORM) layer of the framework, how we automatically generate the database schema, the library support for transactions and errors, the support for additional relational databases, custom ORM mappings, non-relational stores, and other extensions.

5.1 IDE support

5.1.1 Schema evolution IDE support

The framework is seamlessly integrated into the EiffelStudio IDE, and its functionalities are accessible through a specific panel. The three numbered ovals in Figure 5.1 highlight some tool integration details. Oval 1 highlights the four main tool buttons, detailed below; oval 2 shows the automatically created folders for the different releases and the folder containing all the schema evolution handler classes; oval 3 shows the necessary libraries. The *serializer* library is a sample serialization library using a custom human-readable format. The *serializer_code_generator* contains the tool code generation facilities discussed in the remainder of this chapter.

The four buttons available to developers are:

- The *Release* button.
- The *Create Handler* button.
- The *Release Handler* button.
- The *Create Filter* button.

The *Release* button triggers a *release*: the classes are equipped with version numbers, if they don't have them already, otherwise their version

numbers are incremented. The versioning mechanism relies on the notion of release: a versioned, compiled and semantically coherent set of classes constituting a software system. Developers decide, by pushing the Release button, when a software system is ready for releasing. A class in a certain version can be found in several releases; this means that the class itself did not change across those releases. Different versions of the same class can only be part of different releases. The release version is increased if at least one class in the system has changed with respect to the previous release.

In addition, every time a developer releases a software system, the configuration files from the current project are copied to an appropriate release folder, whose name includes the current release number (see oval 2 in Figure 5.1).

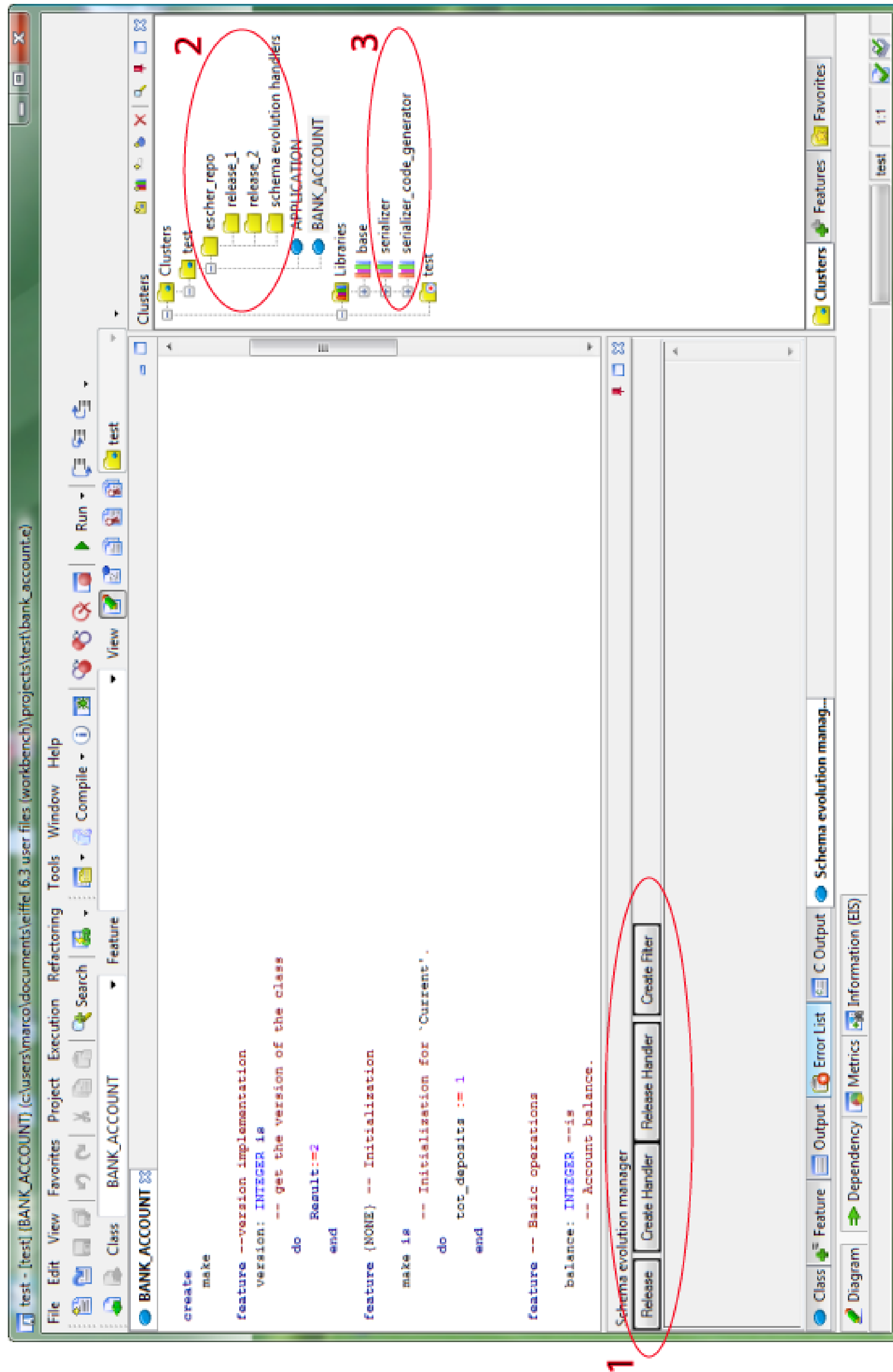

Figure 5.1: EiffelStudio IDE tool integration.

The *Create Handler* button creates a schema evolution handler. The process can be described in three steps. In the first step the tool visualizes a dynamic pop-up window asking for the pair of versions the user is interested in, as shown in Figure 5.2.

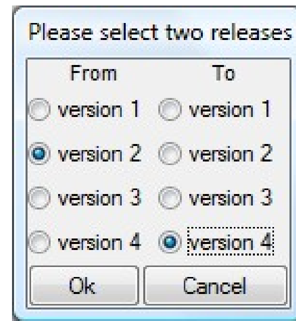

Figure 5.2: Version selection pop-up.

In the second step the tool creates the actual schema evolution handler, adds it to the list of handlers in class *SCHEMA_EVOLUTION_PROJECT_MANAGER*, and notifies the user with another pop-up (see Figure 5.3). The drop-down menu helps select the class of interest in the system.

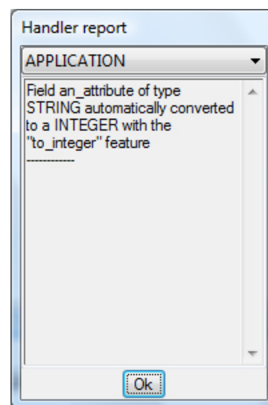

Figure 5.3: Schema evolution handler report.

In the third step the handler is copied in an ad-hoc *schema evolution handlers* directory, shown in oval 2 in Figure 5.1. Section 1.2 shows an example of generated schema evolution handler code including a transformation function.

The *Release Handler* button copies the newly created handler in the appropriate release directory.

Provided the required transformation functions exist, it will always be possible to retrieve any object of a certain class and version into an object of another version of the same class. This applies to both forward and backward transformations. The forward transformations, though arguably less frequent, are important as well, because customers might be running an old system (specified by an old release number) and in need of retrieving objects stored by a newer system release.

The *Create Filter* button creates a "filter", which allows developers to select the attributes they want to be part of the persisted form. This feature is important for any practical application of the tool, because there might be data that is not necessary, possible or efficient to store. The tool then asks about the release and class we are interested in (see Figure 5.4). and then allows choosing the individual attributes to filter, as shown in

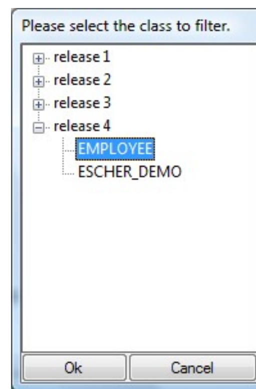

Figure 5.4: Selecting a release and a class for filtering.

Figure 5.5

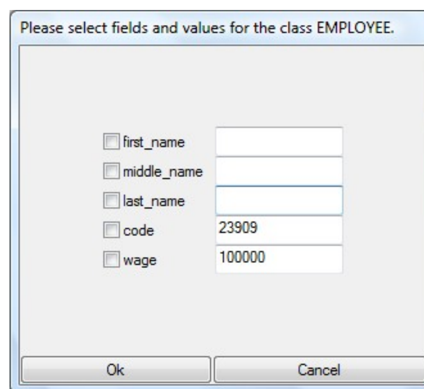

Figure 5.5: Filtering the attributes of a class.

5.1.2 *Serialization support: an object browser GUI*

Object serialization supports storing and retrieving objects in files using binary or textual format, or send them over the wire to remote machines.

Among the top ten commonly known programming languages in the TIOBE index [116], serialization support is limited to serialization libraries, including the corresponding documentation, and to debugging facilities. Microsoft Visual Studio 2010 offers an API to customize the way objects are displayed in the debugger [35]. In Java, apart from the non-GUI tools and plug-ins, DbVisualizer (a third party GUI tool) can display a tree view of serialized Java objects in BLOB columns, without allowing changes [113].

Given that many developers work using some sort of IDE, it seems natural to provide serialization support leveraging the IDE's capabilities. Therefore the framework presented in this thesis provides a browsing facility meant to load previously serialized objects in binary format, visualize their content, change it and possibly write them back, all from within the IDE itself and without any need to run an ad-hoc external program. Figure 5.6 shows how to start the Ebbro object browser from the EiffelStudio "Tools" menu. To use it, one only needs to have a file containing serialized objects stored on the local machine.

Objects can then be read and their content visualized using a simple GUI, as shown in Figure 5.7

Two objects can also be loaded together and compared to each other as shown in Figure 5.8.

The tool will outline the differences in red, while two content-wise identical object are marked with a green bar on top. Finally, apart from modifying the serialized form by changing the values of the attributes and then write them back, it is also possible to store a custom serialized form in which the user selects explicitly which objects to store, as shown in Figure 5.9

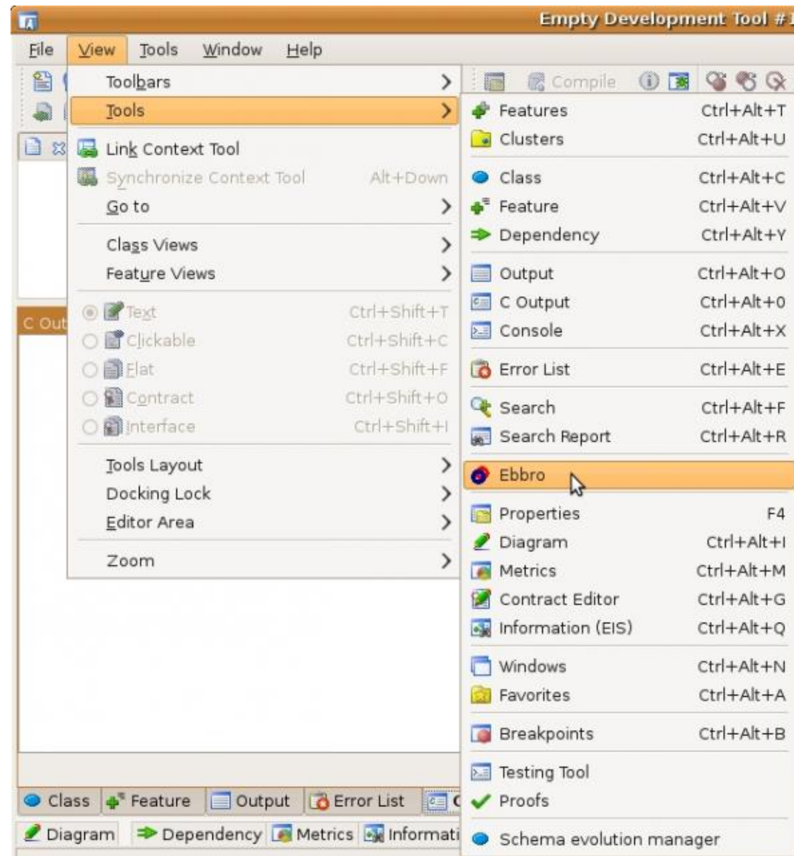

Figure 5.6: Starting the object browsing facility integrated into EiffelStudio.

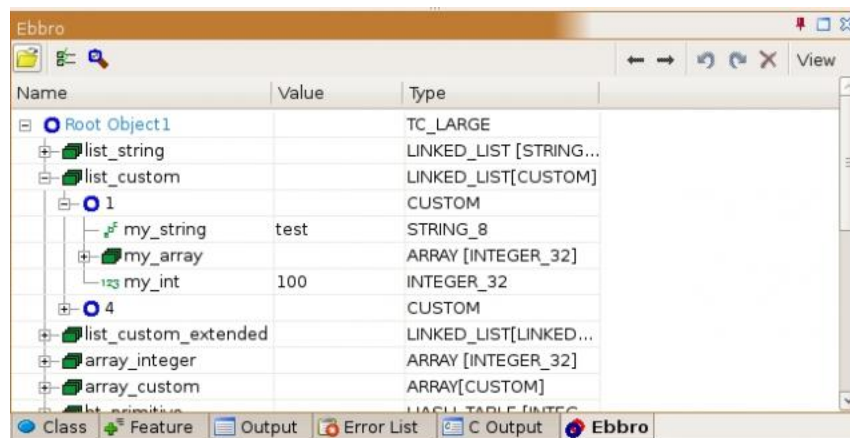

Figure 5.7: Object visualization GUI.

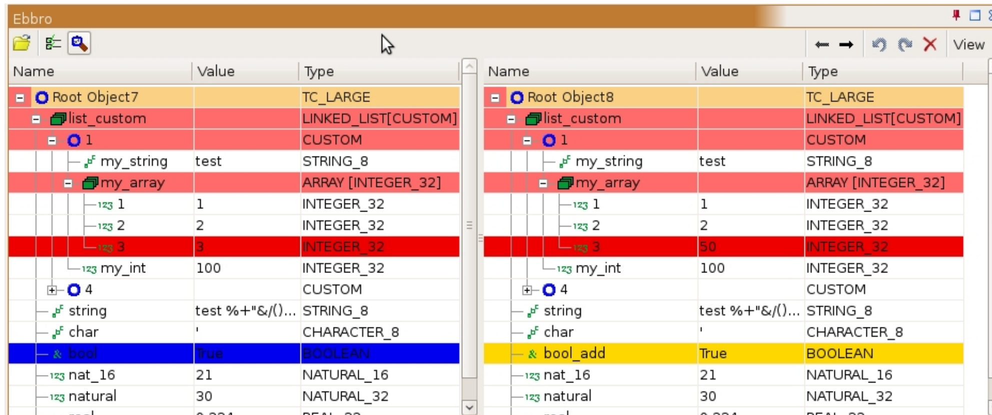

Figure 5.8: Object browsing facility integrated into EiffelStudio.

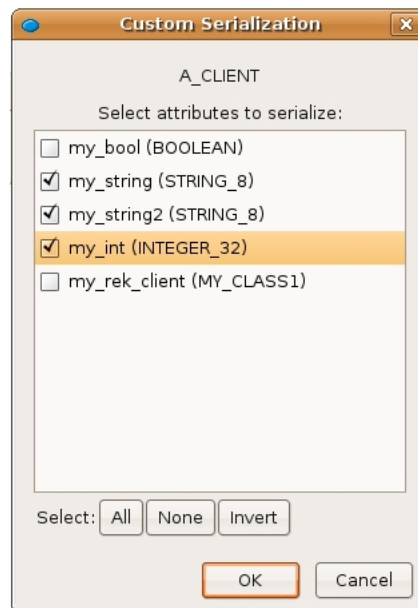

Figure 5.9: Object browsing facility integrated into EiffelStudio.

5.2 Code generation

The framework provides support for code generation to relieve developers from writing boilerplate code to transform instances stored in different versions with respect to the current one. It also lets developers focus on the actual specification of the transformation function rather than on the framework code necessary to wire all the transformation functions to the rest of the software system.

The framework generates code according to which IDE buttons are pressed (see Section 5.1), including the following:

1. *Release* button: a new release folder is created; classes in the new release are equipped with version numbers, if they don't have them already, otherwise their version numbers are incremented; the configuration files from the current project are copied in the new release folder.

2. *Create Handler* button: a new schema evolution handler is created, added to the list of handlers in class `SCHEMA_EVOLUTION_PROJECT_MANAGER`, and copied into a *schema evolution handlers* folder.

3. *Release Handler* button: the newly created schema evolution handler class is copied in the appropriate release folder.

The most important code generation step occurs when the framework generates a transformation function between two class versions v_1 and v_2 (step 2 above). The step involves a comparison between the abstract syntax trees of the two class versions, looking for known SMOs. To detect the SMOs, it first iterates through the v_2 class attributes, searching for a match with attributes in the v_1 class. Then it repeats the process starting from the v_1 class, in order to gather more information, for example to find all the attributes that were removed. When a known SMO is detected, the framework generates a transformation function template according to the rules in Section 3.5. The generated code and comments is as follows :

1. "Attribute not changed": apart from assigning the stored value to the attribute, the template includes a warning about the possible semantic change.

2. "Attribute type changed": the template includes partial or complete code generation depending on the existence of predefined type converters. As an example, consider the code shown in Section 1. There we see that attribute `info: INTEGER` in version 1 becomes `info: STRING` in version 2. The type converter automatically applied is in this case `out` from class `INTEGER`. For the opposite conversion (from `STRING` to `INTEGER`) the type converter applied would have been `to_integer` from class `STRING`. It is possible to define and apply custom type converters.

3. "Attribute added": the template includes default initialization for the newly added attribute and a warning to check the new class invariant with respect to the old one to avoid retrieval failures.

4. "Attribute removed": the template includes a warning about the possible semantic change.

5. "Attribute renamed": the template includes a warning about a possible "Attribute removed" followed by an "Attribute added".

6. "Attribute made attached": the template includes a warning about the necessity to initialize the attribute. A postcondition checks that this has happened.

After a developer has analyzed, possibly modified the transformation function body and finally saved it, the tool places the transformation function into the *MY_CLASS_SCHEMA_EVOLUTION_HANDLER* class mentioned above, containing all the existing transformation functions between pairs of different versions for that class. This helps avoid polluting the code of *MY_CLASS* with evolution-related code, and makes the whole approach more scalable. Comments, hints and informational messages are always generated to guide developers through the process. In Chapter 1 there is an example of generated code and comments.

5.3 The persistence library implementation

At the time of writing the library is made of 81 classes grouped in 11 clusters (roughly equivalent to packages in Java). Apart from supporting the schema evolution framework with a retrieval algorithm, the library's API tries to achieve a trade-off between ease of use and flexibility, and ultimately provide a seamless experience when accessing different kinds of data stores. The first challenge for the API design was to offer a unified interface to different kinds of data stores. By splitting the API in a front-end API and a back-end API we followed Alan Kay's suggestion to "make simple things easy, and complex things possible" [81]. The front-end provides access to the main features of the framework, targeting the average user and the most common scenarios. The back-end provides object-relational mapper capabilities and support for more sophisticated tuning and adaptation, targeting therefore the advanced user. The boundary between front-end and back-end is represented by the *REPOSITORY* abstraction.

Section 5.3.1 describes the front-end API, validated in the experimental study presented in Chapter 6. Section 5.3.2 describes the retrieval algorithm. The rest of the framework implementation, and in particular the back-end API and support for extensions, is described in Appendix B. More precisely, the appendix describes in detail the implementation of the Object-Relational Mapping (ORM) layer of the framework, how we automatically generate the database schema, the library support for transactions and errors, the support for additional relational databases, custom ORM mappings, non-relational stores, and other extensions.

5.3.1 Front-end API

Figure 5.10 and Figure 5.11 show the most important front-end API abstractions. As it can be see from the class diagrams, they are relatively few, and they reflect our best effort to have a minimalistic and easily usable front-end API.

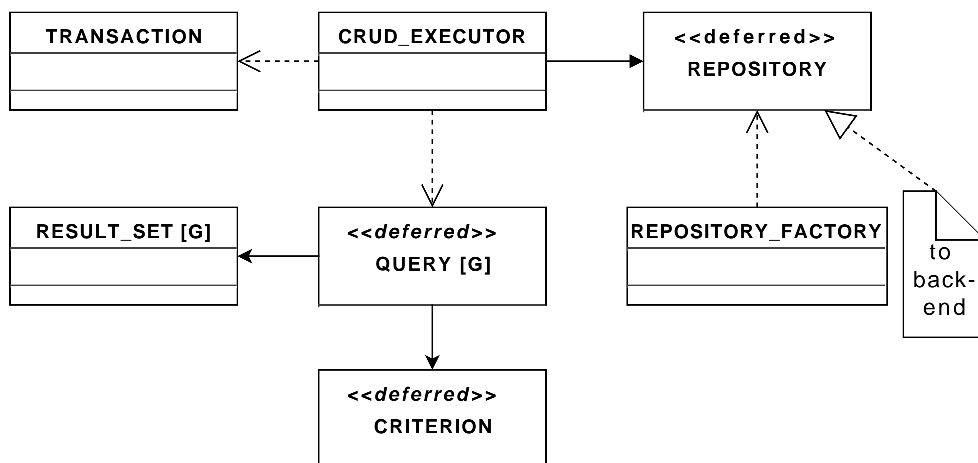

Figure 5.10: The main front-end classes and their relations.

Below there is a list of the main front-end classes, together with a short description:

- *REPOSITORY_FACTORY*: utility class providing ready-to-use database-specific *REPOSITORY* implementations.
- *CRUD_EXECUTOR*: executors of CRUD (Create Read Update Delete) operations.

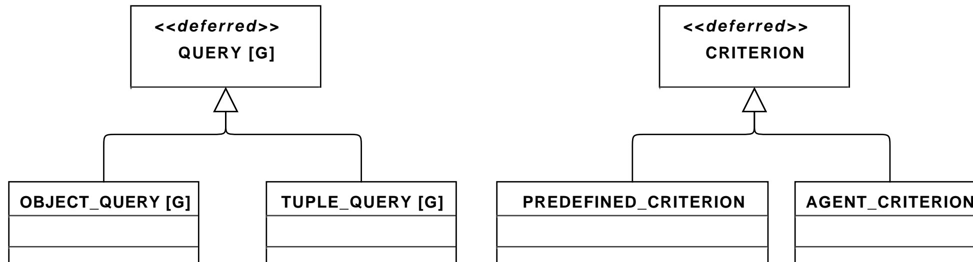

Figure 5.11: The *OBJECT_QUERY [G]* and *CRITERION* class hierarchy structures.

- *QUERY [G]*: generic repository-agnostic queries using criteria objects for filtering. Available implementations (see Figure 5.11):
  - *OBJECT_QUERY [G]* object-oriented style queries.
  - *TUPLE_QUERY [G]* tuple-based queries.
- *CRITERION*: combinable criteria for object selection, used to filter the query results. Available implementations (see Figure 5.11):
  - *PREDEFINED_CRITERION* criteria using string triplets to select attributes, values and operators of interest.
  - *AGENT_CRITERION* criteria using predicates (boolean functions) to encapsulate the needed filtering behavior.
- *TRANSACTION*: transaction facilities, supporting ACID (Atomicity, Consistency, Isolation, and Durability) properties when available in the back-end.

With the exclusion of *REPOSITORY_FACTORY*, the other items in the list describe classes that are back-end-agnostic, as they only depend on the *REPOSITORY* abstraction.

To give an idea of the usage of the front-end API, Listing 5.2 shows the code a programmer would need to write to perform a simple query on a MySQL database. We are assuming that a MySQL database is up and running, and that some objects of type *PERSON* have already been created and inserted (see Figure 5.1).

```
class
  PERSON
```

```
feature -- Status

  name: STRING
    -- Name of current person.

  age: INTEGER
    -- Age of current person.

  -- remainder omitted
end
```

Listing 5.1: Domain class *PERSON*

```
class MYSQL_TEST

create
  make

feature  -- Test features

  test_mysql
      -- Test code.
    local
      repo_factory: REPOSITORY_FACTORY
      mysql_repo: RELATIONAL_REPOSITORY
      executor: CRUD_EXECUTOR
    do
      -- Create the repository factory.
      create repo_factory
      -- Create a MySQL repository.
      mysql_repo := repo_factory.
          create_mysql_repository_with_default_host_port ("
          dbname","userid","password")
      -- Create an executor for CRUD operations.
      create executor.make (mysql_repo)
      -- Query the database for persons and print result
      print_result (query_with_composite_criterion)
    end

feature -- CRUD operations

  query_with_composite_criterion: LINKED_LIST [PERSON]
    -- Query the database using a composite criterion and
       get a list of persons as result.
```

```
    local
      query: OBJECT_QUERY [PERSON]
    do
      create Result.make
      create query.make_with_criterion (
          composite_search_criterion)
      executor.execute_query (query)
      across query as query_result
      loop
        Result.extend (query_result.item)
      end
    end

feature -- Queries with criteria

  composite_search_criterion : PS_CRITERION
    -- Combining criteria.
    local
      crit_factory: PS_CRITERION_FACTORY
    do
      create crit_factory
      -- We are looking for a person named Poldo that is not
           older than 40.
      -- Different flavors of criteria can be composed using
           logical operators
      Result := crit_factory[[ "name", "=", "Poldo" ]]
        and not crit_factory[[ agent age_more_than (?, 40) ]]
    end

feature -- Utility routines

  age_more_than (person: PERSON; age: INTEGER): BOOLEAN
    -- Age check on 'person' used as an agent routine.
    require
      age_non_negative: age >= 0
    do
      Result:= person.age > age
    end

  print_result (lis: LINKED_LIST [PERSON])
    -- Utility to print a query result.
    do
      across lis as local_list
```

```
      loop
        io.new_line
        print (local_list.item.name + " ")
        print (local_list.item.age)
      end
      io.new_line
    end
end
```

Listing 5.2: Querying a MySQL database

It is worth noticing that for the most common scenarios developers interact with the framework exclusively through the front-end API, which is a high-level object-oriented interface that does not allow any SQL command to be passed. In fact developers do not even need to know any of the SQL commands that are commonly used to access relational databases. These commands typically come in the form of SQL strings, and strings tend to be error-prone (as for example happens when escaping SQL strings from within the host language strings). Besides, all the errors will be only caught at runtime, after the command has been sent to the database, which brings us to the next important point. Sending SQL strings to the database can be a potential security threat coming in the form of an SQL injection: ad-hoc malicious code inserted into the SQL string itself [71].

5.3.2 *The retrieval algorithm*

At retrieval time, the persistence library relies on an ad hoc algorithm that is aware of the existing transformation functions.

Assuming that we have a stored object of class `C` in version *v1* and we want to retrieve it from within an object of class `C` in version *v2*, Listing 5.3 presents a simplified pseudo-code description of the algorithm that is applied.

```
if v1 = v2
then
    perform_standard_retrieval
elseif not schema_evolution_handler_exists_for_type
    then
          raise exception ("Schema evolution handler for
             class C does not exist.") -- 1
elseif not transformation_function_exists (v1, v2)
    then
          raise exception ("Incompatible versions") -- 2
else if not attribute.is_convertible
```

```
   then
         raise exception ("A type converter for the
            attribute does not exist") -- 3
else
   perform_across_version_retrieval
end
```

Listing 5.3: Algorithm for object retrieval.

In case of retrieval of objects of a certain class `C`, the algorithm will raise an exception when any of the following conditions are met:

1. The specific schema evolution handler for `C` does not exist. This is the case in which developers, while using the IDE and releasing a new version of the system, deliberately chose not to create a schema evolution handler for that class (see Section 5.1). As a remainder, a schema evolution handler is a framework class associated to a domain versioned class (`C` in this case), containing all the existing transformation functions between versions of the class. The algorithm can easily look for the existence of such a handler because its name is bound to the class name, so in the case of class `C` the handler's name is *C_SCHEMA_EVOLUTION_HANDLER*.

2. The specific transformation function between the two versions (retrieving object's class version and stored object's class version) does not exist;

3. At least one of the required type transformers for the single attributes does not exist. For a concrete example see the discussion about "Attribute type changed" in Section 5.2.

After the execution of the transformation function body, the class invariant is checked. A class invariant failure triggered at this point would mean that the new objects initialized with the old values do not satisfy what the new invariant demands from them. This step provides a substantial contribution to the effectiveness of the approach, because an invariant check failure means that inconsistent objects are not accepted into the system. Unfortunately the number of potentially inconsistent objects that are accepted into the system cannot be always reduced to zero because the class invariant can be too weak and let some objects pass even if they violate the real, albeit unexpressed, class invariant.

CHAPTER 6

EVALUATING THE DESIGN OF THE PERSISTENCE API

It is customary for software developers to learn new APIs. Devising APIs that are easy to learn is therefore an important design target. This chapter investigates the usability of the framework front-end API detailed in Section 5.3.1 by describing an exploratory study in which developers have to solve some programming tasks using the framework front-end API.

Section 6.1 describes background work on API usability that was helpful to outline the guidelines for the user study. Section 6.3 lists the research questions. Section 6.4 describes the pool of participants. Section 6.5 explains the study set-up. Section 6.6 details the data collection protocol used. Section 6.7 synthesizes the results of the study and analyzes the collected data. Section 6.8 analyzes the answers to the questionnaire answered by participants at the end of the study. Section 6.9 describes the lessons learned from the study. Finally, Section 6.10 discusses threats to validity.

6.1 Evaluation guidelines

Every API exposes a set of features that we can see as services offered. There are many possible deficiencies of API design that can make it difficult to benefit of its services, including [91]:

- The API is difficult to learn.
- Functionality is missing.
- Functionality is incomplete.

- Functionality is not useful.
- Functionality is not immediately clear from the API description.
- Functionality is too difficult to use.
- There is a lack of consistency.
- There is a wrong level of abstraction.

To address the issues above, and more, the Microsoft .NET API designers have successfully implemented a user-centric, scenario-based design process, in which they compare the experiences of software developers using the API against a checklist of twelve *cognitive dimensions*. Clarke et al., in particular, adapted the checklist from the *Cognitive Dimensions of Notations* framework from Blackwell et al. [13], originally used to describe the usability of notational systems, to software API usability. Here is the cognitive dimensions checklist [21]:

- *Abstraction level*: what level of abstraction the API exposes to developers.
- *Learning style*: how developers can learn the API.
- *Working framework*: what developers need to know to work effectively.
- *Work-step unit*: how much of a programming task must or can be completed in a single step.
- *Progressive evaluation*: to what extent partially completed code can be executed to obtain feedback on code behavior.
- *Premature commitment*: how many decisions developers have to make in a scenario and their consequences.
- *Penetrability*: how the API facilitates exploration, analysis and and understanding of its components.
- *API elaboration*: how much the API has to be adapted to meet the needs of targeted developers.
- *API viscosity*: how much the API is resistant to change.
- *Consistency*: how much of the rest of the API can be inferred once a part of it is known.

- *Role expressiveness*: how apparent the relationship is between each component exposed by the API and the program as a whole.
- *Domain correspondence*: how clearly the API maps to the domain of interest.

The idea of the Cognitive Dimensions of Notations framework is to compare the expectations of the API users with respect to what the API actually provides. This thesis follows this approach, using a questionnaire (among other tools) to collect feedback from users about their experience in using the API. The questionnaire, detailed in Section 6.6, is extracted by the items above.

6.2 Empirical answers to API questions: example results from previous studies

A previous study about the relationship between software developers and unfamiliar APIs [52] shows that developers have issues with tasks like the following:

- Associating API class names to functionalities.
- Discovering relationships between API types.
- Creating objects without public constructors available.
- Determining the outcome of method calls.

It does not come as a surprise that associating API class names to functionalities, that is, finding the right abstractions and naming them appropriately, may be a difficult task. It is more interesting that even when the class names cannot be used as keywords in search queries to locate relevant code examples because there is no internet access, developers can still use names as hints for educated guesses on the related functionalities. The issue of selecting an appropriate abstraction for the task at hand creates then potential selection barriers [83].

We may also expect that it will be difficult to discover relationships between API types: Stylos et al. [118] confirmed that a type is difficult to discover when is not mentioned—either as attribute, or local variable, or function argument, or even in a comment—from within the type developers are using.

The issues deriving from creating objects when there are not public constructors available, which typically means discovering an appropriate factory class, are similar. Ellis et al. [53] questioned the usage of the concrete factory pattern [66], undoubtedly one of the most widely used design patterns in existing software libraries. As a better alternative to the concrete factory they suggest the abstract factory, as for example implemented in the "Class Cluster" design of the Cocoa framework [78][77].

A somewhat surprising study suggests that argumentless constructors are preferred over constructors with arguments [117]. Argumentless constructors enforce a create-set-call style, versus the create-call style enforced by constructors with arguments. In one of the task results, all the participants designed a File class including a default constructor, allowing the possibility for a File to exist in an inconsistent state (without a file name). Based on the result of the experiment, the authors in the final discussion suggest that it is good to favor argumentless constructors over constructors with arguments. To the author of the thesis this constitutes an example of a clash between usability and correctness. If we accept that a file object does not make much sense without an associated name, we implicitly accept that the File class has an invariant (possibly explicitly) stating that, for example, the string attribute representing the file name exists and is not empty. An argumentless constructor clearly violates the previous assumption, so it should not be allowed. This is probably a scenario in which the API developer should act as a "benevolent dictator" and impose a restrictive choice for the better good, even if it goes against usability. In conclusion, striving for API usability may be sometimes error-prone and potentially dangerous, so it has to be handled with care.

Observing developers' difficulties in determining the outcome of a method call, as shown by Duala-Ekoko and Robillard [52], is also interesting. One of the conclusions is that in certain scenarios (e.g. a method validating a string) developers expect a return value from a method call to provide some feedback on the success or failure of the call itself. When they realize the method has no return value, it becomes hard for them hard to figure out where to look for the equivalent information. This is also a potential usability issue in a language like Eiffel that suggests a separation between commands (changing the state of an object), and queries (providing information about an object).

The thinking aloud protocol, used in the experiment for data collection, is probably the most widely used method in usability testing, and originates from Ericsson and Simon's seminal work [54], in which they suggest some quite restrictive rules for data collection. When trying to apply the method to usability tasks more centered on the product under test

than on the participant, it is more appropriate to adapt the rules mentioned above to use a speech communication model, which is less strict and allows to collect a more interesting set of data [16]. Being less strict means that certain forms of interaction between the participant and the observer are allowed, like in the case in which there is a technical problem, or when there is a request of help on a topic that is unrelated to the task under test, for example remembering what is the Eiffel syntax for loops.

6.3 Research questions

The research questions about the front-end API this section investigates are motivated by findings in the previous work described in Section 6.1:

- Which keywords best describe a functionality provided by the API?
- How do developers assess the API with respect to how easy is it to discover relationships between API types?
- How do developers assess the API with respect to object creation? Do they find it problematic when public constructors are not available, but factories are provided instead?
- Do developers prefer argumentless constructors to constructors with arguments?
- Do developers manage to always determine the semantics of the outcome of a method call?

The way the study uses the cognitive dimensions framework is not to "prove" the usability of the front-end API. Rather, the purpose is to discover the existence of issues, in the same way in which testing cannot prove a program correct, but can prove the existence of bugs. This is in line with the accumulated experience of researchers that have been using the framework for up to 15 years [44].

6.4 Participants

Before describing the participant pool we briefly summarize the tasks, to make it easier to assess whether the characteristics of the participants are suitable for the tasks. More details on the tasks can be found in Section 6.5.

Each participant was asked to complete five programming tasks. The tasks consisted in implementing some standard operations against an existing MySQL database, using the front-end object-oriented API, and required the discovery of the fundamental API abstractions, the combination of different objects and of specific ways to create them. These are items that previous studies found to be critical [118] [53]. More specifically:

- Task 1 involved object creation, discovering a factory class, and finding a class to insert objects into the database.
- Task 2 involved finding a class to handle queries, create an appropriate type of query, execute it and print its results.
- Task 3 involved finding a way to update an object in memory and in the database.
- Task 4 involved finding a way to delete an object both from memory and from the database.
- Task 5 involved finding a way to express a more complex query involving *criteria*. This meant finding the criterion abstraction, creating a concrete criterion and use it to get the right result from the query.

The participant pool is described in Table 6.1, which lists 25 participants, of which 20 males and 5 females, divided according to their qualifications as follows:

- 6 bachelor students from ETH Zurich. In their first semester at ETH, bachelor students are exposed to the Eiffel language through an Introduction to Programming course [101]. Afterwards they may choose to attend courses that require to design and implement applications in Eiffel.
- 4 Master's students, also from ETH.
- 7 PhD students, of which 5 are computer science students, and one is a mechanical engineering student.
- 2 post-doctoral researchers, one in computer science and one in mechanical engineering.
- 6 professional programmers working for software companies.

Participant tag	Current occupation	Years Eiffel experience	Years O-O experience	Years professional industry experience	Time spent in min
P1	Industry	18	22	21	43
P2	Industry	4	4	3	49
P3	PhD	1	12	2	57
P4	PhD	0	7	7	69
P5	PhD	0	16	15	70
P6	Bachelor	1	4	0	89
P7	Postdoc	0	7	5	72
P8	Industry	1	10	6	78
P9	Bachelor	1	5	0	103
P10	PhD	1	5	3	97
P11	PhD	2	10	1	53
P12	Industry	4	10	7	59
P13	Industry	11	12	12	48
P14	Bachelor	0.5	6	0	58
P15	PhD	0	1	0	105
P16	Industry	7.5	13	10	65
P17	Master	5	5	1.5	72
P18	Postdoc	0	3	1	104
P19	Bachelor	1	4	0	61
P20	Bachelor	2	5	0	87
P21	Master	1	3	0	42
P22	PhD	3	12	4	32
P23	Master	4	4	0.5	71
P24	Bachelor	1	4	0	118
P25	Master	3	5	0	81
Mean		3	8	4	71
Median		1	6	2	69

Table 6.1: Participant pool information.

No compensation was provided. The participants were recruited either via email or directly asking them if they wanted to participate. The only technical prerequisite was to have at least one year of object-oriented programming experience, in Eiffel or in another O-O programming language. All those who accepted satisfy these criteria. The participants reported

- 0 to 18 years of Eiffel experience.
- 1 to 22 years years of experience with an object-oriented programming language.
- 0 to 21 years of professional programming experiences.

As the experience appears to be an important factor of variability, we controlled for it is the analysis of results shown in Section 6.8. Most of the participants were familiar with Eiffel and the EiffelStudio IDE. Those not familiar with Eiffel and the EiffelStudio IDE were provided a 15-minute crash-course before starting the actual study. This included showing the basics of the Eiffel language and the EiffelStudio IDE main features. While none of the participants was familiar with the specific front-end API object of the experiment, all of them declared to be at least minimally familiar with relational databases. There were not pre-set time restrictions to solve the tasks, because the focus was not on measuring how fast developers were, but on how they proceeded in discovering an unknown API. Nonetheless, we measured how long it took for each participant to complete all the tasks. This varied from 42 minutes to 1 hour and 58 minutes, with a mean of 71 minutes and median of 69 minutes.

6.5 Study setup

The study was conducted through individual sessions between the author and one participant. During the study, internet access was disabled. The only documentation available to the participants was the class header comments and the feature comments in the code. A previous study suggested that the lack of resources like code snippets and tutorials was a significant obstacle to learning APIs [109]. While the front-end API provides a tutorial and some technical documentation, this was not made available because the study intended to focus on the obstacles related to the design of the API.

The framework API used for the experiment had 81 classes grouped in 11 clusters (roughly equivalent to packages in Java) 5.3. The participants

were made aware of the features of the EiffelStudio IDE which enabled class and feature browsing, showing the ancestors, descendants, clients and suppliers of a class.

Each participant was asked to complete five programming tasks. The tasks consisted in implementing some standard operations against an existing MySQL database, using the front-end object-oriented API and some objects of class *PERSON* (see Listing 6.1). Class *PERSON* is a simple domain class whose objects can be stored using the front-end persistence API. It is important to notice that the class does not contain any notation related to persistence, so it does not require any additional knowledge for the developers to be able to use it.

```
class
  PERSON

create
  make

feature {NONE} -- Initialization

  make (first, last: STRING)
      -- Create a new person.
    require
      first_exists: not first.is_empty
      last_exists: not last.is_empty
    do
      first_name := first
      last_name := last
      age := 0
    ensure
      first_name_set: first_name = first
      last_name_set: last_name = last
      default_age: age = 0
    end

feature -- Basic routines

  celebrate_birthday
      -- Increase age by 1.
    do
      age := age + 1
    ensure
      age_incremented_by_one: age = old age + 1
```

```
    end

feature -- Status report

  first_name: STRING
      -- The person's first name.

  last_name: STRING
      -- The person's last name.

  age: INTEGER
      -- The person's age.

invariant
  age_non_negative: age >= 0
  first_name_exists: not first_name.is_empty
  last_name_exists: not last_name.is_empty
end
```

Listing 6.1: Class *PERSON*, whose objects are to be stored.

The participants were then shown the main class *USABILITY_TEST*, which contained the task description as comments, as shown in Listing 6.2.

```
class
  USABILITY_TEST

create
  make

feature {NONE} -- Initialization

  make
      -- Run application.
    local
      p1, p2, p3, p4, p5: PERSON
    do
      -- Data you need: kind of db: MySQL; host:
         "127.0.0.1"; port 3306;
      -- db name: "tutorial", user: "tutorial", password: "
         tutorial"
      -- Task 1: insert the following 5 persons in the db:
      -- Catelyn Stark, Eddard Stark, Arya Stark, Bran Stark
          and Jon Snow.
      -- Task 2: query the db for all persons and print
```

```
       their first names.
    -- Task 3: celebrate the birthday of Arya and update
       her in the db.
    -- Task 4: delete Eddard from the db.
    -- Task 5: query the db for all Starks and print their
        first names and ages.
  end
end
```

Listing 6.2: Class `USABILITY_TEST`, the experiment starting point.

The tasks required the discovery of the fundamental API abstractions, the combination of different objects and of specific ways to create them (e.g. using the factory pattern). These are items that previous studies found to be critical [118] [53]. More specifically, task 1 involved the following:

1. Creating 5 objects of type `PERSON`.

2. Finding the class `REPOSITORY_FACTORY` and use it to invoke a factory method `create_mysql_repository` that uses the credentials shown in Listing as arguments and polymorphically returns an instance of `RELATIONAL_REPOSITORY`. Alternatively, performing a sequence of 4 interconnected operations to achieve the same result.

3. Finding the class `CRUD_EXECUTOR` and use it to insert the persons into the database. The link between this abstraction and `REPOSITORY_FACTORY` is that a `CRUD_EXECUTOR` requires an object of type `RELATIONAL_REPOSITORY` as an argument to its constructor.

Task 2 involved the following:

1. Finding the generic class `OBJECT_QUERY [G]`, understand that the generic parameter should be of type `PERSON`.

2. Finding that class `CRUD_EXECUTOR` has a routine `execute_query` accepting arguments of type `QUERY [G]`, of which `OBJECT_QUERY [G]` is a descendant.

3. Finding that `OBJECT_QUERY [G]` is also a descendant of class `ITERABLE [G]`, and so it can be used to iterate over the query results, or alternatively that `OBJECT_QUERY [G]` exposes an object of type `RESULT_SET` that can be used in a similar fashion.

Task 3 involved finding a way to update an object of type `PERSON` both in memory and in the database using class `CRUD_EXECUTOR`.

Task 4 involved finding a way to delete an object of type *PERSON* both in memory and in the database using class *CRUD_EXECUTOR*.

Task 5 involved finding a way to express a more complex query involving *criteria*. Criteria are represented in the API by class *CRITERION* and its descendants, located in cluster `criteriontree`. The task involved taking a decision to reuse the existing query object after having reset it for new use with routine `OBJECT_QUERY [G].reset`, or using a new query object. In both cases it was necessary to pass a *CRITERION* object either to routine `set_criterion` or to the constructor. Concrete criteria come as objects of classes *AGENT_CRITERION* and *PREDEFINED_CRITERION*, and can either be created directly or using class *CRITERION_FACTORY*. In both cases the arguments expressing the correct filtering should be passed.

The hardware on which the experiment was conducted was a Macbook Pro laptop equipped with EiffelStudio 7.1.8.8986, with the "full void-safety" compiler option enabled. This detail is relevant because void safety enabled the participants to find out at compile-time if they forgot to create an object, therefore wasting less time to track down calls on void references (equivalent to a *NullPointerException* in Java) at runtime. The Apache Web Server and MySQL version 5.1.44 were installed and running as well.

6.6 Data collection protocol

I ran personally all the sessions and collected all the data. To collect the data three techniques were used:

- The thinking-aloud protocol.
- The screen captured videos (including audio).
- Structured interviews.

The variety of thinking-aloud protocol adopted was already described at the end of Section 6.1.

The sessions were conducted individually in an isolated room and recorded using Camtasia2 [37], version 2.2, a program for screen capturing and sound recording. Capturing the video of the programming activity, together with the sound, was essential to review the data collection process and collect accurate data.

The structured interviews, which happened at the end of each programming session, used a questionnaire with open answers. The questions, which address all the items suggested in Clarke's paper [21] and

listed in Section 6.1, have been adapted from the suggestions by Clarke in a series of articles on the application of cognitive dimensions while conducting usability studies at Microsoft [22, 27, 32, 33, 30, 29, 28, 23, 24, 25, 31, 26]:

- Do you find the API abstraction level appropriate to the tasks?
- Do you feel you had to learn many classes and dependencies to solve the tasks?
- Do you feel you had to keep track of information (not represented by the API) to solve the tasks?
- Does the amount of code required for each task seem about right, too much or too little for you? Why?
- How easy is to evaluate your own progress (intermediate results) while solving the tasks?
- Do you feel you had to choose one way (out of many) to solve a task in the scenario?
- Do you feel you had to understand the underlying implementation to be able to use the API?
- Did you need to adapt the API (inheriting from API classes, overriding default behaviors, providing non-API types) to meet your needs?
- Do you feel you would have to change much in your code to access another kind of persistence store, or write another query? Would you know how to do it?
- Once you performed the first two tasks, was it easier to perform the remaining tasks?
- Does the code required to solve the tasks match your expectations?
- Do you find that the API types map to the domain concepts in the way you expected?

6.7 Data analysis: API usability tokens elicitation

The participant's reactions were categorized in the following way:

- By abstraction, identifying the issues with respect to the four main abstractions of the front-end API:
  - *REPOSITORY*-related issues.
  - *QUERY*-related issues.
  - *CRUD_EXECUTOR*-related issues.
  - *CRITERION*-related issues.
- By API usability tokens (AUT). This is an original categorization created during the analysis of this experiment:
  - *Missed* tokens: missed abstractions or important API features while solving the tasks.
  - *Unexpected* tokens: API uses not foreseen by the API designers.
  - *Surprise* tokens: API aspects that were surprising for the participant.
  - *Choice* tokens: choices the participants had to face before proceeding with a task.
  - *Incorrect* tokens: incorrect API usages.

The tables below describe the extracted tokens in descending order with respect to the number of times in which they were observed.

6.7.1 API usability tokens for class *REPOSITORY*

Table 6.2 shows the results of the experiment with respect to the issues related to class *REPOSITORY*.

Many participants (40%) were expecting a *DATABASE* instead of a *REPOSITORY* abstraction. This can be considered reasonable, given they were asked to work with a relational database. The main justification for choosing the *REPOSITORY* abstraction was to be able to represent mechanisms like serialization, which cannot be classified as databases. Two participants (see token number 7 in Table 6.2), once they have discovered the *REPOSITORY* abstraction, were further expecting a *MYSQL_REPOSITORY* abstraction that was not there. Even after they understood they were supposed to work just with a *RELATIONAL_REPOSITORY*, the design choice remained a bit confusing.

Token number 2 highlights that 40% of the participants were expecting a *REPOSITORY* to be able to execute standard database operations

AUT No.	AUT tag	Repository-related issues	Participants involved	No.
1	Surprise	expecting *DATABASE*, not *REPOSITORY*	P2, P7, P9, P14, P16, P17, P18, P19, P20, P21	10 (40%)
2	Surprise	*REPOSITORY* cannot execute CRUD operations	P2, P5, P8, P9, P11, P15, P17, P20, P22, P25	10 (40%)
3	Surprise	found a connection class but it cannot be used	P1, P6, P7, P8, P9, P10, P16, P17, P25	9 (36%)
4	Missed	repository factory	P9, P10, P18, P24	4 (16%)
5	Surprise	*REPOSITORY* does not have a *CRUD_EXECUTOR*	P8, P9, P17	3 (12%)
6	Unintended	inherit from *EIFFELSTORE_EXPORT*	P5, P13, P14	3 (12%)
7	Surprise	expecting class *MYSQL_REPOSITORY*	P7, P23	2 (8%)

Table 6.2: API usability tokens (AUT) for repository-related issues.

like insert, query, update, and delete directly, instead of having to discover a separate abstraction for that such as the `CRUD_EXECUTOR`. Among them, three noticed that the `CRUD_EXECUTOR` was not mentioned from class `REPOSITORY`, and so it was difficult to find (token number 5) . This is consistent with what observed by Stylos et al. [118] about the issues developers have in discovering classes not mentioned from within the class they are using.

One third of the participants (36%, token number 3) were confused by not having a usable `CONNECTION` abstraction. This is reasonable, because some typical connection data were given, and it was not clear at all that a connection object was implicitly created by the framework and a factory method should have been used instead.

Finding the `REPOSITORY_FACTORY` was a problem only for 3 participants (12%, token number 4), probably due to the fact that in class `REPOSITORY` header comment there was no mention of the fact that objects of descendant classes could be created using the factory. Another likely reason was that the framework did not make it easy to do without a factory (there were four nontrivial steps to perform).

Finally, 3 participants tried to bypass some visibility issues by inheriting form `EIFFELSTORE_EXPORT`, therefore using the API in an unintended way (token number 6). Such attempts are very useful for the API designers during the API testing phases, because they suggest refactorings to restrict the user choices to the intended ones.

6.7.2 API usability tokens for class `QUERY`

Table 6.3 shows the results of the experiment with respect to the issues related to class `QUERY`.

Token number 1 refers to the choice developers had to make in assuming (or not assuming) a read before an update or delete. In the tasks it was not needed, because the objects to update and delete were already in

memory, but in general the assumption that a query is needed is correct, and a common safe practice with databases. This is therefore not really an issue.

Token number 2, missed by 7 participants (28%), highlights the fact that the argumentless constructor of class `QUERY [G]` does not explicitly state in its comment the intended purpose for the constructed object. The intended semantic—query all objects of the class generic type G—was difficult to guess without looking at the implementation. To add to the confusion, there is another constructor accepting a `CRITERION` argument that cannot be used to query for all objects, because class `EMPTY_CRITERION` has a restricted visibility for clients.

Among the tokens numbered 3 to 6, the most interesting one is token number 4: the API incorrect usage referring to the fact that a query needs to be made explicitly reusable by invoking feature `reset`. This is a good example of what a usability test may find, and it is also something suggesting that reusable queries would increase the API usability.

The remaining tokens were surprises: token number 3 is a consequence of the fact that participants were expecting to find the result of a query in the return type of method `CRUD_EXECUTOR.execute_query`.

Token number 5 is related to class `ITERATION_CURSOR` in the EiffelBase library. This abstraction is usable also to iterate over streams, for which it does not make sense to have a command to set the cursor at the start of the stream.

Token number 6 uncovers a subtle naming issue: in class `QUERY` there is an attribute `result_cursor` of type `RESULT_SET` that looks like it is both the query result (by looking at its type) and an iteration cursor over the query result (by looking at its name and feature comment). As it is truly an iteration cursor, the type is a bit misleading, but this name was chosen to make developers familiar with databases feel more at home. The issue itself is easy to fix by making `RESULT_SET` inherit from `ITERABLE`, so that the `across` syntax can be used.

Token number 7 is just one of the choices the API offers: use the object queries or queries returning tuples of values, which sometimes can be more efficient or appropriate.

Token number 8 indicates that three participants expected that a query had a feature `execute` to execute itself, instead of `CRUD_EXECUTOR.execute_query`. This was clearly a design choice. Given the fact that the majority of participants did not have issues with this, it can be ignored.

Three people missed an interface to execute SQL queries directly (token number 9). The answer to this is that the API was purposely designed not to offer such an interface, to shield object-oriented developers from

AUT No.	AUT tag	Query-related issues	Participants involved	No.
1	Choice	do objects need to be read again before an update?	P2, P3, P8, P10, P18, P19, P20, P21	8 (32%)
2	Missed	default query is for all objects	P9, P10, P11, P12, P17, P18, P25	7 (28%)
3	Surprise	*QUERY* contains the result of the query itself	P2, P3, P9, P21, P23, P24	6 (24%)
4	API incorrect usage	tried to reuse the same query without resetting it	P1, P6, P12, P13, P18, P24	6 (24%)
5	Surprise	expecting command to reset cursor for iteration	P13, P14, P16, P17, P23	5 (20%)
6	Surprise	across syntax did not work with result set	P3, P4, P8, P11, P23	5 (20%)
7	Choice	object query or tuple query?	P3, P6, P10, P25	4 (16%)
8	Surprise	a query object cannot execute itself	P7, P10, P22	3 (12%)
9	Surprise	expecting a SQL interface	P5, P12, P13	3 (12%)
10	Surprise	*QUERY [STRING]* was not appropriate	P6, P23	2 (8%)
11	Surprise	*QUERY* does not allow insert, updates or deletes	P9, P10	2 (8%)
12	Surprise	feature *new_cursor* suggests a new object, but it is not	P13, P24	2 (8%)
13	Surprise	the side effect on the query argument of *execute_query*	P13	1 (4%)

Table 6.3: API usability tokens (AUT) for query-related issues.

the need to deal (and know) SQL, and to provide a more secure interface (for example not allowing SQL injection attacks by construction). A partial trade-off when one does not need to retrieve whole objects but just value tuple of values is offered by class *TUPLE_QUERY*.

Token number 10, though occurred to only 2 participants, uncovers a potential issue with understanding the correct use of genericity: choosing the right type for the generic parameter. Given that in the experiment this issue was quite quickly understood and fixed by the two participants, it probably does not deserve too much attention.

Token number 11 is again about design choices (assigning responsibilities to classes). If one accepts the *CRUD_EXECUTOR* abstraction, then it makes sense to have the operations on the database there. Even if the executor abstraction is eliminated, it is the designers' opinion that these operations would fit best in the *REPOSITORY*.

Token number 12 highlights that feature *QUERY*.*new_cursor* suggests that a new object is created, while in fact it is not. As the interface of class *ITERABLE*, the class from which *QUERY* is inheriting from, demands to provide a fresh cursor, this is a problem of the implementation that does not conform to the specification.

Token number 13 shows that *CRUD_EXECUTOR*.*execute_query* accepts a query as an argument, to then make side effect on the argument itself, which can considered a slightly unusual programming style.

6.7.3 API usability tokens for class *CRUD_EXECUTOR*

Table 6.4 shows the results of the experiment with respect to the issues related to class *CRUD_EXECUTOR*.

Token number 1 questions the choice of *CRUD_EXECUTOR* as an abstraction name. First, "CRUD" is simply a mysterious acronym to many people that are not seasoned database practitioners. Second, it suggests a link to relational databases which it is not a good idea for an API that aims at providing uniform access to different kinds of persistence stores. The challenge is of course which name to choose instead. Some proposals, given by the participants themselves, were *QUERY_EXECUTOR*, just *EXECUTOR* and even *ENTRY_POINT*, hinting at the fact that *CRUD_EXECUTOR* is the ideal entry point for accessing the whole library functionality.

Token number 2 points out the fact that 7 participants (28%) were expecting *execute_query* to be a function and return the query result. This kind of pattern is not common in the Eiffel development practice, where it is preferred to keep commands (modifying the state of an object, in this case the *QUERY* passed as an argument) separate from functions that return a value and are not meant to change the state of objects. The importance of discussing this issue lies in the fact that it provides an answer to the third research question mentioned in Section 6.3. This appears to be an example in which a design choice like the command-query separation principle [89] comes with a possible trade-off in terms of usability.

Tokens number 3 to 6 were only mentioned by one participants each. While two of them are not particularly relevant, tokens number 4 and 5 are worth more discussion. The surprise about the presence of an explicit update feature (token number 4) was mentioned by a participant having experience with object-relational mappers. The most sophisticated ones keep the objects in memory and in the database automatically synchronized. The framework reaches a different trade-off: simplify the implementation and giving a bit more control to developers at the cost of limiting the number of tasks executed automatically. Token number 5 is a simple naming issue: the claim is that given that there exists feature *execute_query*, it would have been more consistent to have features named *execute_update*, *execute_insert*, and *execute_delete* instead of just *update*, *insert*, and *delete*.

6.7.4 *API usability tokens for class CRITERION*

Table 6.5 shows the raw results of the experiment with respect to the issues related to class *CRITERION*.

The main highlight here is the fact that 14 participants (56%) missed the *CRITERION_FACTORY* completely. Given that the criteria could be easily created also without the factory, this is a clear signal of the fact that this class is a candidate for being eliminated, and its functionality distributed

AUT No.	AUT tag	Crud_executor-related issues	Participants involved	No.
1	Surprise	what does CRUD mean?	P6, P9, P10, P12, P14, P15, P16, P17, P18, P19, P23	11 (44%)
2	Surprise	expecting a result type in feature *execute_query*	P7, P14, P15, P19, P21, P23, P24	7 (28%)
3	Missed	update feature in *CRUD_EXECUTOR*	P11	1 (4%)
4	Surprise	An explicit update is needed	P5	1 (4%)
5	Surprise	why *execute_query* and not *execute_update*?	P13	1 (4%)
6	Choice	used explicit transactions	P22	1 (4%)

Table 6.4: API usability tokens (AUT) for crud_executor-related issues.

in the other classes in the criterion tree cluster. Incidentally, this result confirms the previous literature about the issues in finding and using factories mentioned in Section 6.1 and provides an answer to the second research question mentioned in Section 6.3.

After tokens number 2 and 3, just showing that the participants chose both the kinds of criteria available, token number 4 signals the difficulty faced by 9 participants (36%) in understanding what the string arguments for the `PREDEFINED_CRITERION`'s constructor should be (see Listing 6.3). Related to this is token number 7: one participant did not understand that the string argument `attr` of feature `make` was referring to an attribute name.

```
class
  PREDEFINED_CRITERION

inherit
  CRITERION

create
  make

feature {NONE} -- Initialization

  make (attr, op: STRING; val: ANY)
      -- Initialization for 'Current'
    require
      correct_operator_and_value: is_valid_combination (op,
          val)
    do
      ...
    end
...
```

AUT No.	AUT tag	Criterion-related issues	Participants involved	No.
1	Missed	criterion factory	P2, P6, P8, P10, P11, P13, P16, P17, P18, P19, P20, P22, P23, P24	14 (56%)
2	Choice	predefined criterion	P3, P8, P9, P10, P15, P17, P18, P19, P20, P22, P23, P24, P25	13 (52%)
3	Choice	agent criterion	P1, P2, P5, P6, P11, P12, P13, P14, P16	9 (36%)
4	Choice	which strings are operators in predefined criteria?	P3, P8, P9, P10, P15, P18, P20, P24, P25	9 (36%)
5	Choice	no criterion: queried all objects and then selected	P4, P7	2 (8%)
6	Choice	[[ ]] syntax with any of the 2 criteria above	P21	1 (4%)
7	Missed	Arg. 1 in predefined criterion's *make* is attr. name	P24	1 (4%)

Table 6.5: API usability tokens (AUT) for criterion-related issues.

```
end
```

Listing 6.3: Constructor of class *PREDEFINED_CRITERION*.

Here there are two issues. One is easily fixed with a better comment providing also a usage example. The other is to question the usage of strings when more specific information is needed. While for the first argument (attribute names) choosing a string is appropriate, the second argument (the operator) could be a encapsulated in a type. This would mean moving the checks about the operators validity from runtime (see the precondition) to compile-time, a sure improvement.

Token number 5 refers to the fact that 2 participants chose not to go through criteria but just query for all the objects and then selecting them explicitly, a reasonable choice that should be possible in this context.

Finally, token number 6 refers to a specific syntax allowing to mix the two kinds of criteria (predefined and agent) in a uniform style, and possibly combine them with logical operators. Feedback from the videos confirms that the feature was discovered, but judged to be quite cryptic, especially because there was no usage example, and it is probably not by chance that only one participants dared to use it.

6.8 Data Analysis: final questionnaire

The questions were open, therefore in the analysis the answers were fitted into three categories (yes, no, and "sometimes"). Table 6.6 collects the questions of the questionnaire, possibly shortened for compactness of presentation, and the corresponding answers. In analyzing the answers we will refer to a series of articles by Clarke on the application of cognitive dimensions while conducting usability studies at Microsoft [22, 27, 32, 33, 30, 29, 28, 23, 24, 25, 31, 26].

Most of the participants (80%) answered to question 1 that the abstraction level proposed by the API was appropriate for the tasks. Some felt that

it was too high-level, because it did not provide a SQL interface. This was a conscious design choice: provide an object-oriented API that works with domain objects. If one wishes to write directly SQL against, say, a MySQL database, other libraries are available, for example EiffelStore [114]. In general, the API components can be described as "aggregate", because the all the tasks could be accomplished by using the same set of components [22].

In answer to question 2, 72% of the participants though they did not have to learn many classes to be able to use the API effectively. Three of the participants thinking they had to learn too many classes had missed the *REPOSITORY_FACTORY* entirely, which probably explains their feedback. It can be concluded that the API supports an incremental and minimal learning style, because each scenario required a small number of classes and dependencies to be realized successfully [27].

The answers to question 3 confirm that the API has been designed to be "local", that is, all the information the user needs to maintain is represented in the API itself [32]. As a counter-example, consider an API providing support for writing two different file formats (binary and XML). If users have to know which format they need in order to understand which part of the API to use, then this API is considered "global", defeating the front-end API's designers intentions.

Another result is that most of the participants found the amount of code required for each task reasonable (question 4). This was certainly one of the design goals of the front-end API. In terms of cognitive dimensions, one could say that the work step unit is "local incremental", because the code required to accomplish the tasks is contained in one local code block, and can be written incrementally [33].

The answers to question 5 highlight that most of the participants (68%) found it easy to evaluate their progress while doing the tasks. This number could have possibly been higher if there were a facility to clean up the database from intermediate tests, something that is outside the scope of the API. In terms of cognitive dimensions, the API supports "progressive evaluation at the functional chunk level" [30], because users can only stop and evaluate their progress on a standard database operation like insert, update or delete only after having performed an additional task, that is querying the database.

The answers to question 6 highlight the fact that the API was perceived as providing choices. The choice between the two criteria is probably the most interesting. Choosing an agent criterion instead of a predefined criterion leads to significative differences in the implementation, which means that while in general the API exposes a minor and reversible level of premature commitment, in the case of the criteria the API exposes an irre-

versible level of premature commitment [29]. This means that most of the API choices can be easily reverted to try the alternatives, while in the case of criteria there is a stronger commitment in terms of written code when choosing a predefined criterion with respect to an agent criterion, and to replace one with the other requires more work than for the other choices.

The answers to question 7 highlight that 48% of the participants felt the need to inspect the implementation to use the API. This is clearly going against the inspiring principles of the front-end API. The most important instances noted by the analysis are the meaning of the arguments to the *PREDEFINED_CRITERION*'s constructor, the meaning of the default constructor for *QUERY*, and the fact that every operation against the database happens within an implicit transaction. With respect to the cognitive dimensions, the API appears to be in the middle between a "context driven view" and an "expansive view" of its details [28]. This means that even if the API exposes enough information to allow users to understand the context they are working with, sometimes users have to inspect the code to get a more precise understanding. Here the goal of minimal penetrability—reached when users only need to know about different methods and classes—has not been fully achieved.

The answers to question 8 reflect the fact that only two participants used an API class through inheritance. Using inheritance was in this case neither necessary nor an issue, because Eiffel supports multiple inheritance, in contrast to languages like Java where one has to be very careful in choosing which class goes to occupy the only inheritance slot available. For the rest, no particular adaptation was needed, which can be considered positive with respect to usability for the kind of users the front-end API targets. In terms of cognitive dimensions, the front-end API does not support "elaboration" and can mostly be used as is [23].

The answers to question 9 clearly point out at the fact that the API has a "low viscosity" [24]. This means that users perceive that they can easily make changes to code written against the API. On the designer's side, while it is certainly true that working with another kind of relational database should not present any surprises, the question of how the API will adapt to non-relational databases is yet to be explored.

The answers to question 10 reflect the observation that after an initial relatively hard learning-by-discovery phase, the API lends itself to quick and effective use, the filtering criteria being probably the only serious obstacle. With respect to the cognitive dimensions, one can say that the API "exposes core consistency" [25], because it mostly uses the same idioms for similar goals, with just the partial exception constituted by the criteria.

Most of the participants had comments with respect to the code they

were expecting to see (question 11), and this largely depended on their previous experience. Most of the remarks went in a direction of asking for simplification of the API, and have already been mentioned in the analysis to the previous questions. One refers to the choice of having, in *CRUD_EXECUTOR*, a method *execute_query* and then having methods *update* and *delete*, when *execute_update* and *execute_delete* would have been more consistent. This is clearly a usability issue that needs to be fixed. In terms of cognitive dimensions, the API can be assessed as "plausible" [31], because the code required to accomplish the goals can be interpreted correctly but does not fully match users' expectations. One more observation connects the library design with the evolution concerns that are an important element of this thesis: the analysis of the videos and the questionnaire showed that some participants wondered about how it was possible that they did not have to worry about mapping the objects to the relational tables. Most of them saw, while having a look at the data they wrote in the database, that there was a meta-mapping done for them by the library as described in Section B.3. They therefore understood that with such a meta-mapping evolving the system by changing the attributes of a class becomes really easy for developers, because in the standard scenario they don't have to do anything at all to migrate the database schema. Of course such a meta-schema is a trade-off and has an impact on the overall efficiency in some scenarios, because more joins are necessary on average.

With respect to the answers to question 12, we have already discussed the possibility of replacing *REPOSITORY* with *DATABASE*, and of course *DATABASE* with something else, and to rename the unfortunately named *CRUD_EXECUTOR*. Another suggestion is to use class *CRITERION*, which could be called *FILTER*, but here which name to prefer is less clear. In terms of cognitive dimensions, the API has a "plausible" domain correspondence [26], because some of the exposed types only map to the types expected after describing the mapping.

We have already mentioned in Section 6.4 that the different level of experience among the participants appears to be an important factor of variability. Therefore we controlled for it by dividing the participant pool in two groups according to their experience and analyzing the results again. The first group contains the 11 participants having a declared experience with object-oriented applications greater than the median (six years). The second group contains all the other participants. The data are synthesized in Table 6.7 and Table 6.8 respectively. While most of the answers confirm the general trend, it is worth commenting more in detail the answers that present the most remarkable differences.

The answers to question 2 highlight that all the experienced developers

No.	Question	Yes	No	Sometimes
1	Do you find the API abstraction level appropriate to the tasks?	20 (80%)	5 (20%)	0
2	Do you feel you had to learn many classes and dependencies to solve the tasks?	7 (28%)	18 (72%)	0
3	Do you feel you kept track of information external to the API to solve the tasks?	4 (16%)	21 (84%)	0
4	Does the amount of code required for each task seem about right?	22 (88%)	3 (12%)	0
5	Was it easy is to evaluate your own progress while solving the tasks?	17 (68%)	8 (32%)	0
6	Do you feel you had to choose one way (out of many) to solve a task in the scenario?	0	3 (12%)	22 (88%)
7	Do you feel you had to understand the underlying implementation to use the API?	12 (48%)	13 (52%)	0
8	Did you need to adapt the API to meet your needs?	2 (8%)	23(92%)	0
9	Would you have to change much code to access another database or write another query?	4 (16%)	21 (84%)	0
10	Once you performed the first two tasks, was it easier to perform the remaining tasks?	23 (92%)	2 (8%)	0
11	Does the code required to solve the tasks match your expectations?	4 (16%)	1 (4%)	20 (80%)
12	Do you find that the API types map to the domain concepts in the way you expected?	4 (16%)	0	21 (84%)

Table 6.6: Answers to the final questionnaire.

think that they did not had to learn many classes and dependencies to solve the tasks, while for 50% of the less experienced developers this was not the case. In fact, some of them are the ones who missed the repository factory.

Unsurprisingly, the answers to question 5 confirmed that for experienced developers it was easier to evaluate their own progress while solving the task.

From the answers to question 6 we learn that all the experienced developers had the feeling of having choices when programming against the API, while three of the developers in the other group (21%) thought that the API did not offer choices. This might be a step in the right direction for the library design, because the front-end API targets both categories of users and it is good for less experienced developers to “guess right” without being fully aware of the available choices.

The answers to question 7 almost reversed (between the two groups) the percentages of developers that had to understand the implementation to use the API. Most of the more experienced developers thought they did not have to, while the opposite was true for the other group.

No.	Question	Yes	No	Sometimes
1	Do you find the API abstraction level appropriate to the tasks?	8 (73%)	3 (27%)	0
2	Do you feel you had to learn many classes and dependencies to solve the tasks?	0	11 (100%)	0
3	Do you feel you kept track of information external to the API to solve the tasks?	1 (9%)	10 (91%)	0
4	Does the amount of code required for each task seem about right?	10 (91%)	1 (9%)	0
5	Was it easy is to evaluate your own progress while solving the tasks?	9 (82%)	2 (18%)	0
6	Do you feel you had to choose one way (out of many) to solve a task in the scenario?	0	0	11 (100%)
7	Do you feel you had to understand the underlying implementation to use the API?	4 (36%)	7 (64%)	0
8	Did you need to adapt the API to meet your needs?	2 (18%)	9 (82%)	0
9	Would you have to change much code to access another database or write another query?	2 (18%)	9 (82%)	0
10	Once you performed the first two tasks, was it easier to perform the remaining tasks?	11 (100%)	0	0
11	Does the code required to solve the tasks match your expectations?	1 (9%)	0	10 (91%)
12	Do you find that the API types map to the domain concepts in the way you expected?	9 (1%)	0	10 (91%)

Table 6.7: Answers to the questionnaire: experienced group.

No.	Question	Yes	No	Sometimes
1	Do you find the API abstraction level appropriate to the tasks?	12 (86%)	2 (14%)	0
2	Do you feel you had to learn many classes and dependencies to solve the tasks?	7 (50%)	7 (50%)	0
3	Do you feel you kept track of information external to the API to solve the tasks?	3 (21%)	11 (79%)	0
4	Does the amount of code required for each task seem about right?	12 (86%)	2 (14%)	0
5	Was it easy is to evaluate your own progress while solving the tasks?	8 (57%)	6 (43%)	0
6	Do you feel you had to choose one way (out of many) to solve a task in the scenario?	0	3 (21%)	11 (79%)
7	Do you feel you had to understand the underlying implementation to use the API?	8 (57%)	6 (43%)	0
8	Did you need to adapt the API to meet your needs?	0	14(100%)	0
9	Would you have to change much code to access another database or write another query?	2 (14%)	12 (86%)	0
10	Once you performed the first two tasks, was it easier to perform the remaining tasks?	12 (86%)	2 (14%)	0
11	Does the code required to solve the tasks match your expectations?	3 (21%)	1 (8%)	10 (71%)
12	Do you find that the API types map to the domain concepts in the way you expected?	3 (21%)	0	11 (79%)

Table 6.8: Answers to the questionnaire: group with less O-O experience.

6.9 Lessons learned

The main results can be now expressed as answers to the research questions mentioned in Section 6.3, and lessons can be learned from the experience coming from the experiment:

- Which keywords best describe a functionality provided by the API? The evaluation showed at least one clear example of poor abstraction choice: the *CRUD_EXECUTOR*. In addition, there were some other names of classes and features in need of being re-assessed, like *REPOSITORY* versus *DATABASE*, *CONNECTION*, and `update` versus `execute_update`. The problem of *CONNECTION* in particular is that it is not supposed to be used by clients, so its visibility should probably be further restricted and an appropriate comment should be put in the factory methods of class *REPOSITORY_FACTORY*. The lesson learned is to pick consistent names that do not assume too much knowledge on the developer's side, and that are as common as possible without being vague.

- How do developers assess the API with respect to discovering relationships between API types? The most relevant issue discovered was that *CRUD_EXECUTOR* is not accessible from the *REPOSITORY* class. This was particularly severe because *CRUD_EXECUTOR* is de facto the entry point for the API. This result is consistent with the observations from previous authors [118], stating that discovering relationships between API types is difficult. Another critical point was discovering the *REPOSITORY_FACTORY*. The remaining relationships between the API types were relatively straightforward to figure out in comparison. The lesson learned is to make the necessary types accessible from the type from which developers are supposed to start. The proposed solution involves improving the documentation of class *REPOSITORY* to specify that its objects are to be created using *REPOSITORY_FACTORY*, and adding in class *REPOSITORY* a utility method returning an instance of *CRUD_EXECUTOR*.

- How do developers assess the API with respect to object creation? Do they find it problematic when public constructors are not available, but factories are provided instead? One of the two factories, *CRITERION_FACTORY*, was mostly ignored, but creating objects of type *CRITERION* remained possible and viable without it. The other factory, *REPOSITORY_FACTORY*, was more useful but unfortunately not well documented in class *REPOSITORY*. Creating and initializing

correctly *REPOSITORY* objects without using *REPOSITORY_FACTORY* was longer and more demanding, because it required four non-trivial operations. While the first case seems to confirm the issues that people encounter in discovering factories already mentioned by previous authors [53], the second case suggests a way to improve the situation when a factory is needed: document the factory existence in the constructor comments and class header comments of the types for which the factory is needed (in our case *REPOSITORY* and its descendants).

- Do developers prefer argumentless constructors to constructors with arguments? Contrary to what Stylos et al. observed [117], in the case of class *QUERY*, no participant had issues with the constructor accepting a criterion as an argument, while 7 participants (28%) had issues with the argument-less constructor because it was not clear that it would create a query for all objects of the query generic type.

- Do developers manage to always determine the semantics of the outcome of a method call? In programming languages like Java methods returning void can sometimes be confusing because it is not clear to developers where to look for the expected outcome of the method. For example, a method could validate a string in a way that returns void if the validation went well, or that throws an exception if the validation did not go well. According to Duala-Ekoko and Robillard [52], this design confused developers. In the front-end API, a similar issue occurs with feature *execute_query*, where its documentation comment states that the result of the query is in an attribute of class *QUERY* of type *RESULT_SET*. While on the one hand participants did not find any issue with interpreting the intended semantics, on the other hand more than one participant claimed to have “guessed a lot”, and that it worked. This can be interpreted as the API design presenting somewhat expected behaviors even when the documentation was not read or not clear.

6.10 Threats to validity

A possible threat to internal validity is that the choice of participants was far from random. E-mails were sent to persons that were potentially interested in the subject of the experiment, like former Introduction to Programming course students and former members of the Chair of Software Engineering in Zurich. Positive answers came from roughly 40% of them.

Most of the people were known to the person conducting the experiment. This element, which may be seen as introducing some bias, might actually even have helped during the experiment to make the participants feel at ease and reduce the stress that would come from thinking about performing badly in front of an unknown person. Another factor mitigating possible selection bias is that the pool of participants was quite heterogeneous with respect to their programming experience.

Absence of Internet access might have influenced the outcome of the study as well. The reason for such a choice was to reduce the informational noise and let developers focus on the API itself—class names, class relationships and API interfaces—while solving the tasks. Besides, how much developers may benefit from looking for solutions on the web is controversial [52].

Some of the questions on the questionnaire may be seen as rather subjective. This is certainly a thread to validity, partially mitigated by the fact that 9 questions (75%) received answers chosen by at least 80% of the participants.

Finally, external validity is not too much of concern, because the purpose of the study was to validate one single library, not to generalize the findings to other libraries.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the comprehensive solution this thesis provides for the evolution of object-oriented systems in need of persisting their objects. Section 7.2 summarizes the contributions of this thesis by contrasting them with previous work. Section 7.3 describes future work.

7.1 Tackling the limits of existing approaches

The software development community provides practical but limited solutions to the problem of evolving persistent classes. Such solutions imply that software developers write transformation functions to adapt old objects to the new classes. Three factors constrain such solutions: a class schema approach, an invariant-unaware retrieval process, and lack of tool support. The following subsections highlight how this thesis tackles the aforementioned constraints.

7.1.1 Multi-version model

The most widely adopted programming languages promote the "class schema approach" described in Section 2.1. Given an object to retrieve, the approach compares the object's class schema available in the retrieving system with the class schema of the stored object. If a mismatch occurs, developers need to code a transformation function to adapt the retrieved object to the current, newly created one. This approach does not fit well long-term evolution scenarios with many possible class schemas, and can potentially complicate the transformation function, because the func-

tion needs to react appropriately in presence of multiple retrieving class schemas.

We address the above issue by acknowledging that there can be many versions of a certain class, and by providing the infrastructure to deal with it from the beginning, that is from the very moment in which a new version is released. Therefore we integrate support into an Integrated Development Environment, elevating class schema evolution to the status of first-class citizen of the software development realm rather than undesirable side effect of the software production activities. The framework can handle a transformation function for each pair of versions, keeping all the transformation functions in a specific handler class.

7.1.2 *Invariant-safe evolution*

Object retrieval can be regarded as "an extralinguistic method for creating objects" (that is, without using a constructor [14]). This implies that once restored, and before being used, the object needs to satisfy its class invariant, whether implicitly or explicitly stated. The notion of class invariant in general, and its use in connection to object persistence in particular, are not widespread in the current software development community. To make things more critical there are many lenient retrieval algorithms willing to accept potentially inconsistent objects into the retrieving system.

We address the above issue by the framework retrieval algorithm. The persistence library presented in the thesis performs at retrieval time a series of checks to make sure that appropriate transformation functions exist and finally that the retrieving class invariant holds. This leads to an invariant-safe class schema evolution.

7.1.3 *Release-time evolution handling*

Current software systems rely completely upon developers to provide support for class schema evolution. In our experience developers write transformation functions between two versions when they need them, that is, only when they get a retrieval error. In the scenario in which they are using lenient retrieval algorithms that accept inconsistent object into the system, developers may not even immediately realize that the retrieved data are semantically wrong.

The solution we propose to mitigate the issue is to handle class schema evolution as early as possible, that is at system release time. A system release triggers support for the creation of schema evolution handlers, intended to help developers to better focus on possible issues that may arise

from previous versions of the newly released code, and support for filtering the attributes to be stored.

7.2 Conclusions

This thesis proposes a comprehensive approach to the evolution of object-oriented applications that persist their objects and need to be able to retrieve them in the future. The semantically interconnected parts that work together for the purpose of mitigating the issues that persistent evolving software typically presents are the following:

- A formal model for changes in a class schema, resulting from object-oriented theory and experience.
- A schema evolution tool integrated into an IDE and implementing the formal model.
- A persistence library integrating the proposed schema evolution approach and featuring seamless access to different kinds of persistence stores.
- A measure of robustness for the evolution of persistent applications, devised to understand to what extent they are able to successfully retrieve previously stored objects.

The analysis of existing and realistic code bases has shown that the theoretical model proposed is viable, and that it is also possible to assess how robust a software system is with respect to its capacity to retrieve previously stored objects. This is intended to facilitate acceptance of the proposed framework in realistic software projects.

The front-end persistence API, supporting the model by providing an invariant-safe retrieval algorithm and seamless access to different kinds of persistence stores, has been validated by an experimental study. The study, performed with a group of software developers with diverse backgrounds, has shown that the API is usable in practice in commonplace scenarios even without consulting external documentation.

This work has potential to increase awareness of the issues connected to object-oriented class schema evolution and shows one reasonable way to deal with it.

The tool and the libraries developed as part of the thesis are available as open source software and integrated into EVE, the research branch of the EiffelStudio IDE [57].

7.3 Future work

We now propose some future work following the contributions highlighted in the previous section.

The formal model can benefit from a broader empirical validation: more studies can be performed on more code repositories, focusing in particular on object-oriented programming languages other than Java and Eiffel.

In the study about invariant evolution in Section 4.4.1 we notice that most of the invariant clauses analyzed are relatively simple. This suggests that it may be reasonable to investigate the use of tools to automatically detect invariant strengthening and therefore warn about possible retrieval errors before they actually happen.

It also seems interesting to consider, in the model of software evolution, the evolution of the persistence software mechanisms themselves. A first step is to investigate if the issue is relevant in existing code bases.

The schema evolution tool can be improved by implementing an IDE-integrated recommendation system aiming at facilitating the discovery of the API methods and types not directly reachable from the type developers are currently working with, similarly to what proposed by Duala-Ekoko and Robillard [51], and re-evaluate the API usability to see if any difference in usability is detected.

Adapting the tool to existing versioning systems, though out of scope for this thesis, would certainly help achieving broader validation and acceptance, and therefore needs to be considered for future work. At the moment the tool is generating its own version numbers, and has its own semantics for a system release. In a realistic scenario, the tool might rely on some software configuration management tool (e.g. CVS, SVN, Mercurial, or GIT) to get version identifiers. This would bring advantages and disadvantages: on the positive side the configuration management tool will generate the identifiers for us, while on the negative side every small change in and outside the class (possibly not related to persistence) will trigger a new class version identifier. The situation would be improved by assuming that there is some release management mechanism in place (a typical example is tags following the semantic versioning [107]).

The mechanism to detect attribute renames can be improved by tracking developers' direct use of a renaming IDE functionality, and by performing some code analysis of the class routines using the attribute candidate to a rename. This helps to detect attribute renames with more confidence.

We are improving the front-end API by incorporating the suggestions

resulting from the performed study. It can be interesting to perform another experimental study to see if with a different pool of participants the results can be replicated.

The back-end API can be extended to support more kinds of relational databases and NoSQL data stores as well (a CoachDB extension is under development at the time of writing).

APPENDIX A

EIFFEL AND DESIGN BY CONTRACT

While applicability to other programming languages is possible, the examples, the tools and the libraries implemented as part of this thesis are all based on the Eiffel language [89]. This chapter provides a brief overview of the basics of Eiffel and the Design by Contract software development method, essential to the results presented in the thesis.

A.0.1 Types

Eiffel is a purely object-oriented (O-O) language. It uses static typing and dynamic binding and supports multiple inheritance and genericity. The type hierarchy has a common root: class *ANY* from which all other classes inherit by default.

Eiffel supports two kinds of types: *reference types* and *expanded types*. An entity declared of a reference type *C* represents a reference that may become attached to an instance of type *C*, while an entity declared of an expanded type *C* directly denotes an instance of *C*. A special case of expanded types are the *basic types* (also called "primitive" in other languages), such as *INTEGER*, *REAL*, *CHARACTER*, and *BOOLEAN*. The instances of these types are also objects, but they are implemented through special compiler support for efficiency reasons. Class *NONE*, which exists only in theory, inherits from all reference types, cannot be inherited from and has only one instance: the special value **`Void`**, denoting an unattached reference, the equivalent of `null` in other programming languages.

Eiffel does not support routine overloading, so all routines in a class must have different names.

A.0.2 Information hiding

Eiffel does not use keywords to express the export status of features; rather, it allows the specification of a list of classes to which features are available. For instance, features exported to classes *A* and *B* will be callable from *A*, *B* and their descendants. Hence, features exported to *ANY* are available to all classes (the equivalent of the `public` access modifier in Java/C#), and features exported to *NONE* are not callable from outside their class (the equivalent of the `private` access modifier in Java/C#). When no list of classes is specified, the compiler assumes that all the features are public, that is, exported to *ANY*. This is the default.

A.0.3 Code organization

Eiffel does not support namespaces. Classes can be organized in *clusters*, which are simple folders that have no scoping effect. Two clusters in a software system should hence not contain two classes with the same name, or one of the classes must be renamed or excluded from the system. Within a class, the language allows a *feature* clause to help organize features by semantic categories and to help expressing information hiding.

A.0.4 Terminology

An Eiffel class has a set of *features* (operations), which can be either *routines* or *attributes*. In Java/C# terminology, features are called "members" and routines are called "methods". Eiffel also makes a distinction between *functions* (routines returning a result) and *procedures* (routines that do not return a result). Objects are created through calls to *creation procedures*, known in Java/C# as "constructors". Creation procedures in Eiffel do not have to conform to any naming scheme; they are normal procedures, which acquire the special status of creation procedures by being declared in the **`create`** clause of a class.

Eiffel also distinguishes between *commands* — features that do not return a value (procedures), and *queries* — features that do return a value (attributes and functions).

Eiffel uses the notions of *supplier* to denote a routine or class providing a certain functionality, and *client* for a routine or class using that functionality. Client-supplier and inheritance are the two fundamental relationships between classes.

Eiffel routines and classes can be **`deferred`** (or "abstract" in Java/C# terms). A class having one or several deferred routines must be deferred itself and cannot be instantiated. A class can be declared deferred even if

it does not contain any deferred routines, just for the purpose of making it impossible to instantiate it.

A.0.5 Design by Contract

Eiffel supports the Design by Contract software development method [90], through which classes can embed specification checkable at runtime. *Contracts* are the mechanism allowing this: *routine preconditions* specify with the keyword **`require`** boolean conditions that must be fulfilled by any client upon calling the routine; *routine postconditions* specify with the keyword **`ensure`** boolean conditions that must be fulfilled when the routine is done executing. Preconditions are thus an obligation for the client, who has to fulfill them, and a benefit for the supplier, who can count on their fulfillment; conversely, postconditions are an obligation for the supplier and a benefit for the client. Postconditions can contain the **`old`** keyword. This keyword can be applied to any expression and denotes the value of the expression on routine entry. The most important kind of contract for persistence is the *class invariant*, specified with the keyword **`invariant`**. Class invariants are boolean conditions that must hold whenever an instance of the class is in a visible state, that is, after the execution of a creation procedure and before and after the execution of exported routines. A class is responsible for satisfying its own invariant.

Routine pre- and postconditions and class invariants should be written already in the design phase of the software system and are part of the interface of the class. Eiffel also supports other kinds of assertions:

- Loop invariants: conditions that must hold before and after each execution of the loop body.
- Loop variants: integer expressions that must always be non-negative and must be decreased by each execution of the loop body.
- **`check`** assertions: conditions that can appear inside routine bodies, expressing properties that must hold when the execution reaches that point; these are similar to the `assert` statements in languages with a C-derived syntax.

Calling functions from assertions is allowed and greatly increases the expressiveness of Eiffel contracts, but it also introduces the possibility of side effects in contract evaluations.

Any Eiffel assertion can be preceded by a tag, followed by a colon, as in `balance_positive: balance > 0`, where `balance` is an integer variable in scope.

Runtime contract checking can be enabled or disabled. It is enabled typically during the development phases of a software system and (partially) disabled in production mode, due to its performance penalty. Contract violations are signaled at runtime through exceptions, and signal the presence of bugs.

The support for Design by Contract in Eiffel is essential to the use of executable specification by programmers in this language. This is made clear by a study [19] which shows that Eiffel classes contain more assertions than classes written in programming languages that don't support Design by Contract. In the classes examined in the study, 97% of assertions were located in contracts rather than in inline assertions.

Eiffel was the first language to support Design by Contract in this form. Since then, many other languages natively contain or have introduced features supporting executable contracts in the form of pre- and postconditions and class invariants; among them, some of the best known are JML [85], Spec# [11], C# [58] and D [88].

A.0.6 Void Safety

A program is void safe if it has no void dereferencing errors. A void (or null) dereferencing error happens when the operation `x.f()` fails because the target object `x` denotes a void reference at execution time [93]. Since release 6.1 (2007) the Eiffel language has incorporated the void safety mechanism into the compiler, requiring developers to indicate, for every attribute, local variable or function result type, whether it can accept void values (keyword `detachable`) or not (keyword `attached`). It is possible to select a default, so that only one of the two keywords is necessary.

One rule concerns assignment and argument passing: if the target is `attached`, the source must be `attached` too. If the target is `detachable`, the source can be either `attached` or `detachable`. Another rule states that an attribute declared as `attached` should be attached to an object before any creation feature of the class terminates, otherwise the compiler will signal an error. Finally, in the cases in which we are sure that there is an object attached to a `detachable` variable `ref_A`, the assignment can be done within an *object test*, that will guarantee the attachment as illustrated below.

```
if attached {B} ref_A as x then
  -- do something with x, now certainly of type B
end
```

Listing A.1: Object test.

APPENDIX B

THE PERSISTENCE LIBRARY'S BACKEND IMPLEMENTATION

The back-end API provides a way to interact with the more advanced framework functionalities and the repository implementations.

The deferred class *REPOSITORY* is the gate from the front-end to the back-end, provides the highest level of abstraction of persistence operations and works with plain Eiffel objects. This means that to provide persistence services the framework does not require any specific annotation from developers.

This appendix describes in Sections B.1 and B.2 the details of the implementation of the Object-Relational Mapping (ORM) layer of the framework, which contributes significantly to abstract the database away from the object-oriented application. Section B.3 describes how we automatically generate the database schema. Section B.4 describes the library support for transactions and errors. Section B.5 illustrates the library's support for additional relational databases, custom ORM mappings, non-relational stores, and other extensions.

B.1 The ORM layer: from *REPOSITORY* to *BACKEND*

Class *RELATIONAL_REPOSITORY* has the responsibility to decompose the object graph of each object coming from the front-end into object-parts, for example attributes and collections. Each part is then mapped to its relational counterpart and sent to the *BACKEND* class. The deferred class *BACKEND* only deals with one object graph part at once, and it is responsible to map it to the actual persistence mechanism. The ORM layer lies be-

tween the *RELATIONAL_REPOSITORY* and the *BACKEND*. As said previously, the *RELATIONAL_REPOSITORY* class has the responsibility to decompose the object graph of each object coming from the front-end into object parts. The decomposition (see Figure B.1) is designed to support different object-

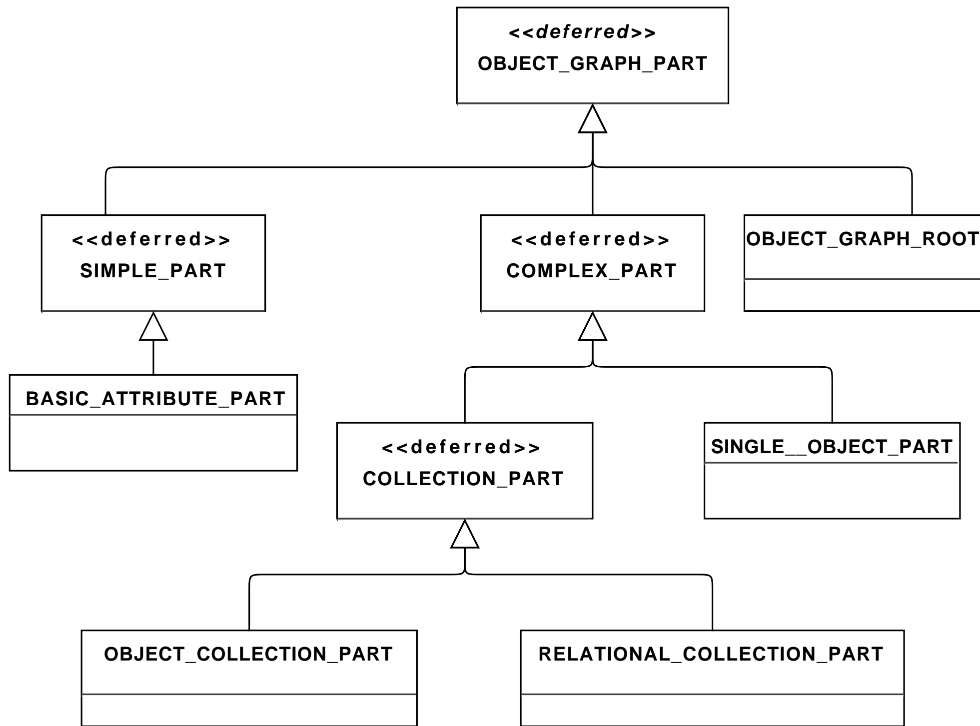

Figure B.1: *OBJECT_GRAPH_PART* hierarchy

relational mapping techniques, because each object part is easily mapped to a relational database. The set of classes used to represent an object graph and to prepare a write operation in the *BACKEND* is described below:

- *OBJECT_GRAPH_ROOT*: represents the root of the object graph.
- *BASIC_ATTRIBUTE_PART*: represents an object of a basic type.
- *OBJECT_COLLECTION_PART*: represents a collection to be stored in an object-oriented fashion.
- *RELATIONAL_COLLECTION_PART*: represents a collection to be stored in a relational fashion.

- *SINGLE_OBJECT_PART*: represents an object that is neither a basic type nor a collection.

The classes described above interact with the main classes in the ORM layer, described below (see also Figure B.2):

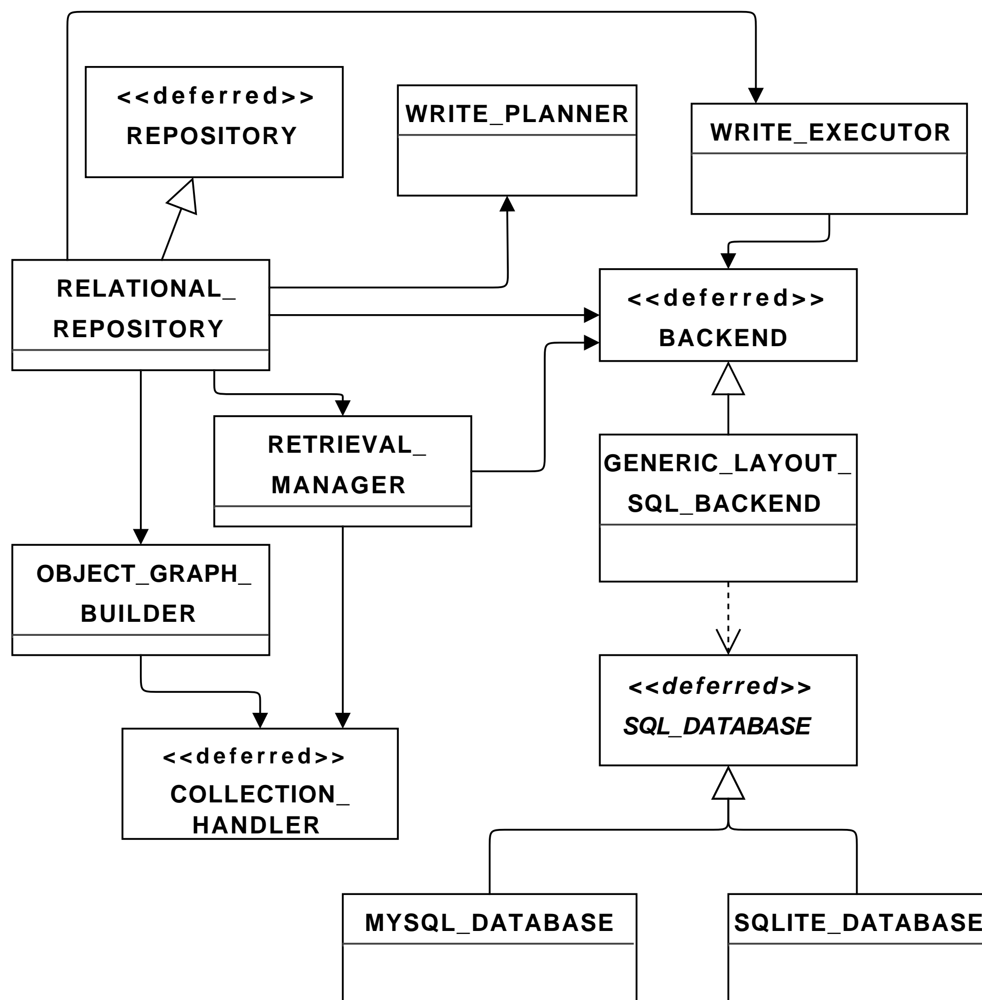

Figure B.2: The Object-to-Relational API important classes.

- *OBJECT_GRAPH_BUILDER*, responsible to create the explicit object graph representation.

- *WRITE_PLANNER*, responsible to generate a total order on all write operations, taking care of the dependency issues.
- *WRITE_EXECUTOR*, responsible to perform the single write operations against the back-end.
- *RETRIEVAL_MANAGER*, responsible for building objects from the parts obtained from the backend, and responsible for loading the entire object graph.
- *COLLECTION_HANDLER*, responsible to add collection handling support to the basic ORM layer. The actual support is implemented in its descendants.

To better explain how the ORM classes described above interact with each other, Figure B.3 shows how inserting and retrieving objects works using a UML collaboration diagram.

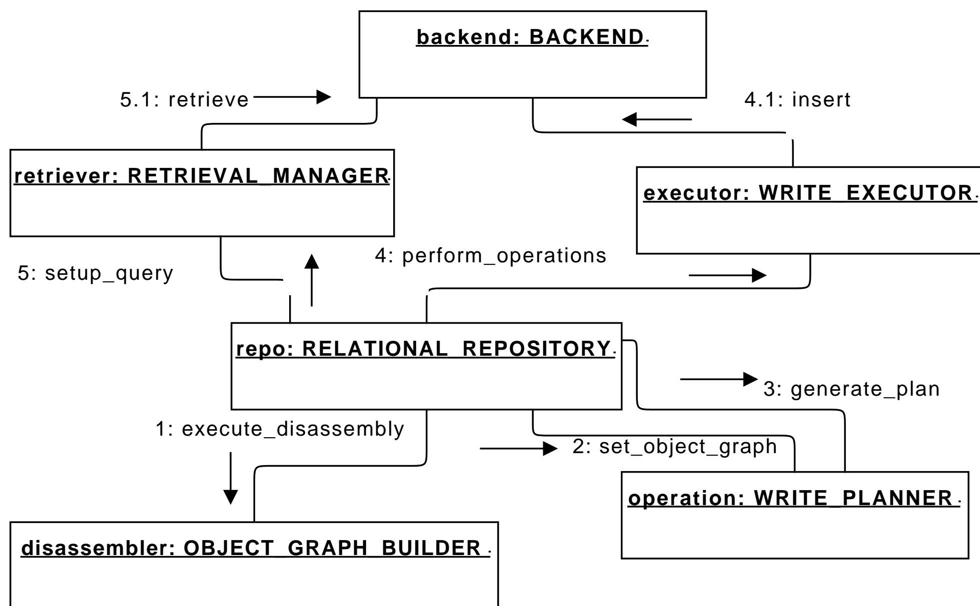

Figure B.3: Inserting and retrieving objects using *RELATIONAL_REPOSITORY*.

Finally, to clarify the data flow it may be useful to further describe what happens to an object when being stored by the ORM layer:

1. The `RELATIONAL_REPOSITORY` passes the object to the `OBJECT_GRAPH_BUILDER`.
2. The `OBJECT_GRAPH_BUILDER` creates the corresponding object graph and passes the `OBJECT_GRAPH_ROOT` to the `WRITE_PLANNER`.
3. The `WRITE_PLANNER` creates a `LIST [OBJECT_GRAPH_PART]` and passes it to the `WRITE_EXECUTOR`.
4. The `WRITE_EXECUTOR` separately invokes write operations on the back-end for each element of `LIST [OBJECT_GRAPH_PART]`, for example for each basic type and collection.

Object retrieval is comparatively simpler: the `RELATIONAL_REPOSITORY` passes a query object to the `RETRIEVAL_MANAGER`, which handles the communication with the `BACKEND` to retrieve the object parts and put them together to reconstruct the object(s) and give it back to the `RELATIONAL_REPOSITORY`.

B.1.1 Collection handling

Collections are important in an object-relational mapping scenario, because they are mapped as relations (1:M or M:N) in the back-end. The framework offers support for custom collections. In particular, developer can implement support for their own collections by inheriting from `PS_COLLECTION_HANDLER [COLLECTION_TYPE -> ITERABLE [ detachable ANY]]`, where the constrained generic parameter requires the collection type to be iterable.

The framework supports two types of collections:

- `RELATIONAL_COLLECTION_PART` fits well the scenario in which there is a typical database layout, with tables for every single class and collections stored in one of two ways:
  - Collections stored within the table of the referenced object as 1:M-Relations (see Figure B.4). These collections are not forwarded to the backend. Instead, each item in the collection gets a new dependency, expressed with a foreign key to the collection owner.
  - Collections stored inside their own table as M:N-Relations (see Figure B.5). While these collections depend on both the collection owner and all the items the collection references, each collection owner does not depend on the collection itself. This is

because to insert a single row in an M:N-relation associative table we need two foreign keys, one to an owner and the other to a collection item. In other words, by using an associative table a M:N-Relation is transformed into two one-to-many-relations (1:M and 1:N).

- *OBJECT_COLLECTION_PART* is intended for a scenario in which there is a separate table storing the collections, each with their own primary key value, while the collection owner is using this key as a foreign key (see Figure B.6). In this case the owner of the collection object depends on the collection, and the collection depends on all the items it references.

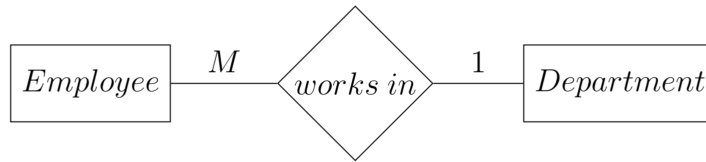

Figure B.4: An ER-model where a *RELATIONAL_COLLECTION_PART* with 1:M mapping can be used.

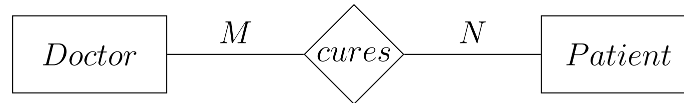

Figure B.5: An ER-model where a *RELATIONAL_COLLECTION_PART* with M:N mapping can be used.

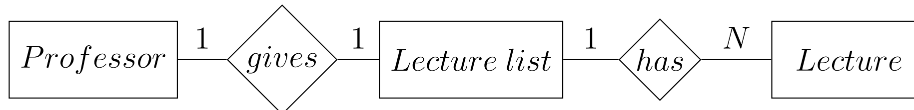

Figure B.6: An ER-model where an *OBJECT_COLLECTION_PART* can be used.

Knowing about classes *RELATIONAL_COLLECTION_PART* and *OBJECT_COLLECTION_PART* is not important for the framework user when using one of the predefined back-ends. It becomes essential if in need of extending the framework to a specific database layout.

B.1.2 Handling object references

A running object-oriented application is represented in memory by means of a graph, where objects are vertexes and references are (directed) edges (see Figure B.7).

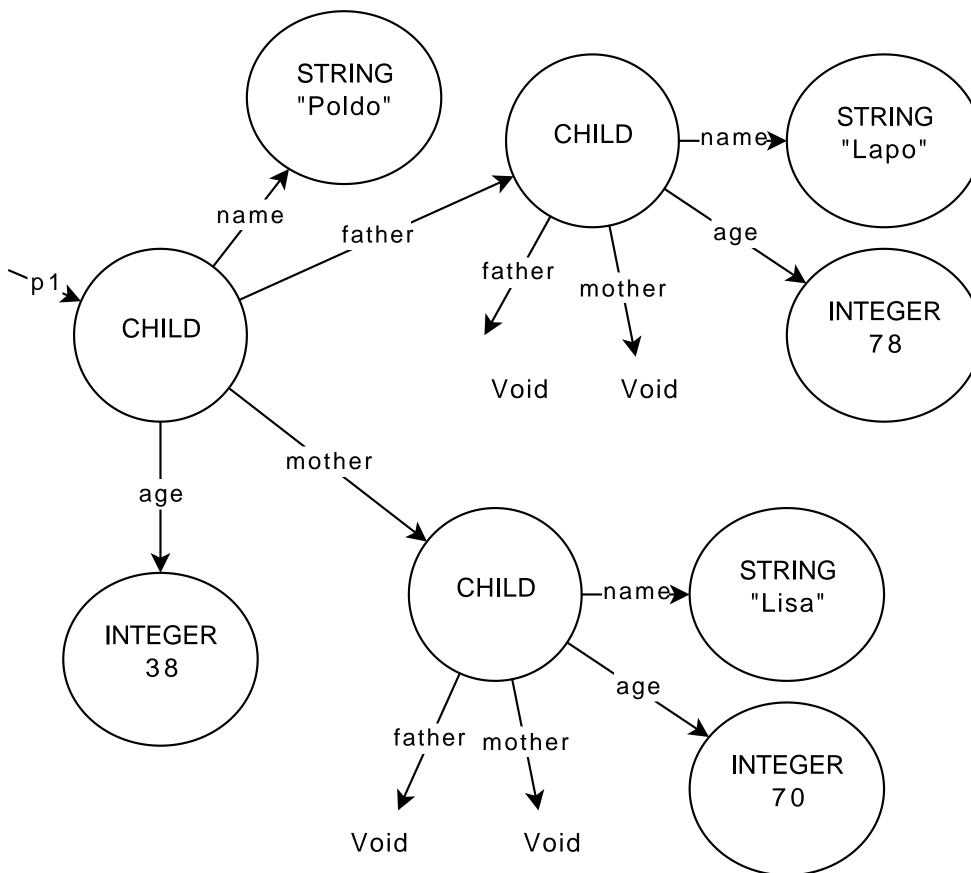

Figure B.7: An object graph in main memory.

We now define a few concepts connected to an application graph, useful for the remainder of the section to understand how the framework handles object graphs, and to explain what kind of flexibility the framework offers:

- “Global” application graph: the object graph of an entire application in memory.

- An object Y is "reachable" from another object X if there is a path between X and Y in the graph, or equivalently if Y is in the transitive closure of X.
- The "object graph of an object X" is the subgraph of the global object graph containing the transitive closure of X.
- The "level" of an object Y contained in the object graph of X is the length (number of arcs) of the shortest path from X to Y.

We will now use the definitions above to see where the framework stores the object graph settings and how it lets developers adjust the defaults, typically to tweak the framework's performance.

The most relevant class in this context is *OBJECT_GRAPH_SETTINGS*, containing a distinct depth attribute for each CRUD operation, each separately settable:

- *insert_depth*
- *query_depth*
- *update_depth*
- *delete_depth*

A depth value of 1 (the minimum) means that only the "basic" types (numeric types, boolean types, and strings) are considered (loaded, inserted, updated or deleted), but no referenced object is considered (see Figure B.8).

A depth of 2 means that, in addition to the basic types, also all the referenced objects are considered, but not further objects referenced by them in turn, and so on. By saying that an object reference is "not considered" we mean that it will point to **Void**. If we set depth 2, all the graph in the previous figures will be considered.

The special depth value *Object_graph_depth_infinite* means that an object has to be loaded completely instead.

The framework default values are *Object_graph_depth_infinite* for inserting and reading objects, and a value of 1 for *update_depth* and *delete_depth*. These default values can be overwritten for a whole repository by means of feature *default_object_graph_settings* in class *REPOSITORY*.

Assuming to have the default values for object graph settings, if we look at Figure B.8 and want to invoke a delete on object $p1$, the objects

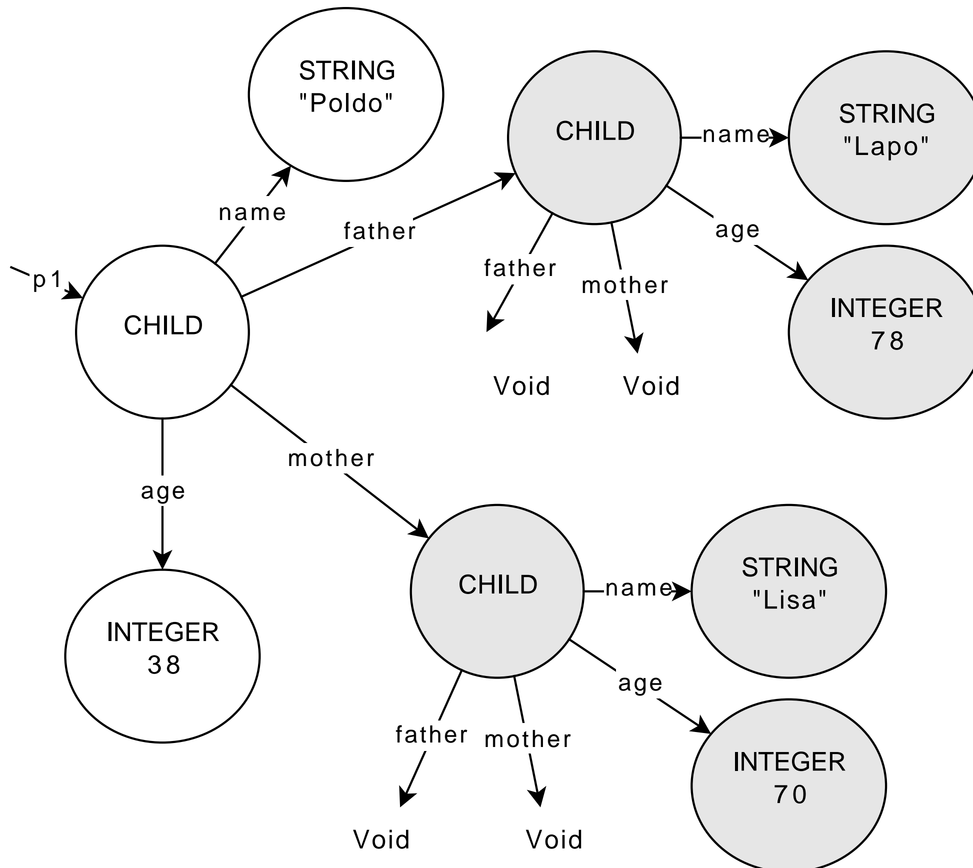

Figure B.8: The previous object graph at depth 1 (grey nodes are not considered).

referenced by $father$ and $mother$ will not be deleted because the default value for `delete_depth` is 1.

This is consistent with the general framework rule that a CRUD operation will only consider an object when the $level(object) < depth$ condition holds.

In the case of the delete in the previous example, the only object satisfying the condition above is the root, because the root has a level of 0 and the default value for `delete_depth` is 1. The situation illustrated in the previous example can be generalized by saying that when an object satisfies the condition $depth = level + 1$, all the basic attributes of the object are

considered, but its references only get considered if the referenced objects are already persistent, in other words if their corresponding object values are already in memory (likely from a previous operation). If this is not the case, then these references will point to **Void**.

The obvious caveat of setting the depth parameters is that it can cause an application to have objects that do not have all the attributes fully loaded in memory, and so one cannot completely rely on them. On the positive side, in Eiffel it is always possible to enforce class invariants checks, so semantically invalid objects would never be accepted in the software system.

Finally, there are other settings in class *OBJECT_GRAPH_SETTINGS* there are worth mentioning:

- *is_update_during_insert_enabled* asks if an object should be updated, if it is already in the database and it is referenced by a new object (default: no).
- *custom_update_depth_during_insert* allows to set the depth for an update of an object, which is found during an insert and it is already in the database (default: *update_depth*).
- *is_insert_during_update_enabled* asks if the framework should automatically insert objects which are not present in the database and found during an update (default: yes).
- *custom_insert_depth_during_update* allows to set the depth for an insert of an object, which is found during an update, but it is not present in the database (default: *insert_depth*).

B.2 The ORM layer: from *BACKEND* to the database

In Figure B.9 we see the lowest level of framework abstractions.

Here is a brief description of the most important classes:

- *SQL_DATABASE* represents an SQL database. The two implementations that come together with the framework as *MYSQL_DATABASE* and *SQLITE_DATABASE*.
- *SQL_CONNECTION* represents a single connection, used to wrap and forward SQL statements to the database, and to provide back an iteration cursor of *SQL_ROW*s. The two implementations that come together with the framework as *MYSQL_CONNECTION* and *SQLITE_CONNECTION*.

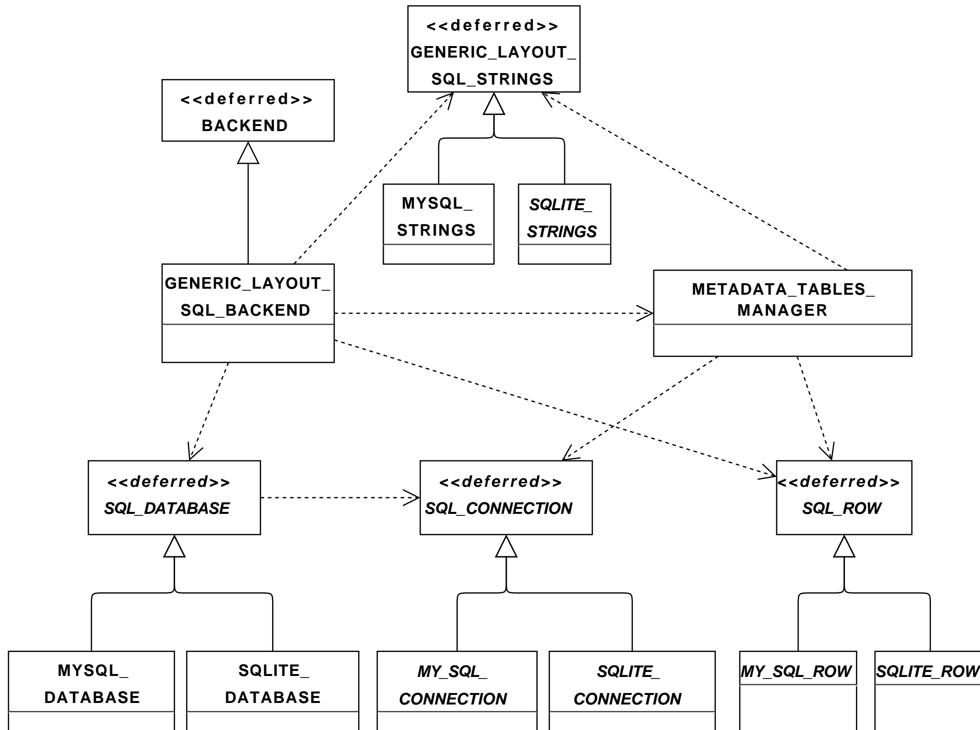

Figure B.9: From *BACKEND* to the databases.

- *SQL_ROW* represents a single row in the result of an SQL query. The two implementations that come together with the framework as *MYSQL_ROW* and *SQLITE_ROW*.

- *GENERIC_LAYOUT_SQL_STRINGS* represents SQL strings across databases. Different string implementations for specific databases are kept in descendants like *MYSQL_STRINGS* and *SQLITE_STRINGS*.

- *GENERIC_LAYOUT_SQL_BACKEND* is at the heart of the object-relational mapping framework capabilities (see Section B.3).

- *METADATA_TABLES_MANAGER* is responsible for reading from and writing to the metadata database tables representing classes and attributes (see Section B.3).

- *OBJECT_IDENTIFICATION_MANAGER* maintains a weak reference to each object that has been retrieved or inserted before, and assigns a repository-wide unique *object_identifier* to it.

- `KEY_POID_TABLE` maps the objects identified by the `object_identifier` to the primary key of the corresponding entry in the database.

B.3 Automatically generating the database schema

While class `BACKEND`'s implementations are not restricted to relational databases, we are now focusing on class `GENERIC_LAYOUT_SQL_BACKEND`, because it provides the most interesting challenges. This is the place where the actual mapping to a relational database is implemented, and it has to do with the kind of database layout chosen, which in turn is one of the most important design choices of the framework.

Given that in this framework we give priority to flexibility, extensibility and a smooth schema evolution, we decided to go, as a default (meaning when no pre-existing database layout is in place) for a schema in which the framework creates and handles meta-tables (see also [2]). The idea is pretty straightforward: the database layout is based upon class metadata, so there will be a table "class", containing the class names, a table "attribute" containing the attribute names , a table "value" containing the attribute values and so on. In Figure B.10 there is an ER-model showing a simplified view of the database model.

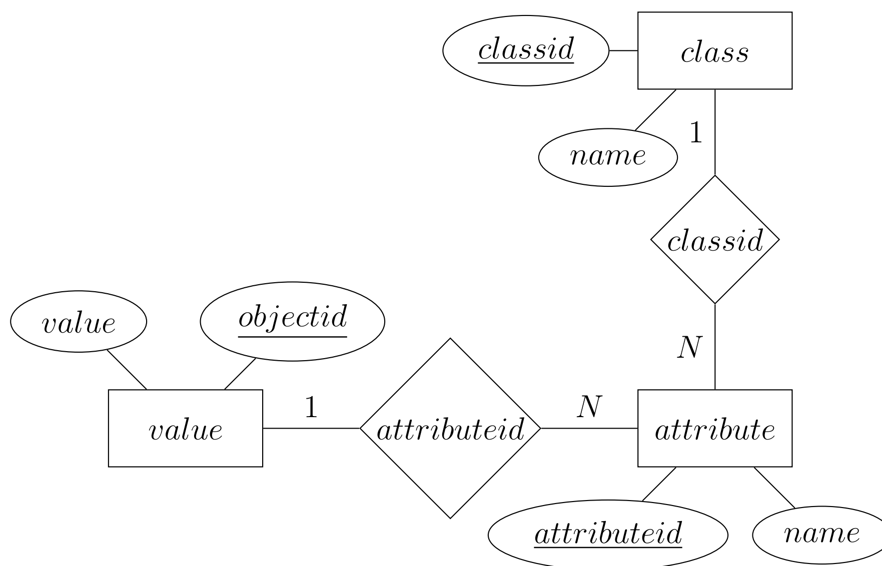

Figure B.10: The ER-Model of the generic database layout.

From the ER-model it should be clear that there are certain class attribute changes that are handled automatically by the framework, without need to modify the ORM layer code:

- A class added can be handled by adding a tuple in the table corresponding to the ″class″ entity.
- A class removed can be handled by removing a tuple in the table corresponding to the ″class″ entity.
- A class renamed can be handled by adding a tuple with the new name in the table corresponding to the ″class″ entity and then removing the tuple with the old name.
- An attribute added can be handled by adding a tuple in the table corresponding to the ″attribute″ entity.
- An attribute removed can be handled by removing a tuple in the table corresponding to the ″attribute″ entity.
- An attribute renamed can be handled by adding a tuple with the new name in the table corresponding to the ″attribute″ entity and then removing the tuple with the old name.
- An attribute that changed type can be handled by changing the ″classid″ in its tuple.

While class *GENERIC_LAYOUT_SQL_BACKEND* is responsible for read and write operations on the “value” meta-table, class *METADATA_TABLES_MANAGER* is responsible for read and write operations on the “class” and “attribute” meta-tables. Of course these tables will be created if they do not exist in the database.

The information class *GENERIC_LAYOUT_SQL_BACKEND* needs to write a tuple in table “value” is the following:

- The primary key “objectid” (stored in the *KEY_POID_TABLE* or generated automatically during an insert).
- The “attributeid” (a foreign key provided by *METADATA_TABLES_MANAGER*).
- The attribute value (provided by *SINGLE_OBJECT_PART*).

The information class *METADATA_TABLES_MANAGER* needs to write in tables "class" and "attribute" is generated when needed ("classid" and "attributeid"), or obtained using the introspection facilities of the programming language (class name and attribute name).

For example, to retrieve all attribute values of a class:

- *METADATA_TABLES_MANAGER* provides the class primary key.
- *METADATA_TABLES_MANAGER* provides the attributes primary keys.
- A predefined SQL query retrieves the corresponding value for each attribute using the above attribute key and class key.
- The result is finally sorted according to the "objectid", so that the attributes of the same object are grouped together.

B.4 Framework support for transactions and errors

Transactions are ubiquitous in the framework. Every CRUD operation invoked by the front-end is wrapped automatically in an implicit transaction, and it is also possible to handle transactions explicitly and programmatically by using the available features in class *CRUD_EXECUTOR*:

- *insert_within_transaction* (*an_object*: *ANY*; *transaction* : *TRANSACTION*)
- *execute_query_within_transaction* (*a_query*: *OBJECT_QUERY* [*ANY*]; *transaction*: *TRANSACTION*)
- *update_within_transaction* (*an_object*: *ANY*; *transaction* : *TRANSACTION*)
- *delete_within_transaction* (*an_object*: *ANY*; *transaction* : *TRANSACTION*)

For example one could decide to rollback the transaction at a certain point of execution by invoking feature *rollback* in class *TRANSACTION*.

The back-end is also very aware of transactions, because it is interfacing the actual database and needs to be able, for example, to handle the ACID (Atomicity Consistency Isolation Durability) properties. The framework supports the four standard transaction isolation levels found in almost every database system:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

The different levels are defined in class *TRANSACTION_ISOLATION_LEVEL*, and the default transaction isolation level is defined by the current back-end.

The other responsibility of class *TRANSACTION* is error propagation. The framework makes a distinction between irrecoverable errors (like a dropped connection or a database integrity constraint violation) and recoverable errors (like a conflict between two transactions). For irrecoverable errors the default behavior is to rollback the current transaction and raise an exception. As the exception propagates upwards, every layer in the backend has a chance to take the appropriate steps in an attempt to bring the library back to a consistent state, typically using the error information stored in the transaction itself. For recoverable errors, the default behavior is to attempt to recover from the error, for example retrying a certain operation. The framework does not raise an exception in this case.

The framework maps database-specific error messages to its own representation for errors, which is a hierarchy of classes rooted at *ERROR*. Figure B.11 illustrates part of the hierarchy:

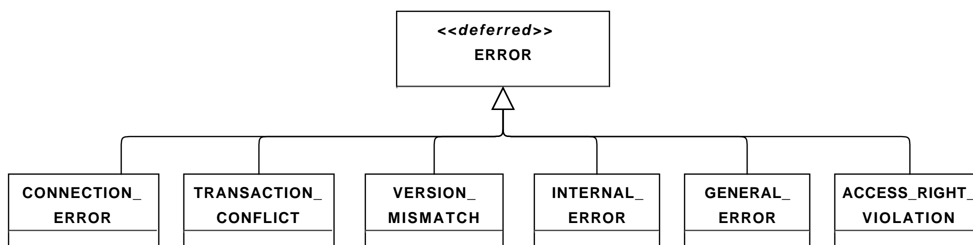

Figure B.11: Partial error class hierarchy.

To handle errors in a convenient way, the framework also offers *ERROR_VISITOR*, a utility class that developers can inherit from to implement their own "visitors" to react in the most appropriate way to the various errors (for the visitor design pattern see [66]).

B.5 Framework extension points

The framework has been designed with extensibility in mind. Apart from the automatic support for class and attribute changes seen in Section B.3, developers can also extend the framework by using some predefined extension points. The following subsections describe how to add support for an additional relational database, additional ORM mappings, non-relational databases, and for cross-implementation extensions.

B.5.1 Supporting an additional relational database

Providing support for a custom relational back-end (say Oracle) is relatively simple. If the kind of object-relational meta-mapping the framework described in Section B.3 is satisfying, then all developers need to do is writing the implementations for four new classes:

- `ORACLE_DATABASE`, which will inherit from the framework class `SQL_DATABASE`, will handle the specific way in which the Oracle database interacts with the software applications.
- `ORACLE_STRINGS`, which will inherit from the framework class `GENERIC_LAYOUT_SQL_STRINGS`, will contain the SQL command strings specific to an Oracle database.
- `ORACLE_CONNECTION`, which will inherit from the framework class `SQL_CONNECTION`, will wrap and manage a connection to an Oracle database.
- `ORACLE_ROW`, which will inherit from the framework class `SQL_ROW`, will handle the Oracle result rows.

The rest of the framework should stay unchanged.

B.5.2 Supporting additional ORM mappings

If developers need to adapt to an existing relational database schema instead of relying on the meta-schema the framework offers, then they have to write your own class that inherits from `BACKEND`, and implement another class that inherits from `COLLECTION_HANDLERS` to handle the mapping of all the collections that need it.

To outline the simplicity of the approach, we will show via some source code examples how to do it. We will use two simple domain classes: class `PERSON` (see Figure 5.1), and class `ITEM`:

```
class
  ITEM

feature -- Status

  value: INTEGER
    -- The value of current item.

  -- Remainder omitted.

end
```

Listing B.1: Domain class `ITEM`

Let's suppose that the database schema is the one illustrated in Figure B.12. In the database, table `Items` has a foreign key `owner` to table `Persons`.

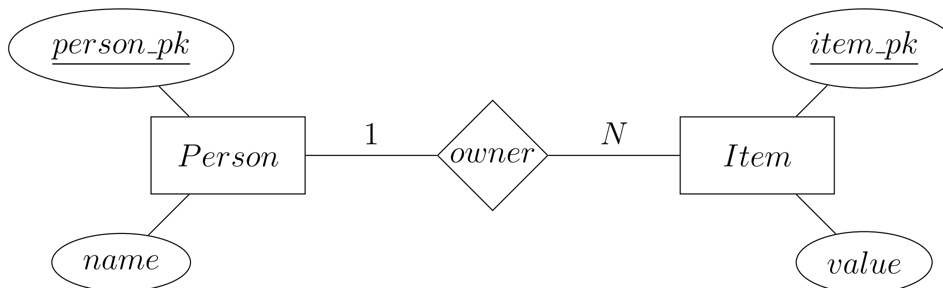

Figure B.12: ER-Model for persons and items.

One of the two classes to code is a specific collection handler for `LINKED_LIST`, whose instances are mapped as a 1:M relationship. Figure B.2 shows the code of class `LINKED_LIST_HANDLER` which inherits from the generic framework class `PS_COLLECTION_HANDLER [G]`. The value of the generic parameter is the collection to handle: `PS_COLLECTION_HANDLER [LINKED_LIST [detachable ANY]]`. Notice that elements of the linked list can be of any type, and even void. This is because the framework is fully void-safe, which means that the compiler enforces that no exceptions caused by a void reference can happen. To achieve this, it is necessary to explicitly declare that a certain reference could be possibly void.

```
class
  LINKED_LIST_HANDLER
```

```
inherit
 PS_COLLECTION_HANDLER [LINKED_LIST [detachable ANY]]

feature -- Queries

 is_relationally_mapped (collection, owner_type:
    TYPE_METADATA): BOOLEAN
   -- Is 'collection' mapped as a 1:N or M:N - Relation?
   do
    Result:= True
   end

 is_mapped_as_1_to_N (collection, owner_type:
    TYPE_METADATA): BOOLEAN
   -- Is 'collection' mapped as a 1:N - Relation?
   do
    Result:= True
   end

feature -- Basic Operations

 build_relational_collection (collection_type:
    TYPE_METADATA; objects: LIST[ detachable ANY]):
    LINKED_LIST[detachable ANY]
   -- Build a new LINKED_LIST.
   do
    create Result.make
    Result.append (objects)
   end

end
```

Listing B.2: The collection handler for *LINKED_LIST*

The implementation of the second needed class , *CUSTOM_BACKEND*, will inherit from the framework class *BACKEND*, and will appropriately handle the insertion and retrieval of *PERSON* and *ITEM* objects into the database.

In this case the object-relational mapping layer of the framework adds an attribute with name *items_owned* to the *ITEM* object. This is the default behavior for 1:M relations as described previously.

```
class
 PERSON

feature -- Status
```

```
  name: STRING
    -- Name of current person.

  age: INTEGER
    -- Age of current person.

  items_owned: LINKED_LIST [ITEM]
    -- Added by the ORM framework layer.

  -- Remainder omitted.
end
```

Listing B.3: Domain class `PERSON` extended by the ORM

The implementation of feature `insert (an_object: SINGLE_OBJECT_PART; a_transaction: TRANSACTION)` will have to check if the object passed is a `PERSON` object. If this is the case it will generate SQL to insert the person name and add an entry representing a primary key `Persons` into the `PS_KEY_POID_TABLE`. If the object passed is an `ITEM` object instead, it will generate SQL to insert the item value, the item owner as a foreign key and again an entry representing a primary key for `Items` into the `PS_KEY_POID_TABLE`.

For the retrieval operation, the process is similar: developers have to implement feature `retrieve_relational_collection`, and select the needed values from the appropriate table.

B.5.3 Supporting non-relational databases

Implementing support for a non-relational database, say a NoSQL database, can be achieved by first creating two new classes: one inheriting from `REPOSITORY`, and the other from `BACKEND`. The mapping layer will need to be programmed explicitly, but it will be relatively simpler than the ORM we have described in the previous sections, because many NoSQL data stores—like key-value data stores and document data stores—tend to have a design that is more similar to an object-oriented application with respect to a relational database.

B.5.4 Cross-implementation extensions

Apart from providing a custom, specific back-end, there are some features that the framework is designed to support. They are cross-implementation extensions, so once implemented, they will impact the whole framework.

They include:

- Custom attribute filtering: it is normal, in some scenarios, to require that certain attributes carrying temporary information are not stored. There is no need to mark the attributes as non-persistent, as some programming languages do, because the framework builds its own representation of the object graph, and so one can operate at that level. The solution will therefore include removing the unwanted attributes from the object graph decomposition discussed in Section B.1, and in particular from class *OBJECT_GRAPH_PART*. Of course during retrieval one needs to provide a reasonable default as well.

- Object caching: the framework comes with class *IN_MEMORY_DATABASE*, that apart from being used as an in-memory database for testing, it can also be used alongside the current backend to cache objects and therefore minimize the actual accesses to external databases.

- Adding support for class schema evolution has been relatively easy, again thanks to the flexibility of the *BACKEND* abstraction. The most important framework class is *VERSION_HANDLER* that inherits from *BACKEND*. This class adds a version attribute to all inserted objects, and checks the object versions during retrieval.

APPENDIX C

GRAPHS

This appendix illustrates the behavior of class invariants for the 8 Eiffel projects analyzed in Section 4.4.2. For each project, the different colored lines signal what happened to the invariants of the classes to whom attributes have been added or removed. As a consequence of at least one attribute added or removed, an invariant could stay the same, become stronger or become weaker. More specifically, each graph's legend lists:

- +A same: at least one attribute added, invariant stays the same.
- + A strong: at least one attribute added, invariant becomes stronger.
- + A weak: at least one attribute added, invariant becomes weaker.
- - A same: at least one attribute removed, invariant stays the same.
- - A strong: at least one attribute removed, invariant becomes stronger.
- - A weak: at least one attribute removed, invariant becomes weaker.

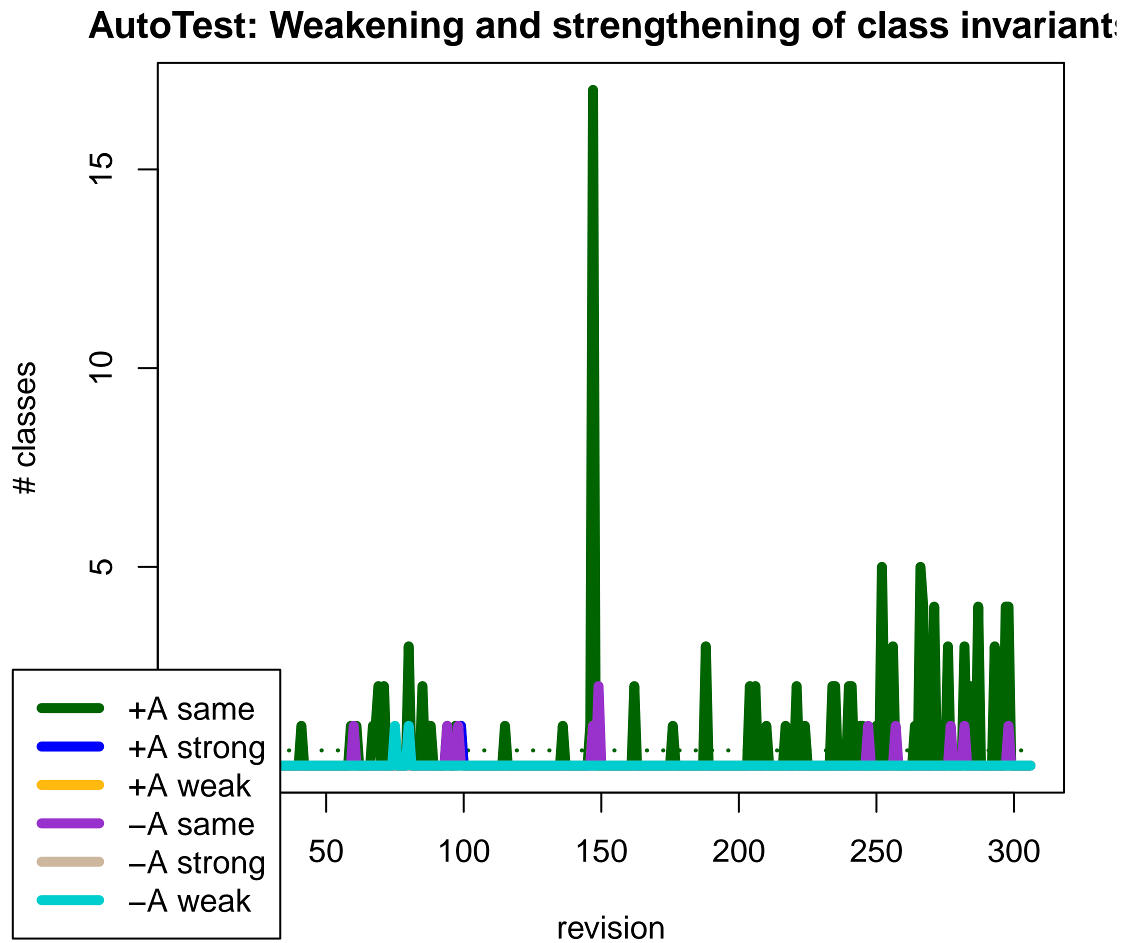

Figure C.1: AutoTest project: weakening and strengthening of class invariant when adding or removing attributes.

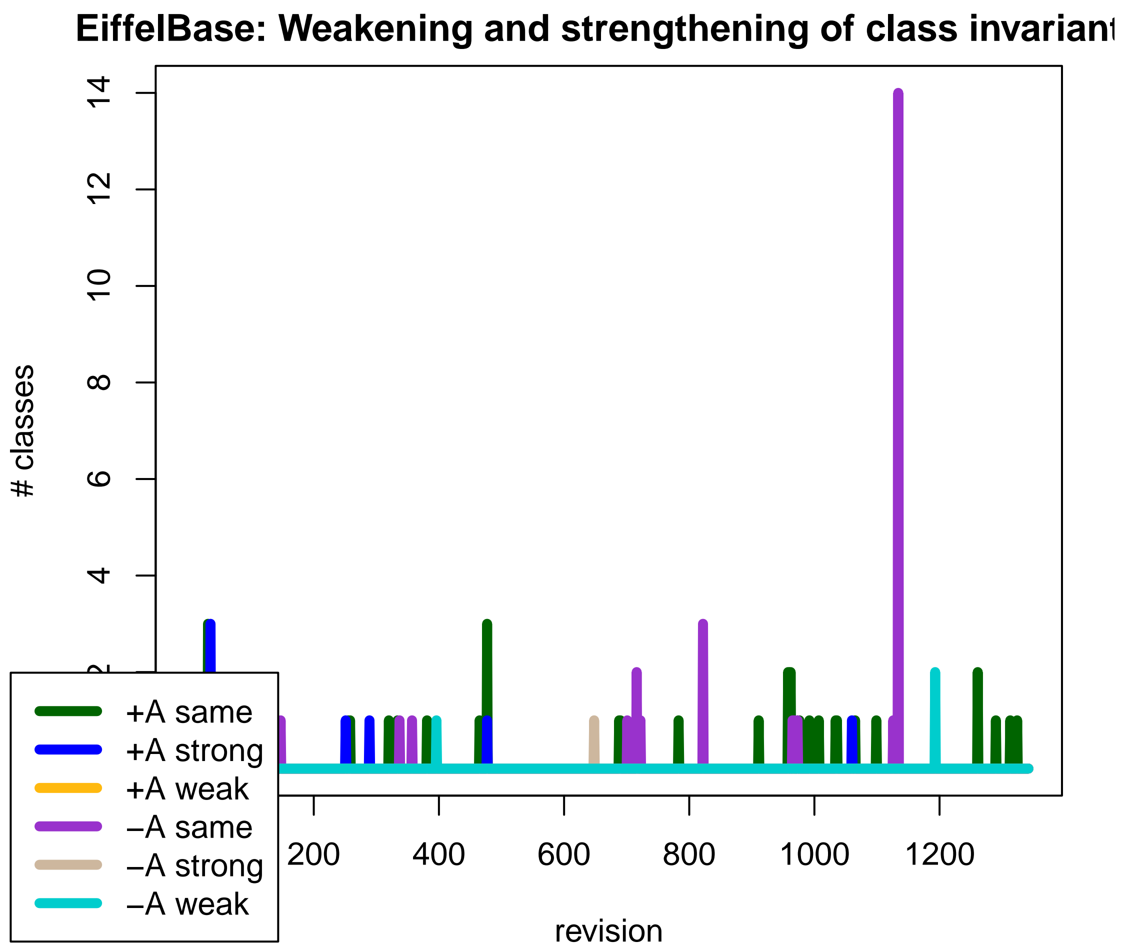

Figure C.2: EiffelBase project: weakening and strengthening of class invariant when adding or removing attributes.

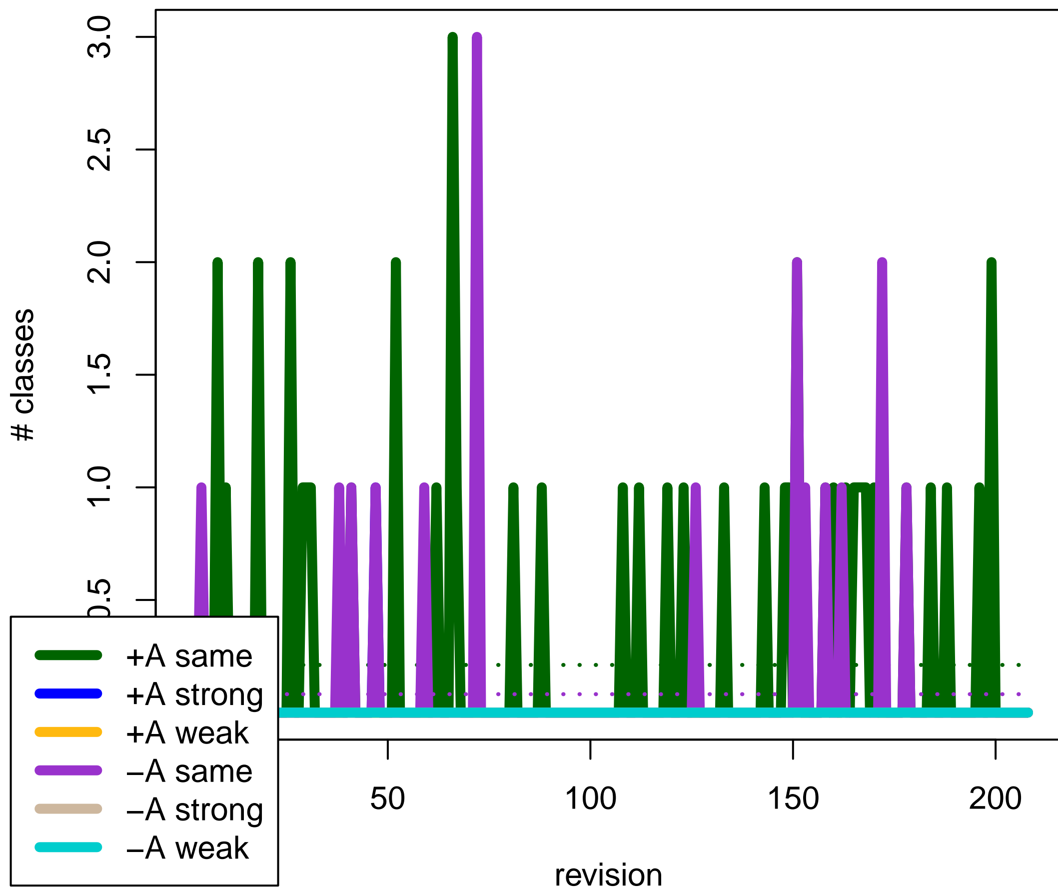

Figure C.3: Eiffel Program Analysis project: weakening and strengthening of class invariant when adding or removing attributes.

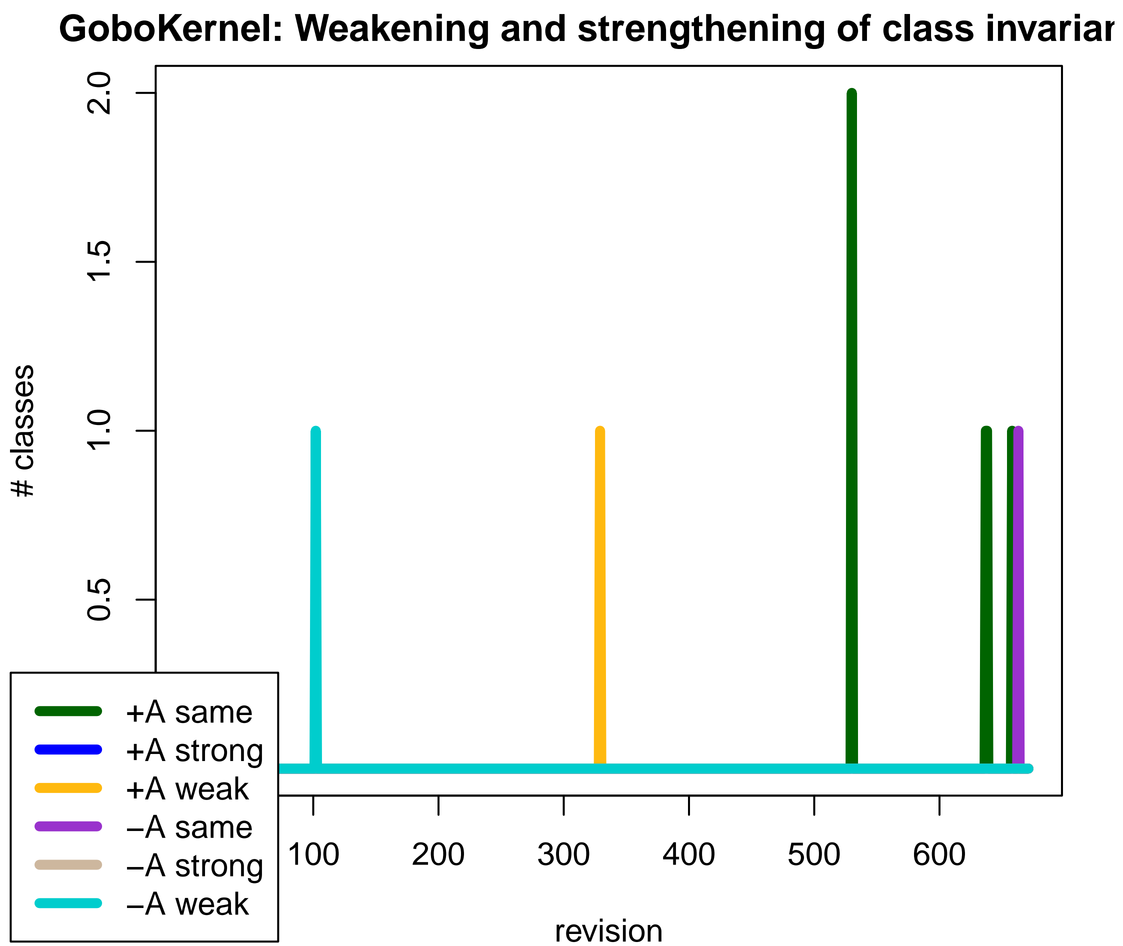

Figure C.4: Gobo Kernel project: weakening and strengthening of class invariant when adding or removing attributes.

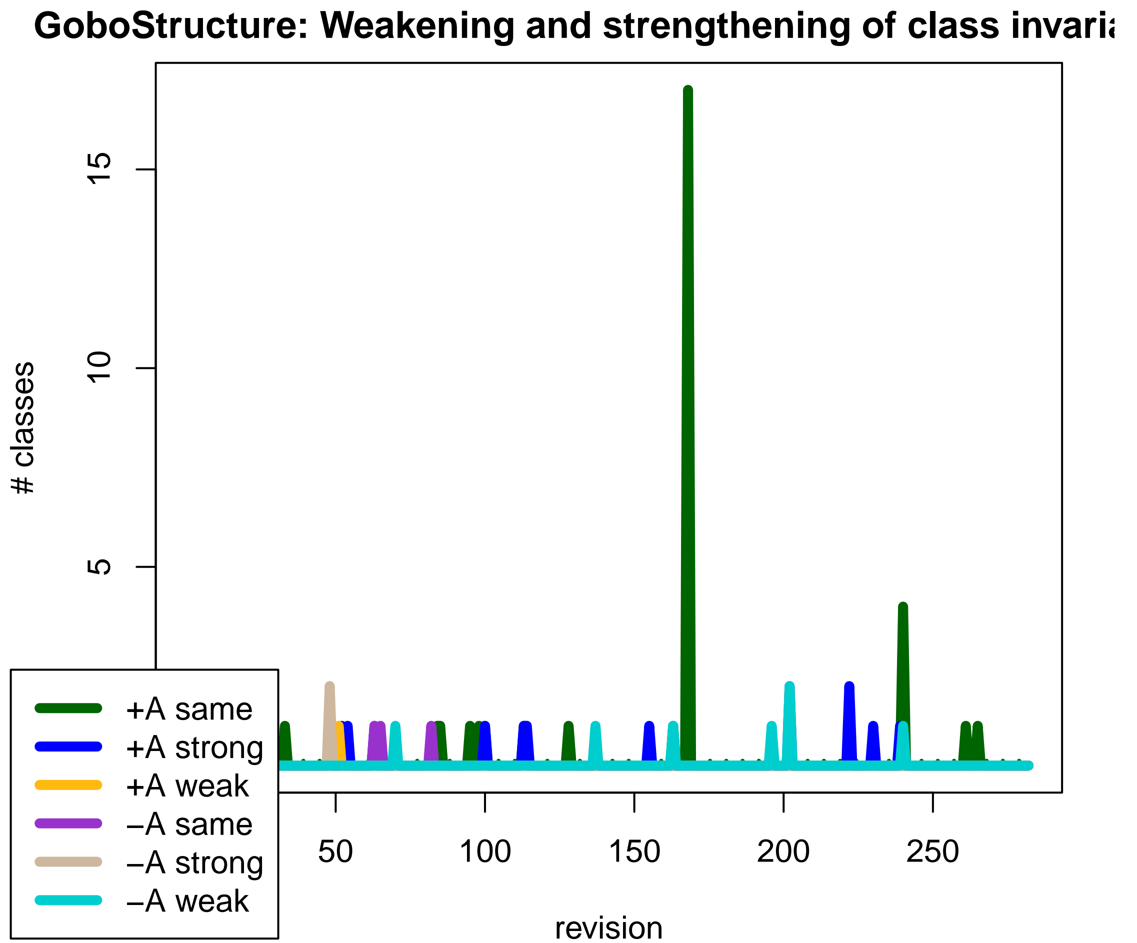

Figure C.5: Gobo Structure project: weakening and strengthening of class invariant when adding or removing attributes.

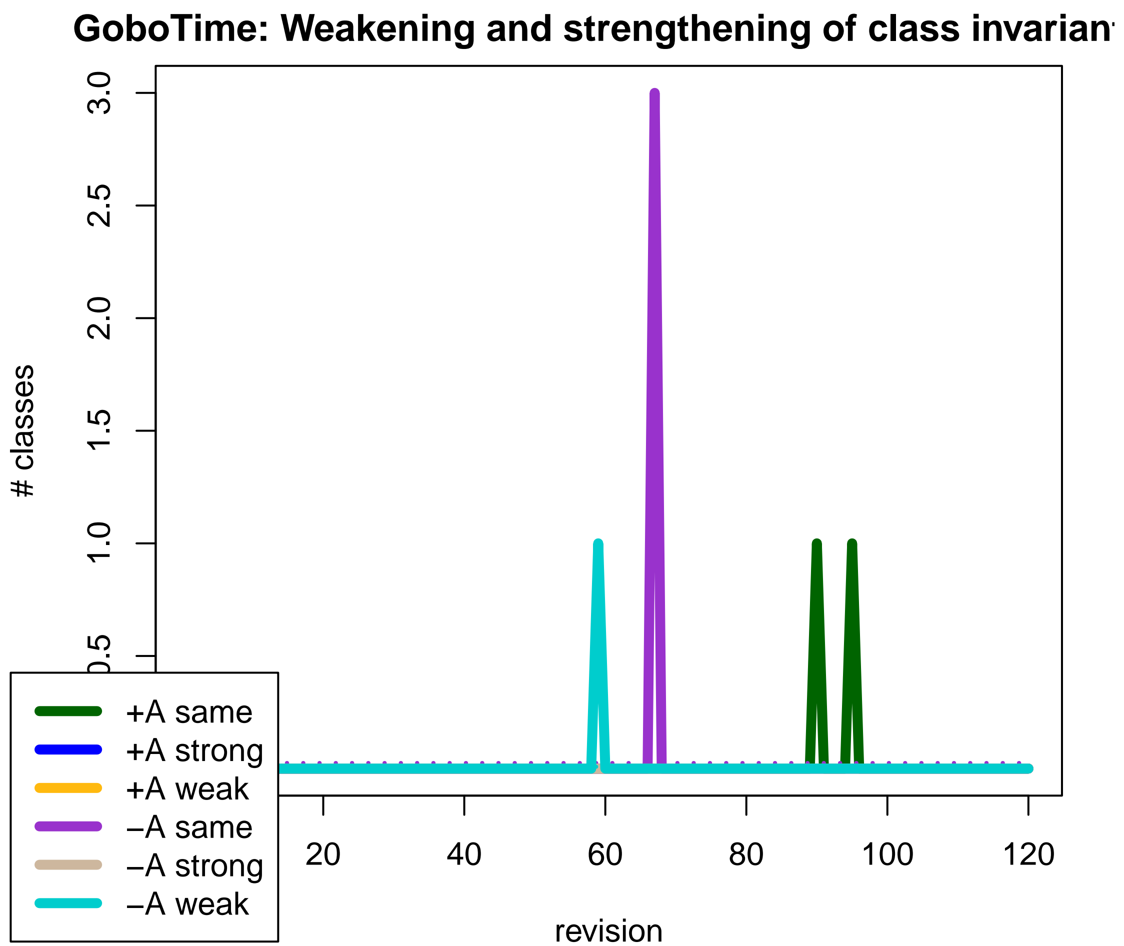

Figure C.6: Gobo Time project: weakening and strengthening of class invariant when adding or removing attributes.

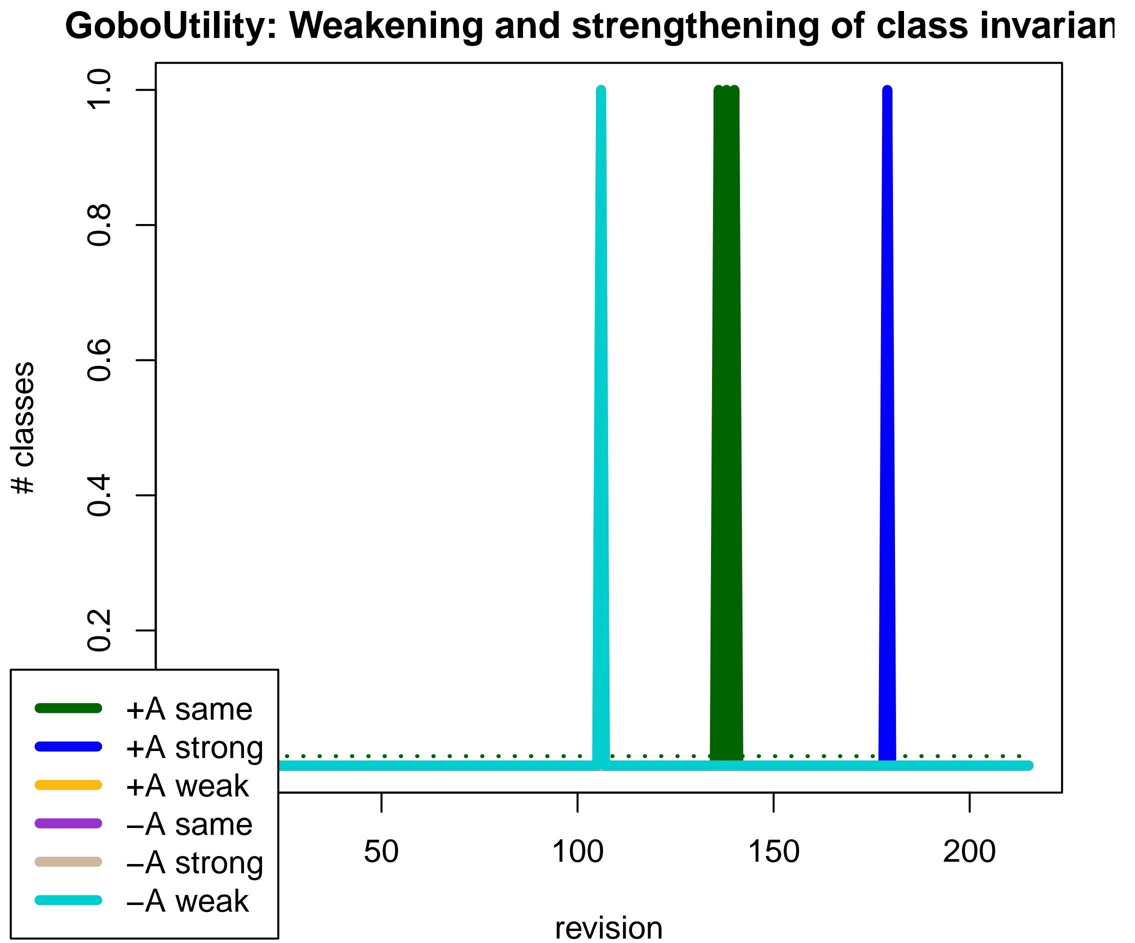

Figure C.7: Gobo Utility project: weakening and strengthening of class invariant when adding or removing attributes.

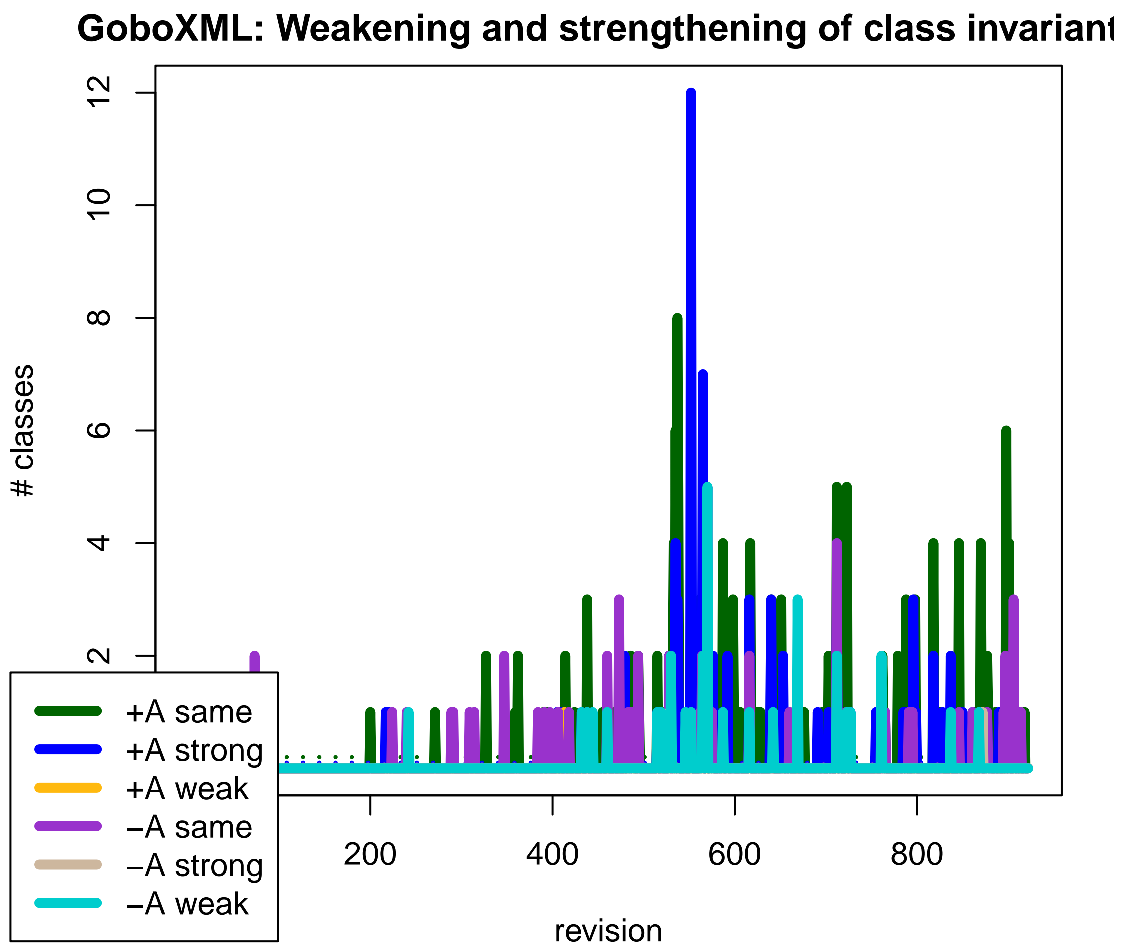

Figure C.8: Gobo XML project: weakening and strengthening of class invariant when adding or removing attributes.

BIBLIOGRAPHY

[1] Deepak Advani, Youssef Hassoun, and Steve Counsell. Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In *SAC*, pages 1713–1720, 2006.

[2] Scott Ambler. Mapping objects to relational databases: O/r mapping in detail. `http://www.agiledata.org/essays/mappingObjects.html`, 2010. Last visited: 5.08.2012.

[3] Malcolm P. Atkinson. Persistence and java - a balancing act. In *Objects and Databases*, pages 1–31, 2000.

[4] Malcom P. Atkinson. Programming languages and databases. In *VLDB*, pages 408–419, 1978.

[5] Malcom P. Atkinson and Mick Jordan. A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical Report SMLI TR-2000-90, Sun Microsystems Laboratories Technical Report, 2000.

[6] Malcom P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[7] Various authors. Schema evolution benchmark. `http://schemaevolution.org/`. Last visited: 22.07.2012.

[8] B. Mathiske B. Lewis and N. Gafter. Architecture of the pevm: A high-performance orthogonally persistent java virtual machine. Technical Report TR-2000-93, Sun Microsystems Laboratories Technical Report, 2000.

[9] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD Conference*, pages 311–322, 1987.

[10] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[11] Mike Barnett, K. Rustam M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS 2004, in Springer LNCS*, volume 3362, 2004.

[12] Gavin M. Bierman, Matthew J. Parkinson, and James Noble. Upgradej: Incremental typechecking for class upgrades. In *ECOOP*, pages 235–259, 2008.

[13] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and R. Michael Young. Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive Technology*, pages 325–341, 2001.

[14] J. Bloch. *Effective Java*. Prentice Hall, 2001.

[15] Daniel Bonniot. Type safety in nice. `http://nice.sourceforge.net/safety.html`, 2012. Last visited: 10.09.2012.

[16] M. Ted Boren and Judith Ramey. Thinking aloud: reconciling theory and practice. *IEEE Transactions on Professional Communication*, 43(3):261–278, 2000.

[17] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, pages 403–417, 2003.

[18] R. G. G. Cattell. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[19] Patrice Chalin. Are practitioners writing contracts? *Springer LNCS*, 4157:100–113, 2006.

[20] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC*, pages 222–231, 2008.

[21] Steven Clarke. Measuring api usability. http://www.drdobbs.com/windows/measuring-api-usability/184405654, 2004. Last visited: 15.10.2012.

[22] Steven Clarke. Using the cognitive dimensions - abstraction level. http://blogs.msdn.com/b/stevencl/archive/2003/11/14/57065.aspx, 2004. Last visited: 15.10.2012.

[23] Steven Clarke. Using the cognitive dimensions - api elaboration. http://blogs.msdn.com/b/stevencl/archive/2004/02/27/81317.aspx, 2004. Last visited: 15.10.2012.

[24] Steven Clarke. Using the cognitive dimensions - api viscosity. http://blogs.msdn.com/b/stevencl/archive/2004/03/10/87652.aspx, 2004. Last visited: 15.10.2012.

[25] Steven Clarke. Using the cognitive dimensions - consistency. http://blogs.msdn.com/b/stevencl/archive/2004/03/17/91626.aspx, 2004. Last visited: 15.10.2012.

[26] Steven Clarke. Using the cognitive dimensions - domain correspondence. http://blogs.msdn.com/b/stevencl/archive/2004/05/17/133439.aspx, 2004. Last visited: 15.10.2012.

[27] Steven Clarke. Using the cognitive dimensions - learning style. http://blogs.msdn.com/b/stevencl/archive/2003/11/24/57079.aspx, 2004. Last visited: 15.10.2012.

[28] Steven Clarke. Using the cognitive dimensions - penetrability. http://blogs.msdn.com/b/stevencl/archive/2004/02/13/72646.aspx, 2004. Last visited: 15.10.2012.

[29] Steven Clarke. Using the cognitive dimensions - premature commitment. http://blogs.msdn.com/b/stevencl/archive/2004/01/22/61859.aspx, 2004. Last visited: 15.10.2012.

[30] Steven Clarke. Using the cognitive dimensions - progressive evaluation. http://blogs.msdn.com/b/stevencl/archive/2003/12/22/45143.aspx, 2004. Last visited: 15.10.2012.

[31] Steven Clarke. Using the cognitive dimensions - role expressiveness. http://blogs.msdn.com/b/stevencl/archive/2004/04/23/119147.aspx, 2004. Last visited: 15.10.2012.

[32] Steven Clarke. Using the cognitive dimensions - working framework. http://blogs.msdn.com/b/stevencl/archive/2003/12/03/57112.aspx, 2004. Last visited: 15.10.2012.

[33] Steven Clarke. Using the cognitive dimensions -work step unit. http://blogs.msdn.com/b/stevencl/archive/2003/12/22/45142.aspx, 2004. Last visited: 15.10.2012.

[34] ECMA committee TC39-TG4. Ecma international standard: Eiffel analysis, design and programming language. Technical Report 367, ECMA committee TC39-TG4, 2005.

[35] Microsoft Corporation. Object internals visualization in microsoft debugger. http://msdn.microsoft.com/en-us/library/zayyhzts.aspx, 2012. Last visited: 19.09.2012.

[36] Oracle Corporation. Java version history. http://www.oracle.com/technetwork/java/archive-139210.html, 2012. Last visited: 25.09.2012.

[37] TechSmith Corporation. Camtasia screen recording and video editing software. http://www.techsmith.com/camtasia.html, 2012. Last visited: 9.09.2012.

[38] Versant Corporation. Db4o documentation. http://community.versant.com/documentation.aspx, 2012. Last visited: 4.08.2012.

[39] Versant Corporation. Versant object database fundamentals manual. http://www.versant.com/pdf/VODFundamentals.pdf, 2012. Last visited: 5.10.2012.

[40] Zope Corporation. Zope application server. http://www.zope.org/the-world-of-zope, 2012. Last visited: 5.10.2012.

[41] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow.*, 1:761–772, 2008.

[42] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (2nd Edition)*. Addison-Wesley Professional, 2008.

[43] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, pages 260–283, 2009.

[44] Jason Dagit, Joseph Lawrance, Christoph Neumann, Margaret M. Burnett, Ronald A. Metoyer, and Sam Adams. Using cognitive dimensions: Advice from the trenches. *J. Vis. Lang. Comput.*, 17(4):302–327, 2006.

[45] Marco D'Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. Analysing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. Springer, 2008.

[46] C. Date. *Introduction to Database Systems*. Addison Wesley, 8 edition, 2003.

[47] Kilian Sprotte David Lichteblau, Hans Hübner and Manuel Odendahl. Bknr lisp application environment. `http://bknr.net/html/home.html`, 2008. Last visited: 5.10.2012.

[48] Cecilia Delgado, José Samos, and Manuel Torres. Primitive operations for schema evolution in odmg databases. In *OOIS*, pages 226–237, 2003.

[49] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE*, pages 427–436, 2007.

[50] Mikhail Dmitriev. Safe class and data evolution in long-lived java applications. Technical Report SMLI TR-2001-98, Sun Microsystems Laboratories Technical Report, 2001.

[51] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in apis. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, pages 79–104, July 2011.

[52] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and answering questions about unfamiliar apis: An exploratory study. In *ICSE*, pages 266–276, 2012.

[53] Brian Ellis, Jeffrey Stylos, and Brad A. Myers. The factory pattern in api design: A usability evaluation. In *ICSE*, pages 302–312, 2007.

[54] K. Anders Ericsson and Herbert A. SImon. *Protocol Analysis: Verbal Reports as Data*. MTT Press, 1984.

[55] H.-Christian Estler, Marco Piccioni, Carlo A. Furia, and Martin Nordio. How specifications change and why you should care. Submitted to ICSE 2013.

[56] H.-Christian Estler, Marco Piccioni, Carlo A. Furia, and Martin Nordio. Experimental data: Contracts evolution. `http://se.inf.ethz.ch/data/coat/`, 2012. Last visited: 20.08.2012.

[57] Chair of Software Engineering ETH Zürich. Eiffel verification environment (eve). `https://trac.inf.ethz.ch/trac/meyer/eve/wiki/WikiStart`, 2012. Last visited: 19.09.2012.

[58] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.

[59] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In *MoDELS*, pages 326–340, 2008.

[60] Beat Fluri and Harald Gall. Classifying change types for qualifying change couplings. In *ICPC*, pages 35–45, 2006.

[61] Beat Fluri, Michael Würsch, and Harald Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *WCRE*, pages 70–79. IEEE, 2007.

[62] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[63] Apache Foundation. Apache tomcat source code. `http://svn.apache.org/viewvc/tomcat/`, 2012. Last visited: 25.09.2012.

[64] Python Software Foundation. Python object serialization. `http://docs.python.org/library/pickle.html`, 2012. Last visited: 13.08.2012.

[65] Enrico Franconi, Fabio Grandi, and Federica Mandreoli. Schema evolution and versioning: A logical and computational characterisation. In *FMLDO*, pages 85–99, 2000.

[66] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[67] GemStone. Gemstone/s 64 bit programming guide. `http://www.gemstone.com/docs/GemStoneS/GemStone64Bit/2.4.4.3/GS64-ProgGuide-2.4.pdf`, 2009. Last visited: 22.07.2012.

[68] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 3rd Edition*. Addison Wesley, 2005.

[69] Paul W. P. J. Grefen and Peter M. G. Apers. Integrity control in relational database systems - an overview. *Data Knowl. Eng.*, 10:187–223, 1993.

[70] Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards synchronizing models with evolving metamodels. In *MODSE*, 2007.

[71] William G. J. Halfond and Alessandro Orso. Detection and prevention of sql injection attacks. In *Malware Detection*, pages 85–109. Springer, 2007.

[72] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. Automatability of coupled evolution of metamodels and models in practice. In *MoDELS*, pages 645–659, 2008.

[73] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 2011.

[74] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, pages 9–14, 2007.

[75] Hans Hübner. Schema evolution made easy. `http://netzhansa.blogspot.com/2008/08/schema-evolution-made-easy.html`, 2008. Last visited: 5.10.2012.

[76] Apple Inc. Archives and serializations. `http://developer.apple.com/library/ios/#DOCUMENTATION/Cocoa/Conceptual/Archiving/Archiving.html#//apple_ref/doc/uid/10000047-SW1`, 2010. Last visited: 13.08.2012.

[77] Apple Inc. Class clusters. http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaObjects/CocoaObjects.html#//apple_ref/doc/uid/TP40002974-CH4-SW34, 2010. Last visited: 7.08.2012.

[78] Apple Inc. Cocoa design patterns. http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html#//apple_ref/doc/uid/TP40002974-CH6-SW6, 2010. Last visited: 7.08.2012.

[79] JetBrains. @nullable and @notnull annotations in intellij idea. http://www.jetbrains.com/idea/webhelp/nullable-and-notnull-annotations.html, 2012. Last visited: 10.09.2012.

[80] Mick Jordan. A comparative study of persistence mechanisms for the java platform. Technical Report TR-2004-136, Sun Microsystems Laboratories Technical Report, 2004.

[81] Alan Curtis Kay. Source: The wiki way: Quick collaboration on the web, bo leuf, ward cunningham. http://en.wikiquote.org/wiki/Alan_Kay, 2001. Last visited: 6.08.2012.

[82] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *ICSE*, pages 151–160. ACM, 2011.

[83] Andrew Ko, Brad Myers, and Htet H. Aung. Six learning barriers in end-user programming systems. In *Proceedings of Visual Languages and Human Centric Computing*, pages 199–206, 2004.

[84] Sven-Eric Lautemann. An introduction to schema versioning in oodbms. In *DEXA Workshop*, pages 132–139, 1996.

[85] Gary T. Leavens, Baker Albert L., and Ruby Clyde. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[86] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.*, 25(1):83–127, 2000.

[87] Bertrand Meyer Marco Piccioni, Manuel Oriol. Schema evolution data for java and eiffel. `http://tinyurl.com/ESCHER-data`, 2012. Last visited: 25.09.2012.

[88] Digital Mars. Contract programming. http://dlang.org/dbc.html, 2012.

[89] B. Meyer. *Object Oriented Software Construction*. Prentice Hall PTR, 2 edition, 1997.

[90] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[91] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[92] Bertrand Meyer. Conversions in an object-oriented language with inheritance. *JOOP (Journal of Object-Oriented Programming)*, 13(9):28–31, 2001.

[93] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In *ECOOP*, pages 1–32, 2005.

[94] P. Milne. Using xmlencoder. `http://www.oracle.com/technetwork/java/persistence4-140124.html`. Last visited: 5.10.2012.

[95] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, pages 122–133, 1998.

[96] Simon R. Monk and Ian Sommerville. Schema evolution in oodbs using class versioning. *SIGMOD Record*, 22(3):16–22, 1993.

[97] Iulian Neamtiu, Michael W. Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *PLDI*, pages 72–83, 2006.

[98] Martin Odersky. Scala docs: The option class. `http://www.scala-lang.org/api/current/scala/Option.html`, 2012. Last visited: 10.09.2012.

[99] University of Maryland. Checkfornull annotation in findbugs static analysis tool. `http://findbugs.sourceforge.net/manual/annotations.html`, 2012. Last visited: 10.09.2012.

[100] University of Maryland. Findbugs static analysis tool. http://findbugs.sourceforge.net/manual/annotations.html, 2012. Last visited: 10.09.2012.

[101] Chair of Software Engineering ETH Zurich. Introduction to programming course web page. http://se.inf.ethz.ch/, 2012. Last visited: 9.10.2012.

[102] Oracle. Java serialization specification. http://download.oracle.com/javase/6/docs/platform/serialization/spec/version.html#6519. Last visited: 22.07.2012.

[103] Oracle. Edition-based redefinition. http://www.oracle.com/technetwork/database/features/availability/edition-based-redefinition-1-133045.pdf, 2009. Last visited: 22.07.2012.

[104] J. Paterson, S. Edlich, H. Hörning, and R. Hörning. *The Definitive Guide to db4o*. Apress, 2006.

[105] Marco Piccioni. Invariant evolution data for eiffel: an exploratory study. http://tinyurl.com/cmnjfyh, 2012. Last visited: 13.11.2012.

[106] Marco Piccioni, Manuel Oriol, and Bertrand Meyer. Ide-integrated support for schema evolution in object-oriented applications. In *RAM-SE*, pages 27–36, 2007.

[107] Tom Preston-Werner. Semantic versioning 2.0. http://semver.org/, 2012.

[108] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[109] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):26–34, 2009.

[110] Roman Schmocker. A better eiffelstore library. ETH Bachelor Thesis, 2012.

[111] Teseo Schneider. Escher: Eiffel schema evolution support. ETH Bachelor Thesis, 2008.

[112] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA*, pages 483–495, 1986.

[113] DbVis Software. Dbvisualizer. `http://www.dbvis.com/`, 2012. Last visited: 19.09.2012.

[114] Eiffel Software. Eiffelstore data access library. http://www.eiffel.com/libraries/store.html, 2012.

[115] Eiffel Software. Eiffelstudio svn repository. `https://svn.eiffel.com/eiffelstudio/`, 2012. Last visited: 31.08.2012.

[116] TIOBE Software. Tiobe index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`, 2012. Last visited: 4.08.2012.

[117] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *29th International Conference in Software Engineering*, pages 529–539, 2007.

[118] Jeffrey Stylos and Brad A. Myers. The implications of method placement on api learnability. In *16th International Symposium on Foundations of Software Engineering*, pages 105–112, 2008.

[119] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP*, pages 600–624, 2007.

[120] Michael Würsch, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald Gall. Seon: A pyramid of ontologies for software evolution and its applications. *Computing*, pages 1–31, 2012.

[121] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *ICST*, pages 220 –229, 2008.

CURRICULUM VITAE

General information

Name: Marco Piccioni
Date of birth: July 7, 1965
Nationality: Italian
Web page: `http://se.inf.ethz.ch/people/piccioni`
E-mail address: marco.piccioni@inf.ethz.ch

Education and career history

- October 2006 - December 2012: Research Assistant and, from September 2008, PhD student at the Chair of Software Engineering, ETH Zurich (Swiss Federal Institute of Technology Zurich)
- July 1996 - September 2006: Technical Trainer and Software Engineer at Sistemi Informativi S.p.A. (an IBM company)

Certificates and diplomas

- Master degree in Economics at Università L. Bocconi, Milano (1994)
- Laurea degree in Mathematics at Università degli Studi di Roma La Sapienza (1993)
- Cambridge Certificate of Proficiency in English (1998)

Publications

- **Piccioni, M.**, Oriol, M., Meyer, B.: "Class Schema Evolution for Persistent Object-Oriented Software: Model, Empirical Study, and Automated Support", accepted for publication by IEEE journal: Transactions on Software Engineering (TSE), November 2011.

- **Piccioni, M.**, Oriol, M., Meyer, B., Schneider T.: “An IDE-based, integrated solution to Schema Evolution of Object-Oriented Software”, ASE 2009: 24th IEEE/ACM International Conference on Automatic Software Engineering, (Auckland, New Zealand), November 2009.
- Jin R., **Piccioni, M.**: “Eiffel for .NET Binding for db4o”, ICOODB 2008: First International Conference on Object Databases, (Berlin, Germany), March 2008.
- **Piccioni, M.**, Meyer, B.: “The Allure and Risks of a Deployable Software Engineering Project: Experiences with Both Local and Distributed Development”, Proceedings of CSSE&T 2008: 21st IEEE-CS Conference on Software Engineering Education and Training 2008, (Charleston, South Carolina, USA), April 2008.
- **Piccioni, M.**, Oriol, M., Meyer, B.: “IDE-integrated Support for Schema Evolution in Object-Oriented Applications”, Proceedings of RAM-SE 2007: 4th ECOOP’2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution, (Berlin, Germany), August 2007.

Languages

- Italian — native
- English — advanced (working language)
- German — basic