Faculty of Sciences
Department of Computer Science and Applied Computer Science

Reasoning about first-class changes for support in software evolution

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Applied Computer Science by

Bart Depoortere

2006-2007

Promoter: Prof. Dr. Theo D'Hondt
Advisors: Peter Ebraert, Jorge Vallejos

Abstract

Software systems continuously need to evolve for coping with changing requirements. This leads to a degeneration of their architecture making them less maintainable and understandable. Software evolution research tries to tackle this issue by studying why and how software evolves. This is done by analyzing the history of a software system and by reasoning about it. Current support for software evolution research requires the reconstruction of a (partial) system history based on the artifacts stored at the repository of change management systems. Several approaches for storing those artifacts encounter different problems with regard to software evolution such as information degradation, information loss or unordered information.

The goal of this thesis is to support software evolution researchers in reasoning about software evolution. That support is provided by offering accurate evolutionary information about software applications. The information is acquired by incrementally capturing every applied change as a first-class object which contains all the information that specifies the change. Each change object contains fine-grained information about what it represents, why it exists, when it applies, where it applies and how it applies. The system history maintains references to those change objects, and as such, represents a complete evolutionary history of the software system without degraded, lost or unordered information.

Acknowledgements

First of all, I would like to thank Prof. Dr. Theo D'Hondt for giving me the opportunity and the means to do research. A special thanks goes to my supervisors Peter Ebraert and Jorge Vallejos who continuously steered me in the right direction. I deeply appreciate their involvement and dedication. This dissertation could not have been written without their guidance and comments on preliminary versions. I would also like to thank the people of PROG for taking their time to hear me and for commenting on and suggesting improvements to my methodology.

Thanks to my girlfriend Lieselotte Baekelandt for her great support, endurance and encouragement during the good and bad days of the past three years. I would also like to thank my parents, Rita Houthoofd and Johny Depoortere, for giving me the chance of going to university. I also wish to thank my brother Stijn Depoortere and my family for their support during my education. Another thank you goes to the parents and family of my girlfriend for their hospitality when taking a day off. Finally, I wish to thank all my friends for their kind support and I would like to dedicate this dissertation to two deceased friends of mine, Dries Ostyn and Mario Deferme.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Solution . . . 3
- 1.3 Roadmap . . . 4

2 Change management for support in software evolution **5**
- 2.1 Introduction . . . 5
- 2.2 Criteria for change management systems . . . 6
- 2.3 Change management systems . . . 8
  - 2.3.1 Time dimension . . . 8
  - 2.3.2 Structural dimension . . . 10
  - 2.3.3 Combinations of approaches . . . 14
- 2.4 Problem statement . . . 20
- 2.5 Goal . . . 21
- 2.6 Conclusion . . . 22

3 A meta-model for support in software evolution **24**
- 3.1 Introduction . . . 24
- 3.2 Criteria for meta-models . . . 25
- 3.3 Alternatives . . . 26
  - 3.3.1 Modeling language based alternatives . . . 26
  - 3.3.2 Graph based alternative . . . 30
  - 3.3.3 Comparison of alternatives . . . 31
- 3.4 FAMIX definition . . . 31
  - 3.4.1 Basic data types . . . 31
  - 3.4.2 Object . . . 32
  - 3.4.3 Entity . . . 33
  - 3.4.4 Association . . . 36
  - 3.4.5 Argument . . . 37
- 3.5 FAMIX extensions . . . 38

3.5.1 Extension for dynamic state 39
3.5.2 Extension for Smalltalk 40
3.6 Conclusion 43

4 First-class changes for support in software evolution 45
4.1 Introduction 45
4.2 Properties of first-class changes 46
4.2.1 Detailed information 46
4.2.2 Order preservation 47
4.2.3 Language independent representation 47
4.2.4 Extensibility 47
4.2.5 Abstraction 48
4.3 Design of first-class changes 49
4.3.1 Change 49
4.3.2 EntityChange 51
4.4 Applications for support in software evolution 54
4.4.1 Program generation 54
4.4.2 Merging and conflict detection 55
4.4.3 Program exploration 56
4.4.4 Debugging 57
4.5 Conclusion 58

5 Implementation 60
5.1 Introduction 60
5.2 Environment 60
5.2.1 Smalltalk 61
5.2.2 VisualWorks for Smalltalk 62
5.3 First-class changes in Smalltalk 62
5.3.1 Change classes 62
5.3.2 Preconditions 63
5.4 Change management 64
5.4.1 ChangeLogger 64
5.4.2 Creation of change objects 65
5.4.3 Logging of changes 65
5.4.4 Logging of changes at statement level 68
5.4.5 Application of changes 71
5.4.6 Undoing of changes 72
5.5 Conclusion 73

6 Validation 75
6.1 Introduction 75

6.2 Environment . . . 75
6.3 Intension queries . . . 76
6.3.1 Basic queries . . . 76
6.3.2 Advanced queries . . . 78
6.4 Case study: HotDraw . . . 79
6.5 Reasoning . . . 81
6.5.1 Basic reasoning . . . 81
6.5.2 Advanced reasoning . . . 83
6.6 Conclusion . . . 86

7 Conclusion and future work 88
7.1 Conclusion . . . 88
7.2 Future work . . . 90

A Invariants for the FAMIX meta-model and its extensions 96
A.1 FAMIX meta-model . . . 96
A.2 Extension for dynamic state . . . 100
A.3 Extension for Smalltalk . . . 100

B Design of first-class changes 102
B.1 Base model . . . 102
B.1.1 Change . . . 102
B.1.2 EntityChange . . . 104
B.1.3 AssociationChange . . . 119
B.1.4 ArgumentChange . . . 126
B.2 Extensions . . . 127
B.2.1 Extension for dynamic state . . . 127
B.2.2 Extension for Smalltalk . . . 132

List of Figures

2.1 Snapshot-based vs Incremental approach 10
2.2 File-based reasoning 11
2.3 File-based approach - Degraded information 12
2.4 File- vs Entity-based approach 14
2.5 Snapshot- and file-based change management systems 14
2.6 Snapshot- and entity-based change management systems 15
2.7 Incremental and file-based change management systems 16
2.8 Incremental and entity-based change management systems 17
2.9 Program state representation (taken from [44]) 19
2.10 Comparison of change management systems 20

3.1 UML tagged values - Example 27
3.2 UML stereotypes - Example 27
3.3 UML constraints - Example 27
3.4 Conception of the FAMIX model (based on [10]) 29
3.5 Comparison of alternatives 31
3.6 FAMIX model - Object 32
3.7 FAMIX model - Entity 33
3.8 FAMIX model - BehaviouralEntity 34
3.9 FAMIX model - StructuralEntity 35
3.10 FAMIX model - Association 37
3.11 FAMIX model - Argument 38
3.12 Dynamic extension - Instance 39
3.13 Dynamic extension - Global variable 40
3.14 Smalltalk extension - Class 40
3.15 Smalltalk extension - Behavioral entity 41
3.16 Smalltalk extension - Structural entity 42
3.17 Smalltalk extension - Inheritance definition 43

4.1 Language independent representation of first-class changes 48
4.2 Abstracting changes 49

4.3 Design - Change class 50
4.4 Change hierarchy - EntityChange 51
4.5 Change hierarchy - ClassChange 52
4.6 Application - Program generation 55
4.7 Application - Merging and conflict detection 56
4.8 Application - Pull Up Method 58
4.9 Application - Debugging 58

5.1 Change class 62
5.2 Singleton Design Pattern - ChangeLogger 64
5.3 Example - Parse tree 69
5.4 Visitor Design Pattern - ProgramNodeVisitor 70

6.1 SOUL Query Browser - An extract of the results for change(?CH) 77

B.1 Design - Change 102
B.2 Design - EntityChange 104
B.3 Design - PackageChange 105
B.4 Design - ClassChange 106
B.5 Design - BehaviouralEntityChange 108
B.6 Design - MethodChange 110
B.7 Design - FunctionChange 111
B.8 Design - StructuralEntityChange 113
B.9 Design - AttributeChange 113
B.10 Design - GlobalVariableChange 115
B.11 Design - FormalParameterChange 117
B.12 Design - LocalVariableChange 118
B.13 Design - AssociationChange 119
B.14 Design - InheritanceDefinitionChange 120
B.15 Design - InvocationChange 122
B.16 Design - AccessChange 124
B.17 Design - ArgumentChange 126
B.18 Design - InstanceChange 128
B.19 Design - AttributeValueChange 129
B.20 Design - GlobalVariableValueChange 131
B.21 Design - AddClass 132
B.22 Design - BehaviouralEntityChange 133
B.23 Design - StructuralEntityChange 134
B.24 Design - AddAttribute 134
B.25 Design - ModifyInheritanceDefinition 135

Chapter 1

Introduction

Evolution is a widespread phenomenon present in a broad range of domains (e.g. biology or philosophy). But what is evolution? In [29], evolution is defined as

> "a process of progressive, for example beneficial, change in the attributes of the evolving entity or in one if its constituent elements."

A lot of researchers have stated their view on how evolution could be interpreted in the context of software. Mittermeir for example, believes that *software evolution* should only be considered as the changes that were applied to a software system [38]. In [29], Lehman studies different interpretations of software evolution and elaborates on the theory and practice behind it. He defines software evolution as

> "the dynamic behavior of programming systems as they are maintained and enhanced over their life time." [26, 47]

A software system is *continuously being changed* to cope with a changing environment, a higher demand of functionality or the introduction of new hardware [1, 3]. These changes lead to a more complex structure of the system [31]. Thus when the software is evolving, its architecture tends to degenerate [23, 31], making the software less *maintainable* and less *understandable*. Developers however want to adapt software without loosing on understandability and maintainability. Software evolution researchers try to support this by using "the history of a system to analyze its present state and to predict its future development" [43]. The focus is on studying *how* and *why* the software evolves over time. Analyzing the evolution of software systems provides useful information for a variety of activities (e.g. software maintenance or reverse engineering) [44]. Some benefits gained by studying the evolution of software systems are listed below:

- *Improved program understandability*: Robbes states that "recording the history of a system allows reconstructing the original design intentions of the developers" [42]. This gives a better insight in how and why a program evolves making it easier to understand the purpose of a (sub)system. By increasing the *program understandability*, the *ease of maintaining* a system is also being increased which in its turn encourages *component reuse*.

- *Improved development process(es)*: a better understanding of how software evolves, benefits all phases of the software development process [28, 30, 47]. Lehman and Ramil state that "insight into software evolution indicates into the types of activities, methods and tools required, which are likely to be most beneficial, when and how they should be used and how they relate to one another" [30]. For example, developers can take maintenance into account during the design phase of a software system by designing that system in a way that facilitates future adjustments [47].

- *Reduced maintenance costs*: the majority of resources used to evolve a software system is spent after its first release [27, 47]. Often software systems need to be adapted due to new requirements set out by the management or customers. A better understanding of how and why a program evolves helps *anticipating problems* reducing the effort (and cost) of software engineers to solve those problems at later stages in the development process [47].

- *General business benefits*: business nowadays become more and more dependent on the deployed software systems. Delayed applications, functionalities and bug fixes for example may endanger the business operations and the generated profits. Improved program understandability and development processes help anticipating problems *reducing delayed activities* which influence positively the business and its operations (e.g. quality of products or economic benefits) [28, 30].

1.1 Motivation

In order to gain insights into the evolution of software systems, researchers require *evolutionary information* [43]. They use tools that analyze that information and present them the results. One way to gather evolutionary information is by inspecting and analyzing the artifacts stored at the *repository* of *change management systems*. A repository is a centralized library of artifacts maintained by the change management system (e.g. files containing source code). Change management systems are used for different reasons such as *storing changes* of evolving

software systems, *sharing source code* with colleagues or *backing up source code*. Another approach for acquiring the evolutionary information is by using *clone detection* [48]. This is a well known visualization technique that detects *mismatches*, expressing *changes*, between two or more releases of a software system. There are many other techniques for gathering evolutionary information (e.g. *origin analysis* [18]) of which a small overview is given in [17].

This research work focusses on the first technique where evolutionary information is extracted from the repositories managed by the change management systems. Using change management systems for extracting that kind of information may lead to different problems such as *degraded information*, *information loss* or *unordered information*.

1.2 Solution

The encountered shortcomings of the change management systems are tackled by introducing an *incremental* and *entity-based* change management system. Changes are captured *as they happen* (incremental) and each change stores information about the *program entity* it affects (entity-based). Program entities refer to the "building blocks" of a programming language which are available to the developers for implementing a system (e.g. a class or method).

A *meta-model* for programming languages describes their available building blocks. *Types of changes* are derived from and classified based on the specification of such a meta-model. Changes themselves are expressed as *first-class objects* which are further referred to as *first-class changes*. First-class changes may be stored in a variable or data structure, may be passed as an argument to a function and may be returned as the value of a function.

By providing a *language independent meta-model*, a *language independent classification of first-class changes* is obtained. That classification can for example be used as an intermediary format for exchanging and studying the evolution of programs implemented in *different programming languages*. This dissertation supports the claim that:

An incremental and entity-based change management system with first-class changes based on a meta-model offers accurate evolutionary information and supports reasoning about software evolution.

1.3 Roadmap

This dissertation starts with an overview of several approaches for storing and managing software architectural artifacts at a central repository accessible by software evolution tools. Their benefits and shortcomings regarding extraction of evolutionary information are discussed in detail in Chapter 2.

Chapter 3 seeks a meta-model capable of modeling object-oriented software at source-code level and valid as a language independent taxonomy of program entities. This taxonomy determines what kind of evolutionary information (e.g. what program entities can be changed) is maintained by the new change management system and thus also what information is available for the analysis tools.

Chapter 4 derives types of changes from the meta-model found in Chapter 3 and classifies them into a class hierarchy. Furthermore, Chapter 4 takes a closer look at applications that may benefit from using first-class changes.

Next, in Chapter 5 we discuss the implementation of the first-class changes and their management (e.g. the creation of changes or the application and undoing of changes).

In Chapter 6, the incremental and entity-based approach of the implemented change management system is examined by experimenting with a real evolving case. We validate if the adopted approach does offer *accurate and useful information* and if the available information is suitable for reasoning about the evolution of the monitored case.

We conclude by summarizing our work and by stating possible future work in Chapter 7.

Chapter 2

Change management for support in software evolution

2.1 Introduction

Software evolution researchers require accurate information for reasoning about software systems and their evolution [43]. Most software evolution research tools recover (extract) that information from the *change management system* used by the developers to store changes of evolving software systems. In general, change management systems offer some important benefits to its users [42, 51]. We now list some of these benefits:

- *Efficient sharing of a project*: each involved party has access to the repository in order to retrieve or publish the desired changes. Change management systems allow efficient sharing of a project and thus encourage software development in team.
- *Recovery from user mistakes or system corruption*: users make mistakes such as deleting or erroneously modifying files. Or some files containing source code become corrupted due to a virus, system crash, ... Change management systems retain changes of a particular system. Hence the user has the possibility to *correct the system* at his disposal by backtracking to a previous state of that system. This usually requires some a priori knowledge about the nature of the mistake to locate the artifacts that should be recovered.
- *Analysis of system history*: change management systems record *meta-data* such as the timestamp of storing some artifact or the user who stored it. This

data can help answering questions about *how* a software system reached a certain state.

A high quality change management system is thus needed in order to provide accurate reasoning information. Therefore the following criteria are set out to assist the search for such a change management system: *complete information*, *language independence for monitoring changes and reasoning*, *developer independence*, *order preservation* and *hooks for extensibility for monitoring changes and reasoning*. These criteria are explained in Section 2.2.

Section 2.3 starts with a classification of change management approaches according to a *time dimension* (when are changes stored: *snapshot-based* vs *incremental*) and a *structural dimension* (how are changes stored and managed: *file-* vs *entity-based*). It also gives an overview of four types of change management systems and some examples. The following types are discussed: *snapshot- and file-based system*, *snapshot- and entity-based system*, *incremental and file-based system* and *incremental and entity-based system*.

The last two sections respectively deal with the establishment of a problem statement followed by the goal of this work which reveals a small hint for solving the stated problem.

2.2 Criteria for change management systems

Different criteria are set out in order to find an adequate change management system with respect to reasoning about software evolution.

- *Complete information*: if all desired information is available but with a lower quality than it should be, the change management system suffers from *degraded information*. If that information is not even available, the change management system suffers from *information loss*. Stored changes must thus ensure that all desired information is *available* and of *high quality* i.e. correct and accurate. It is also feasible that the information is directly available thus without the change management system having to do extra effort (e.g. parsing). This is what we refer to as *complete information*.
- *Language independent monitoring and reasoning*: on the one hand developers want to store applications implemented in different programming languages at one central place using one single change management system. The goal of most change management systems is to support as many programming languages as possible which reduces the amount of effort of de-

velopers to tune those change management systems towards specific languages. This is further referred to as *language independent monitoring*. On the other hand software evolution researchers want to study and compare those cases using that same change management system. To accomplish that, researchers have to put in a huge amount of effort to retrieve language specific change information (e.g. writing a plug-in that extracts evolutionary information based on language specific knowledge). Thus they want to use change management systems without adapting them for each used programming language. This is what we refer to as *language independent reasoning*.

- *Developer independence*: mostly, developers themselves are responsible for storing changes during the evolution of a software system running the risk of neglecting that task. Therefore it would be useful that the change management system allows storing changes without asking for the explicit request of developers. This is further referred to as *developer independence*.

- *Order preservation*: an important issue in software evolution research is the sequence in which changes were applied. Software evolution researchers using a change management system that does not preserve that order loose a lot of important information (e.g. when was that change applied? which changes were applied before it?). Therefore it is of great benefit that the change management system contains correct and detailed *time information* for each change allowing to reconstruct a change history in the exact order. This is what we refer to as *order preservation*.

- *Extensibility hooks for monitoring and reasoning*: often software evolution researchers do not dispose of enough artifacts to represent the information they need or to extract data from. Therefore it would be useful that the change management system's available spectrum of changes can be extended to monitor more types of changes. This is further referred to as *extensibility hooks for monitoring*. It is also useful to extend that spectrum to reason about more information to suit additional needs of software evolution researchers. This is further referred to as *extensibility hooks for reasoning*.

2.3 Change management systems

Change management systems are used by developers to store changes of evolving software systems. Those changes are managed at a central repository where each involved party has access to in order to retrieve or publish the desired changes. Change management systems contain thus a lot of useful information about the evolution of the managed software systems (e.g. created classes during the lifetime of a system) hence they are a valuable source for software evolution researchers. The first two subsections classify change management approaches according to the dimensions *time* (when are changes stored) and *structure* (how are changes stored). The third subsection discusses the change management systems quoted in that classification.

2.3.1 Time dimension

This section discusses two approaches which differ in *when* changes are stored at the repository. The *snapshot-based approach* records the evolutionary information at the *explicit request* of developers [44]. The *incremental approach* however *implicitly* records the evolutionary information: changes are recorded as they happen. The following two subsections respectively elaborate on the snapshot-based and incremental approaches.

2.3.1.1 Snapshot-based approach

In this setting, developers *explicitly store changes* to software systems at the central repository of their change management system. Hence, each developer carries the responsibility of storing these changes on a regular basis, called *commits*. Each commit represents a *snapshot* of the software system (or *version*) hence the change management system disposes of successive software versions. This implies that the actual modifications between two committed versions are never stored in the repository resulting in *degraded information* or even *loss of information* [43, 45]. Reasons for this are:

- *Explicit commits* depend on the programmers' will implying that several independent features or bug fixes can be introduced by one single commit. Off course this makes it harder to distinguish between the different changes.
- Let us assume that all changes between two successive software versions are detected and acquired correctly. Even then there would still be information loss because the time information of each change is restricted to the

time information of the corresponding commit. This makes it *impossible to obtain the exact order* in which the detected differences were originally applied.

It is *very hard to extend* the available spectrum of monitored changes: everything is based on comparisons between successive snapshots running the risk of having degraded or lost information. It is not useful to extend the available change types to obtain more information while recovering incomplete information.

CVS, *SVN* and *StORE* are examples of change management systems using the snapshot-based approach. CVS and SVN are explored in Section 2.3.3.1, StORE is explained in Section 2.3.3.2.

2.3.1.2 Incremental approach

In this setting, changes to software systems are *implicitly* stored at the central repository of the used change management system. Such change management system is integrated into the *Interactive Development Environment* (IDE) and continuously captures changes as they are performed by the developer(s). Hence developers do not carry the responsibility of committing versions. Each captured change increments the system history stored in the repository enabling to reconstruct a software version at any point. All modifications to the software system are captured and stored at the repository resulting in *complete evolutionary information* [12, 43, 45]:

- IDE integration enables reacting to events as they happen and thus provides *complete information*. Timestamps for example are now detected at change level instead of being reduced to the time of committing a software version.
- Each change is introduced by a separate commit enabling easier distinction of independent features or bug fixes.
- Changes are sequentially captured in the *order they occur* and there is *no loss of time information* since each change has its own commit time.

The IDE integration allows *easy extension* of the available spectrum of changes. Developers have the possibility to provide more information to the available changes which can be queried via the change management system.

AJC Active Backup, *FileHamster*, *ChangeList* and *SpyWare* are examples of change management systems using the incremental approach. AJC Active Backup and FileHamster are discussed in Section 2.3.3.3, ChangeList and SpyWare are explored in Section 2.3.3.4.

2.3.1.3 Comparison

Criteria/Type	S	I
Complete information	✗	✓
Developer independent	✗	✓
Order preservation	✗	✓
Extensibility hooks		
- Monitoring	✗	✓
- Reasoning	✗	✓

Figure 2.1: Snapshot-based vs Incremental approach

Figure 2.1 depicts an overview in which the stated criteria are compared against the snapshot-based (S) and incremental (I) approaches. An “X” indicates that the criterium is badly or not supported by the change management system while a “V” indicates that there are no problems. Note that the criteria regarding *language independence* are not recorded in the figure because they do not apply to the *time of committing* (when are changes stored). The figure shows that the incremental approach scores positively for all noted criteria while the snapshot-based approach scores negatively for the same criteria. This reveals that change management systems must strive to use the incremental approach instead of the snapshot-based one.

2.3.2 Structural dimension

This section discusses two approaches differing in *how* changes are structured and managed. The *file-based approach* stores and manages changes as textual files while the *entity-based approach* treats changes as changes to program entities. The following two subsections respectively elaborate on the file- and entity-based approaches.

2.3.2.1 File-based approach

Most change management systems aim to support as many programming languages as possible. Functioning at file-level supports *language independent monitoring* of changes: it supports all languages contained within the stored textual files. Figure 2.2 shows the conceptual view of how current *reasoning* algorithms

base themselves on comparing files and lines of two subsequent software versions. As depicted in the figure, each version is represented by a collection of stored files. On the left hand side we see version V_n of a particular software system which consists of some files containing source code. On the right hand side we see version V_{n+1} of that same system which contains in this case less files then version V_n. Extracting information about their evolution involves comparing files and lines of code of V_n and V_{n+1} thus finding the differences between them (denoted by ΔV). The extraction mechanism requires knowledge about the programming language in which the software under study was developed. These algorithms need to reconstruct a (partial) system history before reasoning about the system's evolution.

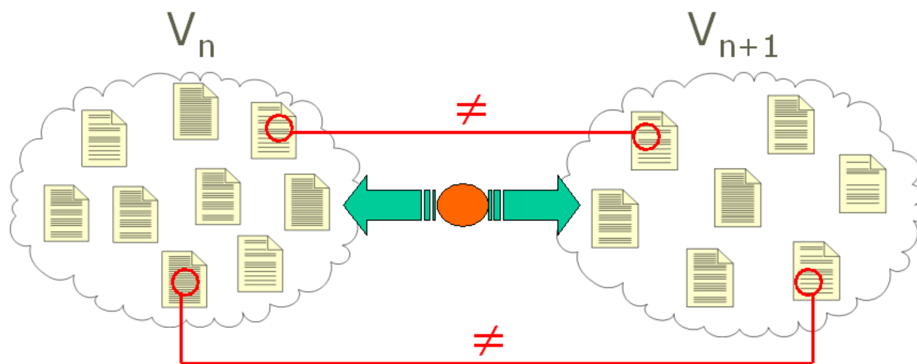

Figure 2.2: File-based reasoning

Reducing changes to the level of textual files and lines implies several problems when reasoning about software [43, 45]:

- *Degradation of information*: the information of the program is spread across multiple files and has no meaning at file-level. In order to recover that information, extra tools are necessary for deriving the (correct) program structure. A small example is used to clarify this problem.
  Figure 2.3 depicts the recovery of the inheritance chain of some class denoted in the Java programming language. On the left hand side, we see four textual files, each one representing a class. Three of them play a role in an inheritance chain of which `ClassA` is the root class, `ClassB` is the middle one and `ClassC` is the deepest subclass. On the right hand side, we see a parser developed to reason about the source code files. The recovery of the inheritance chain involves three steps. Firstly, the parser loads a file and parses it. After loading a file, the parser extracts the name of the superclass as well as the location of the source file containing the definition of that

class. Finally before loading another file, the parser must adjust the answer to the desired query. The same steps are executed to obtain the specified subclasses in the desired inheritance chain.

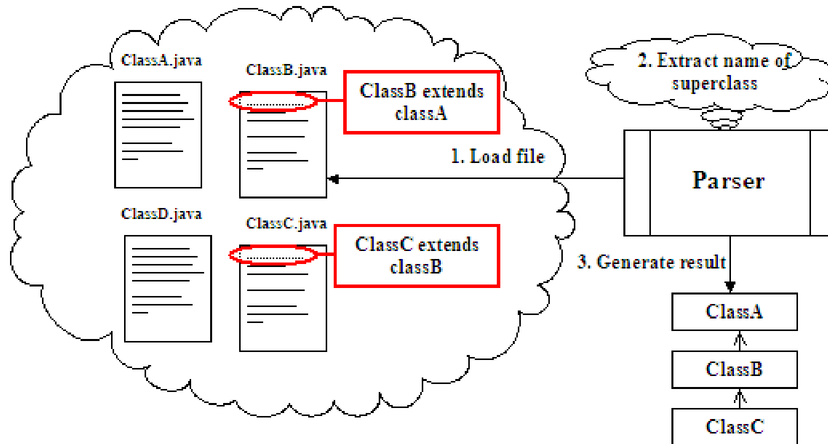

Figure 2.3: File-based approach - Degraded information

- *Complex parsing technology*: building a parser demands a lot of resources which are not always available (e.g. time or effort). Hence it is unfeasible to equip software analysis tools with a parser for each programming language the tool wants to support.
- *Tracking of program entities* among the evolutionary information stored in the repository is difficult since program entities can be *moved* or *renamed*. Software evolution analysis tools have to take into account such events when parsing files contained in the repository.

File-based systems have the possibility of storing time information for each change within the files and thus maintain the *exact order* in which the changes were originally applied. Everything is based on textual comparisons and knowledge about the implementation language of the studied system which leads to degraded or lost information. It is *not useful to extend* the available spectrum of changes based on that information.

CVS, *SVN*, *AJC Active Backup* and *FileHamster* are examples of change management systems using the file-based approach. CVS and SVN are explored in Section 2.3.3.1, AJC Active Backup and FileHamster are discussed in Section 2.3.3.3.

2.3.2.2 Entity-based approach

Entity-based systems store information about changed program entities (e.g. a package or class) [42]. Functioning at entity-level takes away *language independent monitoring* of changes but offers the possibility of *language independent reasoning* by specifying a standard format for exchanging and storing change information. The entity-based approach provides *complete evolutionary information* and solves the encountered problems of the file-based approach [42]:

- The file-based approach needs to *reconstruct program entities* based on the stored files running the risk of loosing information. The entity-based approach automatically provides all information needed to reconstruct program entities which is easier to study. There is *no loss of information* since program entities have a fine-grained granularity (e.g. a class or method).
- Each program entity can be queried directly for its information *reducing extra effort* needed to recover that information (e.g. parsing technology).
- The entity-based approach allows easier *tracking of program entities* among the evolutionary information stored in the repository. Program entities can be compared at a higher-level taking into account a lot more information than file-based systems can.

Entity-based systems have the possibility of storing detailed time information for each change allowing to maintain the *exact order* in which the changes were originally applied. Due to the entity-based nature, the available spectrum of changes can easily be *extended*.

StORE, *ChangeList* and *SpyWare* are examples of change management systems using the entity-based approach. StORE is explored in Section 2.3.3.2, SpyWare and ChangeList are discussed in Section 2.3.3.4.

2.3.2.3 Comparison

Figure 2.4 shows an overview in which the stated criteria are compared against the file- (F) and entity-based (E) approaches. An “X” indicates that the criterium is badly or not supported by the change management system while a “V” indicates that there are no problems. Note that the criterium regarding *developer independence* is not recorded in the figure because it does not apply to the *structure* of changes (how are changes stored). The figure reveals that the entity-based approach scores positively for almost all noted criteria. Only the *language independent monitoring* criterium is not met which is acceptable because language

Criteria / Type	F	E
Complete information	✗	✓
Language independent		
- Monitoring	✓	✗
- Reasoning	✗	✓
Order preservation	✓	✓
Extensibility hooks		
- Monitoring	✗	✓
- Reasoning	✗	✓

Figure 2.4: File- vs Entity-based approach

independent monitoring is accomplished by storing changes at file-level. But Figure 2.4 reveals that the file-based approach scores badly by satisfying only two criteria. This implies that change management systems must strive to use the entity-based approach instead of the file-based one.

2.3.3 Combinations of approaches

This section gives an overview of four kinds of change management systems where each type is based on a combination of two approaches discussed in the previous sections (of each discussed dimension one approach). Each type is briefly explored by explaining how it works and by giving one or more existing examples. The *snapshot- and file-based* (S/F), *snapshot- and entity-based* (S/E), *incremental and file-based* (I/F) and *incremental and entity-based* (I/E) systems are discussed in the following four subsections.

2.3.3.1 Snapshot- and file-based system

	File based	Entity based
Snapshot based	CVS, SVN	
Incremental based		

Figure 2.5: Snapshot- and file-based change management systems

This is the traditional combination for change management systems which are better known as *version control systems*. Version control is the management of

multiple revisions of the same unit of information, for instance source code of applications [13]. Due to its snapshot- and file-based nature developers commit source files at a certain point in time which are then versioned in the central repository of the versioning system. In its most simple form each version is represented by a collection of files possibly categorized into a directory structure.

Figure 2.5 highlights the square with the traditional version control systems. There are a lot of version control systems available on the software market. Just querying a web browser for one of the following terms: *version control system*, *revision control system*, *source control system* or *(source) code management system* results in a large list of web pages referring to version control systems. The following two paragraphs discuss *CVS* and *SVN*, two well known and widely used versioning systems.

CVS *Concurrent Versions System* (CVS) is a version control system that enables the recording of the history of source files and documents. CVS uses a client-server architecture and allows multiple connections from different locations. It is developed to organize and maintain a collection of source files which are stored at the explicit request of team members. Instead of storing each committed file separately, CVS uses an optimized technique. It stores all the committed versions of a file in one single file by only containing the differences between them (ΔV). For more information about CVS we refer the reader to [7].

SVN *SVN* or Subversion is an advanced, open source version control system. Its main goal is to help you track the changes to directories of files under version control. Subversion also uses a client-server architecture allowing multiple connections from different locations. Developers commit revisions whenever they feel like it (or when obliged). Each revision has its own root which is used to access its contents. Subversion maintains for each file a reference to its most recent version. For more information about Subversion we refer the reader to [8].

2.3.3.2 Snapshot- and entity-based system

	File based	Entity based
Snapshot based	CVS, SVN	StORE
Incremental based		

Figure 2.6: Snapshot- and entity-based change management systems

Version control systems retain source-code files but they can version program entities of the software system under development instead. The principle of revision control remains, only the way how changes are stored and accessed differs from the approach discussed in the previous section. Developers still have to commit source code but now in the form of program entities (e.g. a class or method) which are then versioned in a repository and managed by the change management system. This implies that the change management system must have knowledge about the implementation language of the versioned software system. Conceptually each version is represented by a collection of program entities possibly categorized into a structure similar to that of the versioned program.

Figure 2.6 focusses on the snapshot-based approach combined with the entity-based one. The number of available entity-based versioning systems on the software market is restricted but the *StORE* system is one of them. StORE is discussed in the following paragraph.

StORE *StORE* is the version control system used by the VisualWorks for Smalltalk environment [52]. It is based on a client-server architecture: it uses a centralized server with a database acting as the central repository. Developers have the possibility to publish (commit) packages which will be versioned by StORE. Instead of versioning files, StORE works on a granularity-level of program entities (e.g. a class or method) which facilitates for example the merging of source code of different developers.

2.3.3.3 Incremental and file-based system

	File based	Entity based
Snapshot based	CVS, SVN	StORE
Incremental based	AJC Active Backup FileHamster	

Figure 2.7: Incremental and file-based change management systems

Changes can be committed without having to wait for the explicit request of developers (commit). The IDE integration allows continuously capturing changes and committing the source files containing the changed entities resulting in a complete system history. Thus change management systems using this approach retain source-code files and act as an integrated version control system restricting the gaps between two successive software systems to the minimum. Conceptually the

system history can be seen as a collection of files where each file is timestamped allowing to reconstruct the original order.

Figure 2.8 highlights the square with the incremental and file-based change management systems. There are a small number of available systems combining the incremental and file-based approaches. The *AJC Active Backup* and *FileHamster* are such systems and are explored in the following two paragraphs.

AJC Active Backup *AJC Active Backup* is an automatic revision control system that continuously monitors changed files [50]. The user may configure which folders and files are monitored. Every time changes are saved to a monitored file, it is revised in a compact archive that acts as a local repository. The incremental approach implemented by this tool results in a complete record of what the user has been doing. AJC Active Backup also offers the possibility of comparing monitored files and showing the differences between them (*diffing*). Instead of storing each changed file separately, AJC Active Backup compresses the archived files by only storing the changes to files (ΔV).

FileHamster *FileHamster* is a version tracking application focused on meeting the needs of content creators [39]. It functions in a similar way as AJC Active Backup: it continuously monitors user-specified files and automatically creates incremental backups whenever those files are modified. FileHamster allows the annotation of made changes with notes for a detailed overview or for quickly locating specific revisions. It also provides the possibility of viewing the differences between two file revisions. The core of FileHamster stores each changed file separately but the tool supports a multitude of plug-ins for extra functionalities (e.g. compression).

2.3.3.4 Incremental and entity-based system

	File based	Entity based
Snapshot based	CVS, SVN	StORE
Incremental based	AJC Active Backup FileHamster	ChangeList, SpyWare

Figure 2.8: Incremental and entity-based change management systems

The integration of the change management system into the developer's IDE allows implicit commits of changes without having to wait for the explicit request of developers. It also allows continuously capturing changes and committing the changed program entities resulting in a complete system history. Thus change management systems using this approach retain information about changed program entities and store a system history which can conceptually be seen as a sequence of changed program entities. This implies that the change management system must have knowledge about the implementation language of the software system stored in the repository.

Figure 2.8 focusses on the incremental approach combined with the entity-based one. There are a small number of available systems combining the incremental and entity-based approaches of which the ChangeList tool and the SpyWare repository are explored in the following two paragraphs.

ChangeList The VisualWorks for Smalltalk environment incorporates *ChangeList*, a tool that allows editing, comparing, merging and inspecting *change lists* [52]. A change list is a list of items representing changes performed on classes or methods. Each image [1] holds one single change list which records all the performed changes on that image. Hence the user may recover from a crash by backtracking to the most recent non-erroneous state of that image by using the ChangeList tool.

The ChangeList tool is mainly designed for manually merging different code-bases, comparing different code-bases and extracting the differences between several code-bases. It also allows the user to see the evolution of a particular class or method: what changes have been applied to that class or method during development? It can thus be used as a version control tool with the purpose of recording, applying and rewinding changes. The level of granularity however is restricted to classes and methods. A nice feature of ChangeList is the possibility to check for conflicts between existing changes.

SpyWare Robbes and Lanza propose a change-based software repository called *SpyWare* as the solution for the shortcomings introduced by using snapshot- or file-based change management systems for analyzing software evolution [41, 44]. SpyWare is an IDE plug-in for the Squeak Smalltalk environment. In their model, a system history is viewed as the sequence of changes applied to that system. Each

[1] Most Smalltalk systems store the application code (e.g. classes) together with the application state (objects) in one single file called an image. Images can be loaded by the Smalltalk environment restoring the application's code and state.

change is capable of reconstructing its successive state of source code, expressed by an *abstract syntax tree* (AST). A system is thus represented by an evolving abstract syntax tree.

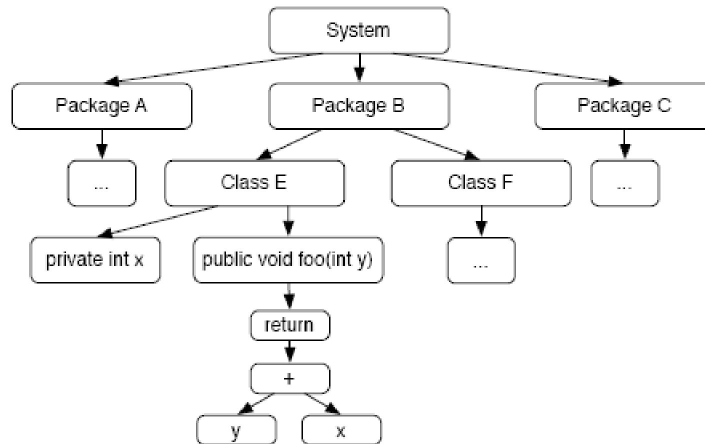

Figure 2.9: Program state representation (taken from [44])

Figure 2.9 shows an example of a program state represented by an AST. The root node represents the whole system, below the root we find the packages and modules of the program. Each package in its turn has children which are its classes. Class nodes have again children which are their attributes and methods. The children of a method form a subtree which is obtained by parsing its source code.

A change is then seen as an operation on the abstract syntax tree (of a particular program) and represents an action performed by the programmer when developing the system. The authors categorized the change operations as follows:

- *Atomic change operations*: operations which manipulate the abstract syntax tree of a particular program. These operations consist of creating, adding or removing a node (e.g. adding a class) as well as changing the properties of a node (e.g. modifying the name of a class). Each atomic change operation is applicable which makes it possible to generate each development stage the program went through during its evolution by iterating over the complete list of changes.

- *Composite change operations*: a system history consisting of only atomic operations would lead to an enormous amount of information. Therefore atomic operations are composable: they can be grouped together into higher-level operations with a more abstract meaning (e.g. renaming a method).

2.4 Problem statement

Criteria / Type	S/F	S/E	I/F	I/E
Complete information	✗	✗	✗	✓
Language independent				
- Monitoring	✓	✗	✓	✗
- Reasoning	✗	✓	✗	✓
Developer independent	✗	✗	✓	✓
Order preservation	✗	✗	✓	✓
Extensibility hooks				
- Monitoring	✗	✗	✗	✓
- Reasoning	✗	✗	✗	✓

Figure 2.10: Comparison of change management systems

The stated criteria are set out in order to find a change management system suitable for reasoning about software evolution (e.g. *complete information*). Figure 2.10 shows an overview in which those criteria are compared against the explored combinations in the previous sections. An “X” indicates that the criterium is badly or not supported by the change management system while a “V” indicates the opposite. As expressed by the figure, none of the discussed change management systems satisfies all criteria. The snapshot- and file-based systems satisfy only the *language independent monitoring* criterium which is not enough for good support (e.g. it does not offer complete information or it does not preserve order). A snapshot- and entity-based system is a better solution since it offers the possibility of *language independent reasoning* but yet again it is the only met criterium. The figure shows that in general the incremental systems score better than the snapshot-based ones. The *incremental and file-based* system is the second best solution but next to *language dependent reasoning* it suffers from two additional shortcomings: *incomplete information* and *no extensibility options*. As expressed by Figure 2.10, the *incremental and entity-based* system scores the best of all explored combinations hence it is favored above the other systems.

The explored incremental and entity-based change management systems however do not satisfy all criteria met by an incremental and entity-based approach.

- ChangeList: the ChangeList tool maintains changes applied to a Smalltalk program and stores change information specific to Smalltalk program entities. This implies that the ChangeList tool *does not support language independent reasoning*. Also, the level of granularity is restricted to classes and

methods and the changes maintained by the ChangeList tool only contain the new source code definition of the changed program entities. Additions or removals of attributes for example have to be detected by differencing different changes of a particular class. Other changes (e.g. additions or removals of packages) are not recorded at all. The available amount of change information is not enough for reasoning about it, ChangeList *does not satisfy the complete information criterium.*

- SpyWare: Opposed to ChangeList, SpyWare does offer *complete information* but it uses language and environment specific concepts such as the supported change operations representing an operation on the programs' abstract syntax tree. That abstract syntax tree is composed of program entities specific for the Smalltalk language implying that SpyWare *does not support language independent reasoning* [44].

2.5 Goal

The main goal of this work is to support reasoning about software evolution by using an *incremental and entity-based* change management system to store the evolutionary information. This dissertation supports our claim that

An *incremental and entity-based* change management system with *first-class changes* based on a *meta-model* offers *accurate evolutionary information* and supports *reasoning about software evolution*.

The encountered shortcomings of the discussed change management systems encourage the need for a *new incremental and entity-based change management system*. Such a change management system incrementally captures every applied change to a program entity and stores that information in the system history.

Instead of storing each changed program entity separately, it is better to maintain only the differences between them denoted by ΔV. A conceptual model consisting of types of changes, denoting those differences, is established based on the specification of the most suited meta-model. Changes to a program are expressed as *first-class changes* that contain all necessary change information enabling to access that information directly. The term first-class refers to the fact that an object may be stored in a variable or data structure, may be passed as an argument to a function and may be returned as the value of a function. The conceptual model

is thus the design of a class hierarchy of which each class represents a type of change.

A *meta-model* for programming languages describes the available program entities of the languages adhering to it. The search for a high quality meta-model is supported by evaluation criteria derived from those specified in this chapter. For example, the resulting meta-model must support multiple programming languages. The kind of change information for each program entity can be derived from the modeled program entities that multiple programming languages have in common.

First-class change objects retained in a repository and acquired by an incremental and entity-based change management system are expressive enough to represent and reconstruct the entire program. They contain all information that specifies them providing *accurate evolutionary information* about the evolution of the represented program.

The change management system composes the system history of an evolving program by capturing first-class changes which are beneficial for applications *supporting software evolution*. For example, the programmer's intentions can be recovered from a system history by mining it for change patterns. Such a pattern represents a set of rules to which a group of changes adheres in order to form a solution for that pattern. Those patterns raise the level of abstraction and provide semantic information increasing the degree of program understandability.

2.6 Conclusion

This chapter starts by giving a brief introduction to the field of *change management systems* and how they are useful for reasoning about software evolution. Most software evolution tools recover (extract) evolutionary information from the *repository* maintained by the change management system. That repository is a centralized library retaining changes of evolving software systems.

Secondly, this chapter sets out different criteria in order to find an adequate change management system for reasoning about software evolution. These criteria are: *complete information*, *language independence for monitoring changes and reasoning*, *developer independence*, *order preservation* and *hooks for extensibility for monitoring changes and reasoning* (see Section 2.2).

Section 2.3 studies change management systems based on a *time dimension* (when are changes stored) and a *structural dimension* (how are changes are stored/managed).

A *snapshot-based* change management system stores changes at *explicit request* of developers which may lead to *incomplete information*. By integrating the change management system into the developer's *Interactive Development Environment* (IDE), changes can be *incrementally* captured and *implicitly committed* without having to wait for the request of programmers. File-level storage of changes allows *language independent monitoring and storage* of changes due to their *textual nature*. The *file-based* approach leads to *incomplete information* which must be reconstructed by means of *complex parsing technology*. *Entity-based* systems store information about changed program entities (*language dependent*) providing *complete information*. Finally, an overview of the four types of change management systems is given.

The next section explores the problems caused by the discussed change management systems. A comparison of those change management systems against the stated criteria reveals that change management systems must strive for *incremental and entity-based* setting. The studied incremental and entity-based change management systems however do not satisfy all criteria met by an incremental and entity-based approach. Both ChangeList and SpyWare do not support language independent reasoning of software evolution. Additionally ChangeList restricts its level of granularity to classes and methods and thus looses a lot of important information.

The main goal of this research is to support reasoning about software evolution using change management systems. The encountered shortcomings of the discussed change management systems encourage the need for a new *incremental and entity-based change management system*. To support *language independent reasoning* a *language independent meta-model* of a programming language is required. Such a meta-model can be considered as an explicit description of what building blocks are defined in a programming language. As such, it can be used to describe programs and changes on programs which are specified in a language adhering to that meta-model.

The incremental and entity-based change management system is used to build a *repository of first-class change objects* by capturing the system history of an evolving program. By using a language independent meta-model as a basis for those change entities, we obtain also a *language independent classification of first-class changes*. The claim supported in this work is that: *an incremental and entity-based change management system of which its first-class changes are based on a good meta-model offers accurate evolutionary information and is thus useful for reasoning about software evolution*.

Chapter 3

A meta-model for support in software evolution

3.1 Introduction

The goal of this chapter is to find a *meta-model* suitable for deriving *first-class changes* to be used in the change management system under development. A meta-model for programming languages describes the building blocks defined in the languages adhering to that model. There are a number of alternatives for representing object-oriented software thus choosing one of them must be based on some criteria. These criteria relate to the different criteria stated in the previous chapter: *complete information*, *language independence for monitoring changes and reasoning about changes*, *developer independence*, *order preservation* and *hooks for extensibility for monitoring changes and reasoning about changes*.

This chapter analyzes some alternatives with respect to the imposed criteria followed by a comparison between the different alternatives in function of those criteria. Afterwards a feasible meta-model is chosen. The remaining sections of this chapter deal with the chosen alternative starting by exploring its specification. The final section of this chapter deals with extending the chosen alternative in order to support:

- *Dynamic state*: some software systems may never be shut down and require modifications to their source-code at run-time. The favored alternative is extended in order to derive changes with respect to the dynamic state of a running system.

- *Smalltalk features*: as a proof of concept that change management system is developed in Smalltalk thus the chosen model is extended to cope with language specific artifacts in order to derive Smalltalk specific changes.

3.2 Criteria for meta-models

Different criteria are set out in order to find a suitable meta-model for deriving a change classification that will be used in the change management system under development. These criteria relate to the criteria imposed in the previous chapter and are listed below. The first two correspond to the *language independent reasoning* and *extensibility hooks for monitoring and reasoning* criteria. The third criterium is derived from the *complete information* criterium while the last one is derived from the *language independent reasoning* criterium.

- *Support for multiple languages*: A *language independent classification of change types* can be derived from a meta-model that supports *multiple programming languages*. That classification may be used as an intermediary format for software evolution tools. This enables software analysis tools to study and compare cases implemented in different languages without adapting those tools for each supported programming language. To accomplish this, the meta-model must be as general as possible and thus omit language specific features.
- *Extensibility hooks*: often software evolution researchers may feel that the available spectrum of artifacts is insufficient to represent the information they need or to extract data from. Hence it must be possible to overcome this need by allowing *extension of the available set of changes*. Since those changes are derived from a meta-model, that meta-model must be *extensible* in order to be able to derive more changes. Extending the meta-model may decrease the degree of language independence (e.g. extending it to be able to cope with language specific features).
- *Derivable system invariants*: stored changes must ensure *complete information*: all desired information is *available* and of *high quality* i.e. correct and accurate. The quality of pending changes influences the state of the system and its consistency: it is not allowed to pend low-quality (e.g. missing information) or conflicting changes. That consistency is enforced by *system invariants* which are constraints that must be satisfied by the system at any time. The meta-model must thus serve as a good basis for deriving those

invariants. This enables software developers to track down which invariant was violated and in some cases conflicts can be resolved automatically.

- *Easy information exchange*: The *language independent change classification* may be used as an intermediary format for software evolution tools. Therefore that classification and the meta-model upon which it is based must be complete, consistent, easy to interpret and can not contain any ambiguities in order to exchange it. This is further referred to as *easy information exchange*. A high degree of language independence increases the ease of information exchange while a high degree of extensibility decreases the ease of information exchange. Therefore it is important to find a good balance between language independence and extensibility.

3.3 Alternatives

This section discusses some alternative meta-models and their support with respect to the stated criteria. The alternatives are classified into two categories: *modeling language based* and *graph based* meta-models. The following two subsections elaborate on those categories.

3.3.1 Modeling language based alternatives

3.3.1.1 UML meta-model

The Unified Modeling Language (UML) is a general-purpose modeling language widely used in the world of software engineering [5, 20]. It includes a graphical notation used to specify, visualize, construct and document designs. The UML specification consists of a *meta-model* that describes the language for specifying UML models (e.g. class diagrams).

- *Support for multiple languages*: UML supports the entire software development process starting from analysis and design omitting implementation specific issues. Therefore its meta-model is designed to model software systems implemented in various object-oriented programming languages.
- *Extensibility*: the UML meta-model provides three extensibility mechanisms as listed below.
  - *Tagged values* permit users to annotate any model element with extra information (value) paired with a keyword (tag). Figure 3.1 shows an

example: the information (visualized as a note) is shown in the figure while the keyword `documentation` is stored in the background.

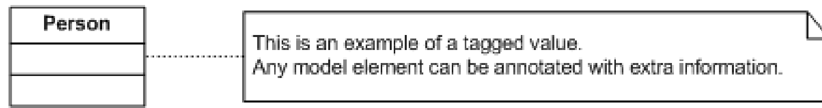

Figure 3.1: UML tagged values - Example

- *Stereotypes* allow users to further specialize model elements, it is an extensibility mechanism equivalent to inheritance. Figure 3.2 depicts an example of a stereotype, the `enumeration` stereotype denotes that `ColorTypes` is an enumeration providing some defined colors.

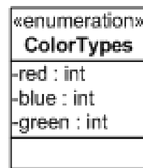

Figure 3.2: UML stereotypes - Example

- Users can apply semantic restrictions to model elements by using *constraints* which may be specified in free-form text or in *Object Constraint Language* (OCL). OCL is a declarative language that is part of the UML standard and is used for describing rules that apply to UML models. An example of OCL is expressed by Figure 3.3. Students have the possibility to enroll for courses whereas each course has a maximum number of allowed students. The visualized constraint restricts the enrollments for one course to its specified maximum.

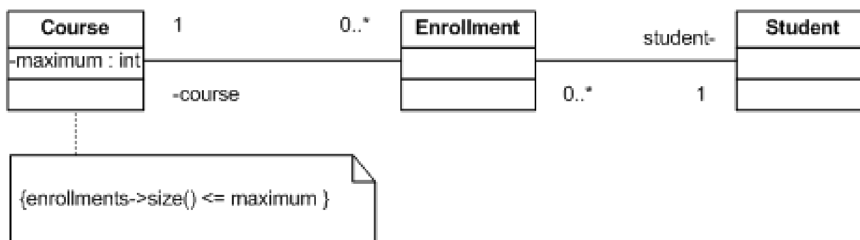

Figure 3.3: UML constraints - Example

- *System invariants*: system invariants can be derived from the meta-model's elements such as relationships, cardinalities or constraints.

- *Information exchange*: UML is very expressive due to its large number of available concepts and its extensibility mechanisms but this makes it harder for interacting tools to exchange information. The higher the extensibility, the less general the intermediary format used to exchange the desired information. Furthermore, there are some additional problems associated with the UML meta-model specification:
  - It is incomplete, vague and inconsistent [21, 40].
  - Modelers using OCL restrict their audience since OCL is a complex language and few people can read and write it [2].

  These problems decrease the ease of information exchange.

3.3.1.2 RevJava meta-model

The RevJava tool operates on compiled Java code and checks that software systems conform to their proposed architecture by checking which design or code rules are violated [14]. Its meta-model defines all relevant concepts of a software system and the associations between them. Examples of these are package, class or method. Some examples of associations are inheritance definition, method call or variable access.

- *Support for multiple languages*: the meta-model is designed for modeling Java-programs but it is general enough to capture software systems implemented in other object-oriented programming languages.
- *Extensibility*: the author has not explicitly specified any extensibility mechanisms.
- *System invariants*: system invariants can be derived from the meta-model specification. All derived invariants however apply to the Java programming language and in some extent to other programming languages.
- *Information exchange*: RevJava's meta-model is easy to read and understand since its number of available concepts remains small. No extensibility mechanisms are specified so RevJava's modeling language is less expressive than UML which makes it easier to exchange information using this meta-model.

3.3.1.3 FAMIX meta-model

FAMIX stands for *FAMOOS Information Exchange Model* and was created to support information exchange between interacting software analysis tools by capturing the common features of different object-oriented programming languages needed for software re-engineering activities [10, 11, 53].

Figure 3.4 shows a conceptual view of the FAMIX model. On the left hand side we see different programming languages used to implement several case studies. On the right hand side we see various experiments conducted by several software analysis tools on the provided case studies. In the middle we see the information exchange model that only captures the common features of object-oriented programming languages such as classes or methods. To cope with language specific features, the FAMIX meta-model can be extended by using the provided hooks represented by the grey bars at the bottom of the figure. The extended meta-model takes as input the source code of the different case studies which in its turn is provided as input to several software analysis tools.

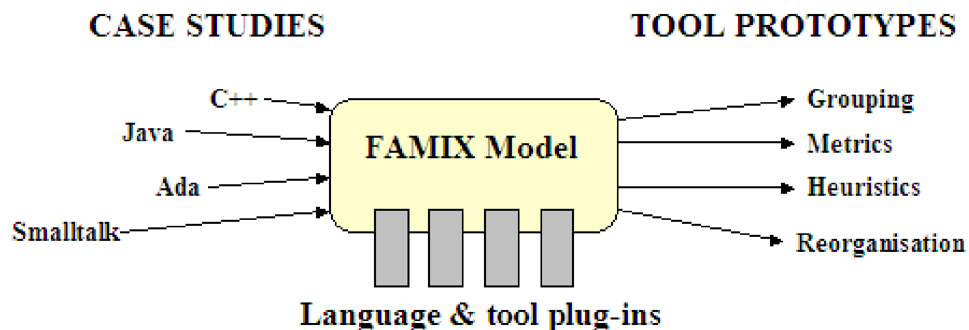

Figure 3.4: Conception of the FAMIX model (based on [10])

- *Support for multiple languages*: the FAMIX meta-model is designed to model software systems at source code-level independent of the implementation language. To achieve language independence, the FAMIX model only captures common features of different object-oriented programming languages, it omits language specific features.
- *Extensibility*: the FAMIX model provides three extensibility mechanisms as listed below.
  - *New concepts*: users have the possibility to define new meta-model elements.

  - *New attributes* to existing concepts: in order to store additional information users may extend existing meta-model elements by adding new attributes.
  - *Annotations*: like UML, FAMIX allows the user to annotate any model element by attaching extra information.

- *System invariants*: system invariants can be derived from the FAMIX specification.

- *Information exchange*: the FAMIX meta-model was created to *support information exchange between tools*.

3.3.2 Graph based alternative

Mens and Lanza suggest representing software systems as graphs. Program entities are then represented by nodes, relationships between them by edges [37]. To accomplish that, they specified a *typed meta-model* consisting of typed edges (e.g. inheritance and accesses) and typed nodes (e.g. class or method). Multiple edges between two nodes are allowed and attributes can be added to each node or edge.

- *Support for multiple languages*: the meta-model behind the graph representation supports any programming language whose concepts can be represented by either nodes or edges.

- *Extensibility*: the authors have not explicitly specified any extensibility mechanisms.

- *System invariants*: system invariants can be derived from the meta-model's specification.

- *Information exchange*: the used meta-model is easy to read and understand since its number of available concepts remains small. No extensibility mechanisms are specified hence this meta-model is also less expressive than UML which simplifies information exchange.

3.3.3 Comparison of alternatives

Figure 3.5 shows an overview in which the stated criteria are compared against the different alternatives discussed in the previous sections. An "X" indicates that the corresponding criterium is badly or not supported by the meta-model while a "V" indicates the opposite. As seen on the figure, the FAMIX meta-model scores the best of all explored alternatives: all criteria are well supported. The other alternatives have only one "X" meaning they are valid second best solutions. Both RevJava and the graph representation score badly for the extensibility criterium while UML suffers at the level of information exchange. Hence choosing between these second best solutions depends on the researcher's personal priorities with respect to the different criteria. Since the FAMIX meta-model supports all stated criteria, it is favored as the meta-model for deriving different first-class changes. The following two sections of this chapter elaborate on the FAMIX meta-model.

Requirements/Meta-model	UML	RevJava	FAMIX	Graph
Support for multiple languages	✓	✓	✓	✓
Extensibility hooks	✓	✗	✓	✗
Derivable system invariants	✓	✓	✓	✓
Easy information exchange	✗	✓	✓	✓

Figure 3.5: Comparison of alternatives

3.4 FAMIX definition

This section covers the complete FAMIX meta-model, subsequentially its basic data types, classes, attributes and relationships.

3.4.1 Basic data types

Before explaining the FAMIX meta-model, we discuss here FAMIX' basic data types available for variables and return types. FAMIX distinguishes two categories:

- *Primitive data types*: the usual data types considered as basic (e.g. `String` or `Integer`).

- *Non-primitive data types*: FAMIX defines three extra data types to be used in their model.
  - Name: a Name is a String that only bears semantics inside the model. A unique name of a concept for example applies only to the model where it is visualized.
  - Qualifier: a String that gets its semantics from outside the model. Source code for example is always stored in one or more files thus an attribute of a model element holding that location is only of value outside the model.
  - Index: an Integer representing a position in some sequence.

The naming conventions used when exchanging information are as much as possible compliant with UML [5, 20]. For more information about naming conventions, we refer the reader to [10].

3.4.2 Object

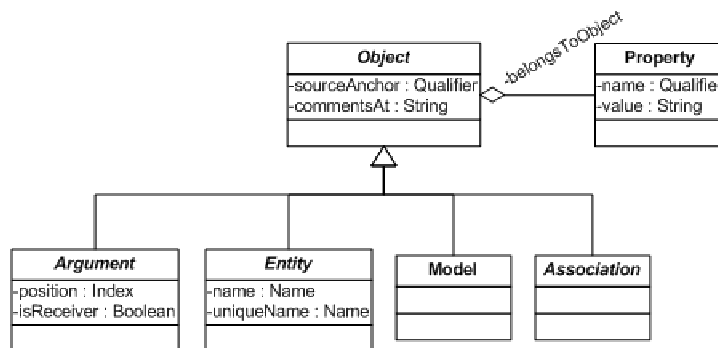

Figure 3.6: FAMIX model - Object

The Object class is the root class of the FAMIX model. As shown in Figure 3.6, Object is an abstract class not inheriting from any superclass but acting itself as a superclass for all other classes presented in the model (except the Property class). An Object holds a sourceAnchor determining the location of the concerned object. A typical example of a source anchor consists of the filename and start and stop indices where a class is physically stored. Developers have the possibility of commenting model elements, these comments are stored in the commentsAt attribute.

One of the mechanisms to extend the FAMIX model is annotating its elements with extra information. This is accomplished by instantiating the `Property` class which represents an annotation that is stored in the model element to which it is attached (`belongsToObject`).

A `Model` represents information concerning the particular system being modeled (e.g. name of the publisher or the used programming language). Models are used by software analysis tools when investigating the provided case studies. The other three subclasses (`Entity`, `Association` and `Argument`) are explored in the following three sections.

3.4.3 Entity

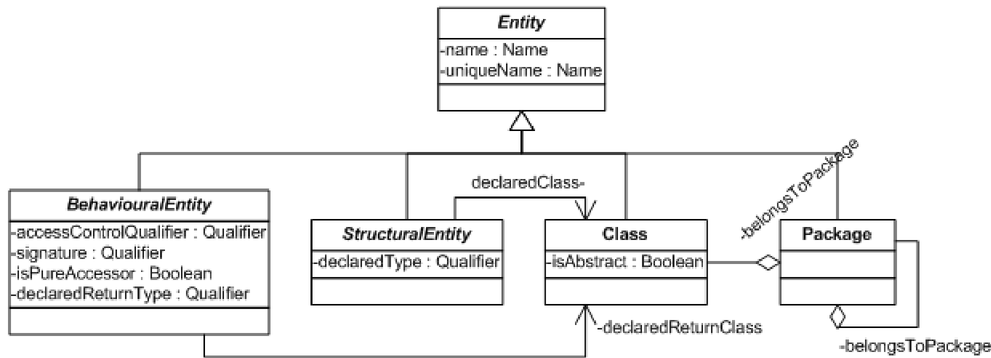

Figure 3.7: FAMIX model - Entity

Figure 3.7 shows the `Entity` class which represents different mechanisms that can be used in an object-oriented programming language to manipulate static structure, behavior and state of the implemented system. As shown in Figure 3.6, `Entity` is an abstract class inheriting from the `Object` class. An `Entity` stores a `name` and a `uniqueName`, that `uniqueName` must be unique for all entities visualized in its model. The `Class` and `Package` classes inherit from the `Entity` class and form the static structure of a software system. `Class` and `Package` represent respectively a class and a package in the context of object-oriented programming. A package organizes source code of a program into several subsystems. Packages and other entities (e.g. classes) may belong to maximum one package represented by the `belongsToPackage` relationship. The following two subsections respectively explain the `BahaviouralEntity` and `StructuralEntity` classes.

3.4.3.1 Behavioural entity

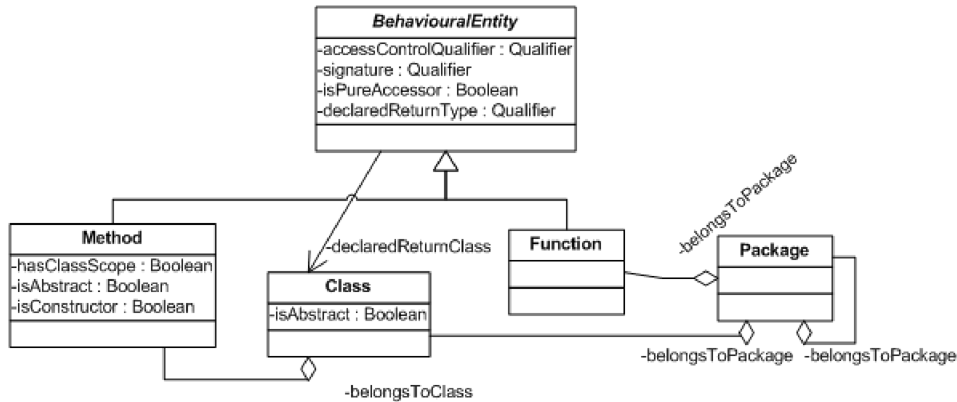

Figure 3.8: FAMIX model - BehaviouralEntity

Figure 3.8 shows the `BehaviouralEntity` class which represents the definition of a behavioral abstraction in source code. When invoked, a behavioral abstraction executes one or more actions defined in its body (e.g. calling other behavioral entities or instantiating classes). Typical examples of behavioral abstractions are methods and functions. As shown in Figure 3.7 `BehaviouralEntity` is an abstract class inheriting from the `Entity` class.

As seen in Figure 3.8, each behavioral entity has a unique signature within a class (`signature` attribute). The signature is not unique within the complete model because a lot of object-oriented programming languages allow the definition of different bodies in different classes for the same signature (*overloading*). A method lookup mechanism determines at runtime which of the overloaded methods is invoked.

By setting an access control qualifier, one can define who is allowed to invoke the behavioral entity (`accessControlQualifier`). Each behavioral entity returns an object, the concerned information is maintained by:

- `declaredReturnType`: the type of the returned object. Typically this will be a class, a reference to an object (*pointer*) or a primitive type.
- `declaredReturnClass`: a reference to the class implicitly held in `declaredReturnType`.

Each subclass of the `BehaviouralEntity` class represents a mechanism for defining a specific behavioral entity, FAMIX provides two such mechanisms:

- `Method`: a class used to represent the definition of some behavior specified within a certain `Class` (as expressed by the `belongsToClass` relationship).

- `Function`: in a similar way the `Function` class is used to represent the definition of some behavior independent from a `Class`. A function can be specified within a package denoted by the *belongsToPackage* relationship between the *Function* and *Package* classes. A function not specified in any package has global behavior.

3.4.3.2 Structural entity

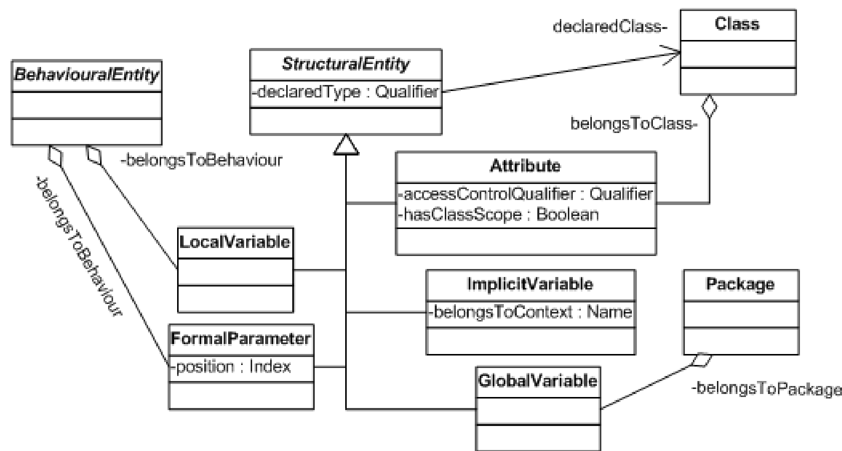

Figure 3.9: FAMIX model - StructuralEntity

Figure 3.9 shows the `StructuralEntity` class which represents the definition of an entity influencing the state of a system. A typical example is an attribute specified in a certain class to which a particular value can be assigned. Structural entities are characterized by their lifetime: some are bound to the lifetime of the class in which they are defined (e.g. attributes). Others have a lifetime equal to that of the entire running system (e.g. global variables).

As shown in Figure 3.7, `StructuralEntity` is an abstract class inheriting from the `Entity` class. In statically typed languages, each structural entity is declared as a certain type (class, pointer, primitive):

- `declaredType`: the type of the structural entity. Typically this will be a class, a reference to an object (*pointer*) or a primitive type.

- `declaredClass`: a pointer to the class implicitly held in `declaredType`.

Each subclass of the `StructuralEntity` class represents a possible variable definition, Figure 3.9 reveals five such mechanisms:

- `Attribute`: a variable declared within a class, represented by the `belongsToClass` relationship. The `hasClassScope` field indicates whether it is defined at instance-level or class-level. Instances of a class have their own copy of instance-level fields so that they can maintain a separate state. The lifetime of a field is bound to the lifetime of the class in which it is specified.
- `GlobalVariable`: a variable with a lifetime equal to that of the entire running system and which is globally accessible. The `belongsToPackage` association determines in which package the global variable is defined.
- `ImplicitVariable`: a context dependent reference to a memory location with a lifetime equal to that of the entire running system (e.g. `this` in C++/Java or `self` in Smalltalk). The result of that reference depends on the behavioral entity, object, ... where it is invoked on.
- `LocalVariable`: a variable defined locally within a behavioral entity, represented by the `belongsToBehaviour` relationship between the `BehaviouralEntity` and `LocalVariable` classes. Another wide spread term for referring to a local variable, is the term *temporary variable*. The lifetime of a local variable is bound to the lifetime of the invoked method or function where it is defined.
- `FormalParameter`: the declaration of what a behavioral entity *expects* as an argument thus *not* the argument passed through in an invocation. The `BehaviouralEntity` class stores the formal parameters defined in its signature. The `FormalParameter` class holds its position in the behavioral entity's parameter list. Formal parameters have a lifetime equal to the lifetime of the method or function where it is defined.

3.4.4 Association

An `Association` defines a relationship involving two entities. As expressed by Figure 3.6, the `Association` class is an abstract class inheriting from the `Object` class. Figure 3.10 shows the `Association` class and its subclasses. Each subclass denotes a different type of relationship, FAMIX provides three types:

- `InheritanceDefinition` represents a `subclass` inheriting from a `superclass`. The `InheritanceDefinition` class keeps a reference

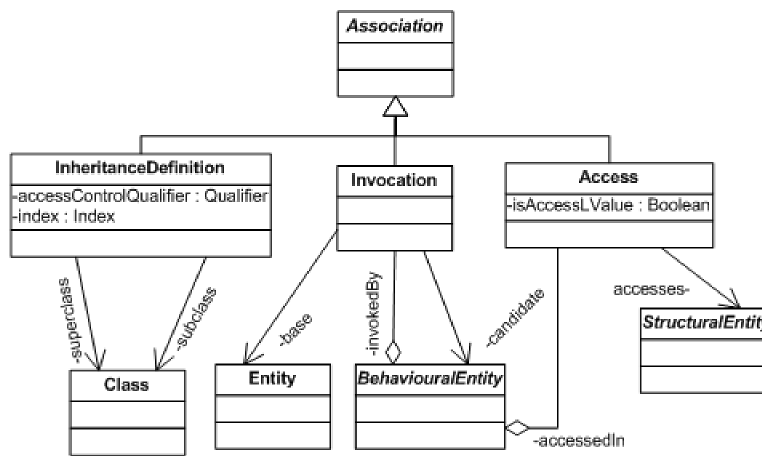

Figure 3.10: FAMIX model - Association

to both subclass and superclass. By setting `accessControlQualifier`, one can define how subclasses access their superclasses. To support multiple inheritance, subclasses maintain lists with their superclasses. The `index` attribute of `InheritanceDefinition` refers to the position of the superclass in such a list.

- `Invocation` denotes a behavioral entity (`invokedBy`) calling another one. An `Invocation` maintains a list of behavioral entities possibly called (`candidate`). All candidates have the same signature. At runtime, the method lookup mechanism reduces these candidates to one actual behavioral entity. The `base` relationship refers to the entity that defines the invoked behavioral entity.

- `Access` is used to represent a behavioral entity (`accessedIn`) accessing a structural entity (`accesses`). The `isAccessLValue` attribute indicates whether the concerned access corresponds to a getter action (which returns the value of a certain structural entity) or a setter action (which assigns a value to a certain structural entity). When `true`, it denotes a setter action.

3.4.5 Argument

An `argument` expresses the passing of an argument when invoking a behavioral entity. As shown in Figure 3.6, the `Argument` class is an abstract class inheriting from the `Object` class. Figure 3.11 shows the `Argument` class and its subclasses. An `Argument` holds its `position` in the corresponding argument

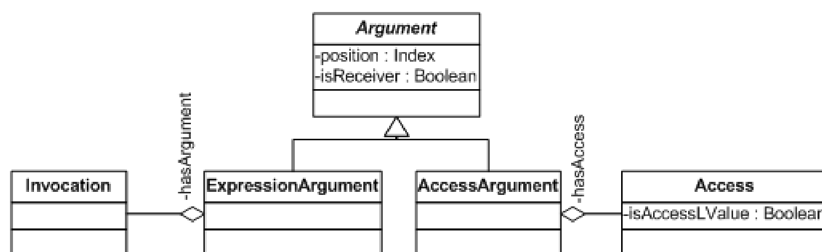

Figure 3.11: FAMIX model - Argument

list. Each subclass defines a specific type of argument, FAMIX distinguishes two types:

- `AccessArgument` specifies the passing of a reference (which refers to a structural entity). Obviously this passing resolves in an access to the concerned structural entity which is maintained by the `hasAccess` relationship.
- `ExpressionArgument` specifies the passing of a complex expression which invokes a behavioral entity. Hence the `ExpressionArgument` class stores the corresponding invocation.

3.5 FAMIX extensions

As mentioned earlier some software systems may never be shut down and require modifications to their source-code at run-time. In those cases, it may be useful to capture the dynamic information of that running system (e.g. the creation of a new instance). Therefore the FAMIX meta-model is extended in order to derive changes with respect to the dynamic state of a running system.

The incremental and entity-based setting for change management systems is implemented in the Smalltalk programming language. For implementation details, we refer the reader to Chapter 5. Any object-oriented programming language (e.g. Smalltalk or Java) defines entities unique to that language. Hence the second extension to the FAMIX meta-model copes with language specific artifacts in order to derive Smalltalk specific changes (e.g. FAMIX multiple inheritance vs Smalltalk single inheritance). The following two subsections respectively explore the extensions for dynamic state and Smalltalk features.

3.5.1 Extension for dynamic state

The FAMIX model does not cover the dynamic state of a running program: it does not provide any elements to store dynamic information such as living instances of a particular class or the value of some global variable. Originally it was not necessary to keep this kind of information: different tools analyze cases at source-code level and exchange their extracted information by using the FAMIX meta-model as an intermediary format. It is however necessary to store dynamic information whenever one wants to provide such a tool with dynamic software evolution or re-engineering functionalities. An example could be a garbage collecting functionality that removes all non-referenced instances. The following subsections discuss the dynamic extension of the FAMIX meta-model.

3.5.1.1 Instance

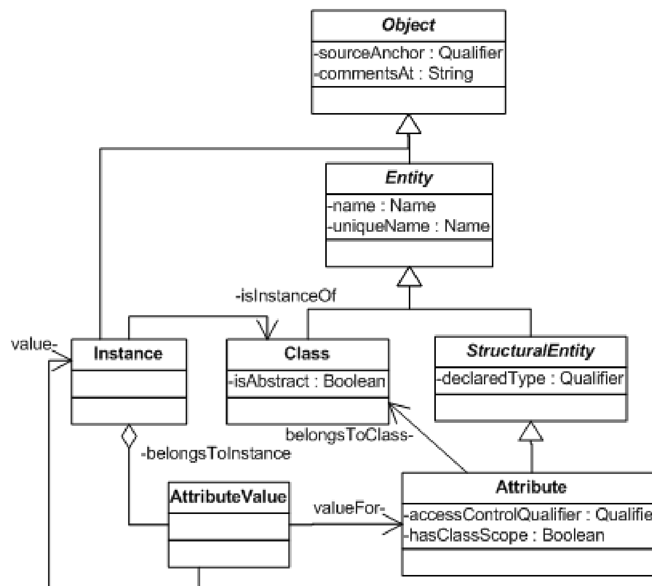

Figure 3.12: Dynamic extension - Instance

Figure 3.12 shows the extensions regarding living instances of a certain class. A new class `Instance` (inheriting from the `Object` class) has been added which keeps a reference to the class of which it is an instance (denoted by the `isInstanceOf` relationship). This enables an `Instance` instance to query the referenced `isInstanceOf` class for all its defined attributes and methods. `AttributeValue` has also been added and it holds a certain value, represented

by the `value` relationship, for a particular `Attribute` (`valueFor`) belonging to an `Instance` instance (`belongsToInstance`).

3.5.1.2 Global variable

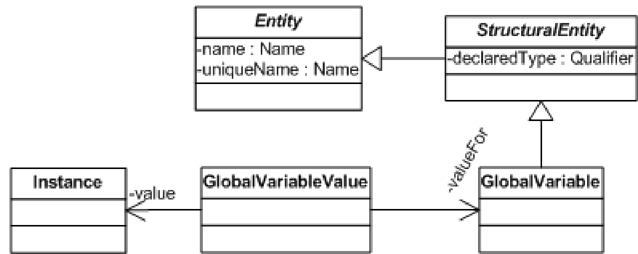

Figure 3.13: Dynamic extension - Global variable

Figure 3.13 shows the extensions regarding global variables of any running program. `GlobalVariableValue` keeps a reference to the global variable of which it holds the value (`valueFor`) which in its turn is an `Instance` instance.

3.5.2 Extension for Smalltalk

The FAMIX model serves as a meta-model for different implementation languages without specifying language specific artifacts and it does not cover the Smalltalk specific language features. This extension deals with those language specific artifacts and is based on the work of Tichelaar who extended the FAMIX model to capture Smalltalk's language features [53]. Extra modifications are provided to capture information not suggested by Tichelaar. Following subsections discuss these extensions: extensions suggested by [53] are indicated with "(T)", additional extensions are denoted by "(*)".

3.5.2.1 Class

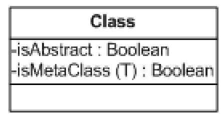

Figure 3.14: Smalltalk extension - Class

Each Smalltalk class has an associated metaclass that can describe it. This metaclass does not have its own name hence Smalltalk generates a name by concatenating the base class name with the " class" `String`. FAMIX defines a `Class` class allowing to model both class types. Figure 3.14 shows the modified entity, one attribute has been added: `isMetaClass`, a `Boolean` indicating whether or not the class represents a Smalltalk metaclass.

3.5.2.2 Behavioral Entity

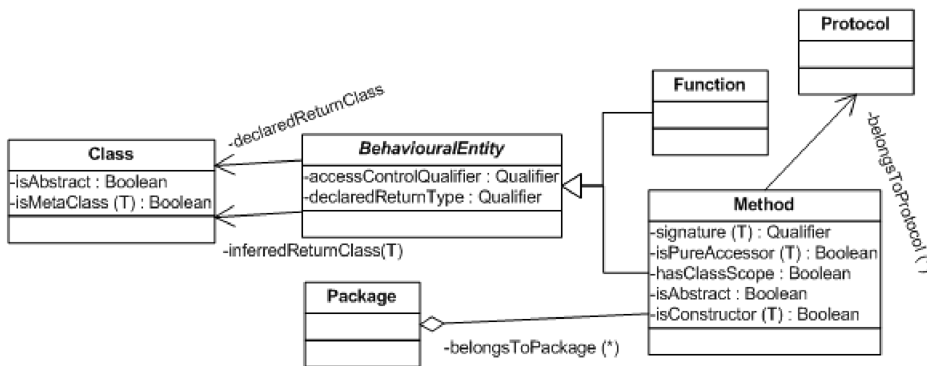

Figure 3.15: Smalltalk extension - Behavioral entity

Return types of methods are not explicit in Smalltalk so Tichelaar proposes to populate `declaredReturnClass` and `declaredReturnType` with the most general type of an object-oriented programming language namely `Object`. In [53], Tichelaar states that functions are not used in Smalltalk which implies that the `Function` entity of FAMIX will never be populated. Smalltalk allows however *block closures* which are very similar to *lambda functions*. These are anonymous functions that take a number of arguments and execute their body. In our extension, we use the `Function` entity to represent Smalltalk's block closures.

Figure 3.15 shows that the `inferredReturnClass` association has been added to the `BehavioralEntity` class. That association refers to all possible candidates for the return type of the concerned behavioral entity. Tichelaar has pushed down the following attributes to the `Method` entity:

- `signature`: each method has a unique signature. Obviously anonymous functions do not have any signature.
- `isPureAccessor`: a `Boolean` indicating whether or not the represented method is a pure getter/setter.

Figure 3.15 also shows that the `belongsToProtocol` relationship has been added. A *protocol* is the name for a group of methods allowing to organize them. For instance the "accessing" protocol groups all accessing methods (getters and setters).

An association between the `Method` and `Package` classes has been added because in Smalltalk a method belongs to exactly one package. The value of the `belongsToPackage` reference may differ from the package in which the containing class is defined. The `isConstructor` field indicates whether or not the behavioral entity creates and initializes new instances of its containing class.

3.5.2.3 Structural Entity

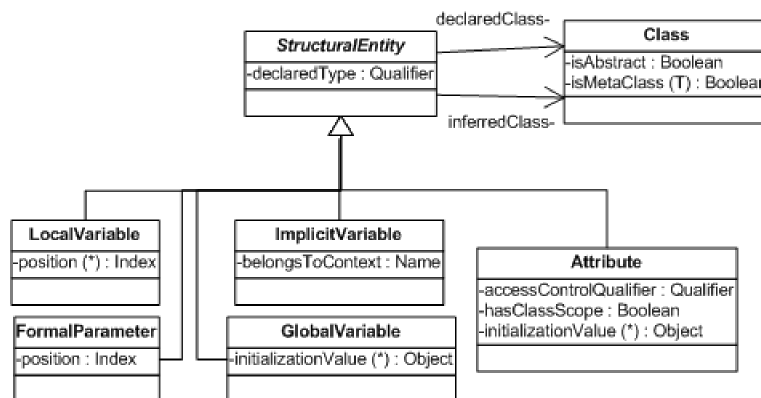

Figure 3.16: Smalltalk extension - Structural entity

Smalltalk is a dynamically typed language meaning it does not require the developer to explicitly type variables. Type checking happens at run-time and types of variables are determined by the values assigned to them. Therefore Tichelaar proposes to populate the `declaredType` field and `declaredClass` association with the most general type of an object-oriented programming language: `Object`. Figure 3.16 reveals that the `inferredClass` association has been added to the `StructuralEntity` class. This association refers to all possible candidates for the type of the structural entity. Smalltalk allows initialization of attributes and global variables thus the `initializationValue` attribute has been added to the `Attribute` and `GlobalVariable` classes. Furthermore the `position` attribute has been added to the `LocalVariable` class, it maintains the index of the local variable in the behavioral entity's list of temporary variables. By standard, Smalltalk protects all attributes. They are only accessible

within the defining class and its subclasses. Hence Tichelaar proposes to populate the `accessControlQualifier` attribute with the "protected" `Qualifier`.

3.5.2.4 InheritanceDefinition

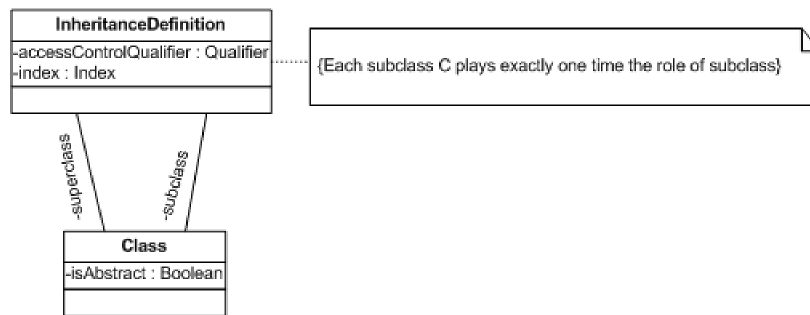

Figure 3.17: Smalltalk extension - Inheritance definition

The FAMIX model allows multiple inheritance whereas Smalltalk does not.
In Smalltalk, classes always inherit from one single class (except the root class, `Object`). Figure 3.17 depicts a constraint imposing single inheritance. Tichelaar proposes to populate the index attribute of the `InheritanceDefinition` class with the `null` value since in this case an index has no meaning. Inheritance in Smalltalk is always publicly accessible: all methods (public) and attributes (protected) are inherited by the subclass and have the same visibility.

3.6 Conclusion

This chapter starts by giving a brief introduction into the need and search for *meta-models* followed by a short overview of this chapter. A meta-model can be considered as an explicit description of which building blocks (program entities) are defined in the programming languages adhering to it. A meta-model is needed to derive *first-class changes* suitable for the change management system being implemented.

Secondly, this chapter discusses four criteria for choosing the appropriate meta-model: *support for multiple object-oriented languages*, *extensibility hooks*, *derivable system invariants* and *ease of information exchange*. Each proposed alternative has been analyzed with respect to those four criteria. After analyzing all

suggested alternatives, they have been compared with each other. That comparison reveals that off all discussed alternatives only FAMIX (*FAMOOS Information Exchange Model*) satisfies all four criteria.

FAMIX was introduced to exchange information between different software analysis tools that study the architecture at *source-code level* of a software system. The FAMIX model serves as a *language independent meta-model* for modeling object-oriented software. The remaining two sections of this chapter deal with the FAMIX meta-model starting by covering the entire FAMIX specification.

FAMIX specifies three relevant model elements: *Entity*, *Association* and *Argument*. The first concept enables the manipulation of static structure, behavior and state of the implemented system. The second element defines a relationship involving two entities (e.g. an inheritance definition). The last concept represents the passing of an argument when invoking a method or function.

This chapter ends by suggesting two extensions to the FAMIX model. Some software systems may never be shut down and require modifications to their source-code at run-time. In those cases, it may be useful to capture the *dynamic information* of that running system (e.g. the creation of a new instance). The FAMIX meta-model is extended in order to derive changes with respect to the dynamic state of a running system. The incremental and entity-based change management system is implemented in the Smalltalk programming language (for implementation details, we refer the reader to Chapter 5). Hence the second extension to the FAMIX meta-model copes with language specific artifacts in order to derive Smalltalk specific changes.

Chapter 4

First-class changes for support in software evolution

4.1 Introduction

Changes are *incrementally* captured and each change stores information about the *program entity* it affects. FAMIX, a language independent meta-model for modeling object-oriented software, specifies the available program entities of the programming languages it supports enabling to derive a classification of changes from it. By expressing those changes as *first-class objects*, change information can be directly accessed. We refer to those changes as *first-class changes*. In [45], Robbes and Lanza argue that first-class changes offer more accurate information about the evolution of a software system than file-based or snapshot-based techniques can.

Section 4.2 takes a closer look at the following properties of first-class changes: *detailed information*, *language independent representation*, *order preservation*, *abstraction* and *extensibility*.

Section 4.3 deals with deriving first-class changes from the FAMIX meta-model in order to obtain a *language independent class hierarchy of first-class changes*. Each first-class change object must be able to answer *what* it represents, *why* it exists, *when* it applies, *where* it applies (on which entities, associations or arguments) and *how* it applies. This matter is also explained in Section 4.3.

The next section discusses some applications that may benefit from using first-class changes. For example, first-class changes turn out to be useful in the context of software development (e.g. *merging* of different code-bases) and are also use-

ful for *reasoning* about the evolution of one or more programs (e.g. improving *program understandability* or *conflict detection*).

4.2 Properties of first-class changes

First-class changes are characterized by some important properties: *detailed information*, *language independent representation*, *order preservation*, *abstraction* and *extensibility*. These properties are discussed in the following four subsections.

4.2.1 Detailed information

First-class change objects encapsulate all the information that specifies them:

- "What" does the change object represent: what kind of change is it (e.g. an addition of a class) and what parameters were provided when creating it (e.g. the name of an added class)?
- "Why" does the change exist? Developers have the possibility of *annotating* changes by stating the reason of existence of a particular change. For example, some changes may be grouped according to the functionality they implement (e.g. security). According to [12], that annotation can be used for reasoning purposes (e.g. conflict detection and/or resolution).
- "When" may the change be applied? Each programming language imposes system invariants which are specified in its meta-model. In the FAMIX meta-model for example, each package has a unique name. These system invariants are used to preserve the system's consistency and can be used to derive preconditions for the first-class changes. Whenever a change is applied, its preconditions are checked ensuring that the system invariants will not be violated by that change. For example one of the preconditions for adding a package is that a package with its name does not yet exist. Appendix A lists the different invariants imposed by the FAMIX meta-model and its extensions.
- "Where" is the change applied? What entities, associations or arguments are affected when applying a change? A package for example may be added to another package so the addition of the new package affects its containing package.

- "How" is the change applied or undone? Each change knows how to apply or undo itself by using its *what* and *where* information.

4.2.2 Order preservation

A *change set* is the collection of all performed and captured changes during the evolution of a software system under development. An important issue in software evolution research is the sequence in which changes were applied however sets are unordered collections and do not take into account the position of elements. Software evolution researchers loose a lot of important information (e.g. when was that change applied? which changes were applied before it?) when not considering the order of changes (see Section 2.2). Therefore it is of great benefit that each change object is provided with a timestamp denoting date and time of creation. Changes are captured incrementally allowing to preserve the order in which they were applied (e.g. by acquiring the date and time at the moment a change is created). This allows the reconstruction of the complete program.

4.2.3 Language independent representation

By using the FAMIX meta-model for deriving first-class changes, first-class change objects may represent changes applied to software systems implemented in different programming languages as long as those languages adhere to the FAMIX meta-model. This allows *language independent storage* of those change objects by introducing a central unit as shown in Figure 4.1. That figure shows two systems (A and B), each implemented in a different programming language which is equipped with an incremental and entity-based change management system. Both implementations of the change management system conform to the same model of first-class changes and store the captured changes in a local system history. The central system, depicted at the top of the figure, responsible for a global system history, disposes of that same model which enables *inter-operability* via an intermediary format based on that model. The central unit takes exchanged changes as input, processes them according to its internal model of first-class changes by creating first-class change objects and maintains them in a global system history.

4.2.4 Extensibility

By working with a hierarchy of classes, the spectrum of types of changes can be extended by subclassing elements of that hierarchy. This promotes reuse when

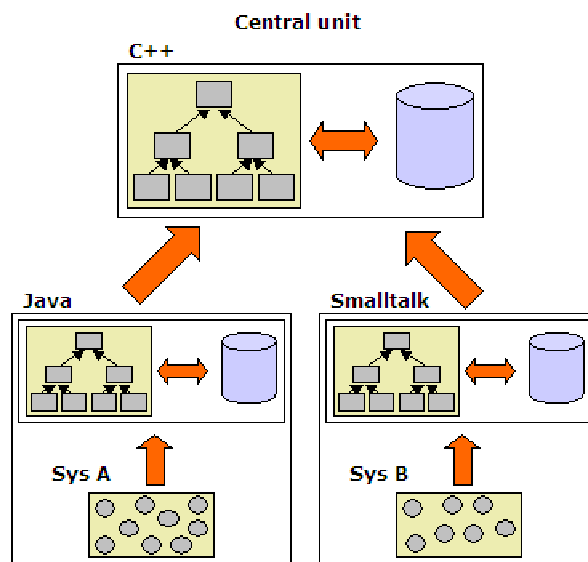

Figure 4.1: Language independent representation of first-class changes

designing new types of changes: a lot of coding effort can be saved by subclassing from existing change classes. For example, a Java developer decides to add two change classes respectively denoting the addition and removal of an interface to a Java program. One way to accomplish that is by subclassing the class representing a change to an `Entity`.

Also, each type of change can be extended by specifying extra parameters in order to provide more information. In a multi-user programming environment for example, it may be useful to know *who* performed *what* changes.

4.2.5 Abstraction

The higher the abstraction level, the easier and more natural it is to talk about changes. That is why different levels of abstraction are introduced for reasoning about changes. Figure 4.2 shows the three levels of abstraction.

- *Extension*: the lowest form of abstraction. The complete history of a software system can be represented for example as a sequential list of change objects. We refer to this as the *extension*.
- *Intension*: a *description* of a group of changes evaluating to (an extract of) the extension. By grouping changes into intensions, a better meaning is obtained (e.g. a group of changes that belong to a rename method refactoring).

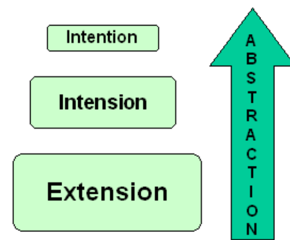

Figure 4.2: Abstracting changes

- *Intention*: the programmer's intention, what were the motifs for changing the system?

4.3 Design of first-class changes

A complete classification (a *class hierarchy*) of first-class changes has been established based on the FAMIX meta-model. This section discusses at a conceptual level a small extract of that hierarchy. For the complete specification, we refer the reader to Appendix B. The visualized classes do not contain trivial methods (e.g. getters and setters). Some diagrams contain associations between classes of the change hierarchy and classes of the FAMIX model described in the previous chapter. This section does not elaborate on those classes: they are visualized without any attributes and methods.

It may seem that change classes contain redundant information due to their associations with one or more FAMIX classes. Both end classes of such an association may contain the same attributes however this does not necessary imply redundant information. During the lifetime of a software system, program entities (an `Entity`, `Association` or `Argument`) may have been modified a numerous of times thus they contain information about their most recent state. First-class change objects however contain information to alter the state of those artifacts and may thus own the same interface as those program entities.

4.3.1 Change

What Figure 4.3 shows the abstract `Change` class which is the root class of the change hierarchy. Each change object (a `Change` instance) influences one or more `Entity` objects which may be contained in packages. In order to maintain

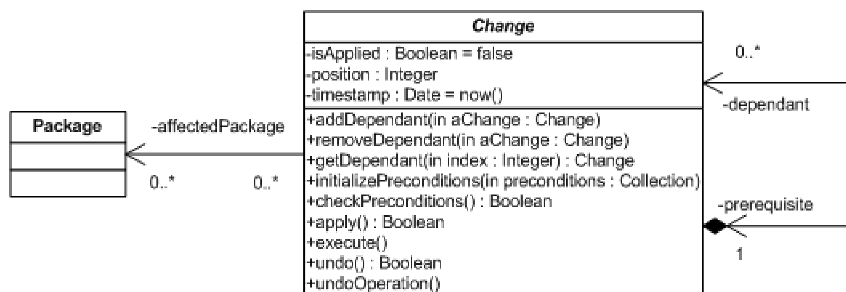

Figure 4.3: Design - Change class

clarity among the system history, changes own an `affectedPackage` association. This allows developers or researchers to group changes according to the package they affect. A first-class change also knows date and time of its creation, represented by the `timestamp` attribute. All changes in the hierarchy are *atomic* meaning they can not be split up however some changes contain information that can be analyzed in order to provide more useful information (e.g. the arguments of an invocation). Therefore it is allowed that such a `prerequisite` change object contains `dependant` changes expressing that information. To support that, the `Change` class provides following (abstract) methods:

- `addDependant`: adds a `Change` object as `dependant`.
- `removeDependant`: removes an existing `Change` `dependant`.
- `getDependant`: retrieves the `Change` `dependant` kept at the passed `index`.

Thus each `Change` subclass overriding those methods, allows its instances to add or remove dependant changes and to manage them. Dependants maintain a reference to their `prerequisite` change object and their `position` in their `prerequisite`'s collection of dependants.

How & Where The `Change` class can not be instantiated because it is an abstract class. As such it can not be applied or undone even if it provides the apply/undo mechanism to its subclasses. The `initializePreconditions` method is defined in the `Change` class and it initializes an empty list of preconditions. Each subclass may override that method to add preconditions. The `apply` method of the `Change` class specifies the necessary actions in order to complete the (re-)application of a change. Algorithm 1 shows the general steps of that `apply` method. Firstly, the `apply` method checks if all specified preconditions are satisfied. If so, the `execute` method is called: each change is

applied differently and may override the standard `execute` method provided by the `Change` class. Afterwards the `isApplied` flag is switched on and is returned. If one or more preconditions are not met by the system, the apply functionality returns `false`. The undo mechanism provides a *compensating action* for undoing changes and is similar to Algorithm 1. A simple example reveals how compensating actions can be used. To undo the creation of a method, that method needs to be removed from the system which is the *compensating action* of a method creation. The `isApplied` attribute is a `Boolean` indicating whether or not the change is applied to the current state of the program structure.

Algorithm 1 Generic apply method

```
if checkPreconditions then
    execute
    isApplied ← true
end if
return isApplied
```

4.3.2 EntityChange

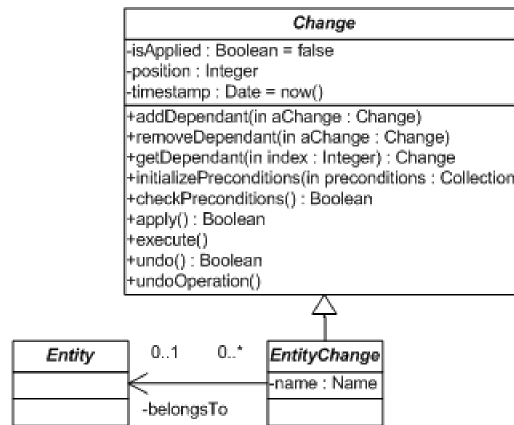

Figure 4.4: Change hierarchy - EntityChange

What Figure 4.4 shows the abstract `EntityChange` class that inherits from the `Change` class. It expresses a change to some entity (e.g. a `class` or `method`) and stores the `name` of the involved `Entity`. It also maintains a reference to the *containing entity* of the changed entity. An example clarifies this: a class is defined in some package P and that class may be changed during its lifetime. Then any change to that class is expressed by an `EntityChange` object and `belongsTo` refers to package P.

When

- `name` is not empty
- Each `EntityChange` belongs to maximum one `Entity`

4.3.2.1 ClassChange

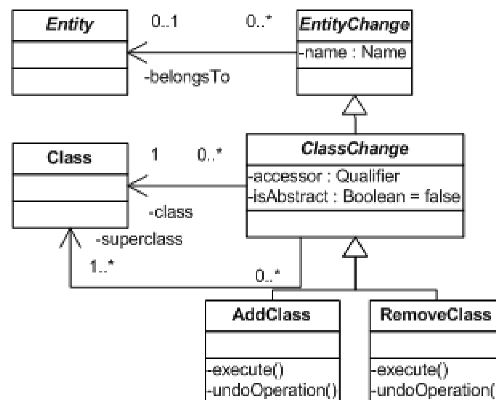

Figure 4.5: Change hierarchy - ClassChange

What The abstract `ClassChange` class (depicted in Figure 4.5) represents a change with respect to class entities (thus no other entities such as methods or functions). As seen in Figure 4.5, there is a `class` association between the `Class` and `ClassChange` classes. That `class` association refers to the `Class` entity manipulated by the `ClassChange` object. The `superclass` association refers to the superclass(es) of the changed class. The `ClassChange` class owns following attributes:

- `accessor`: the access control qualifier of the changed class (e.g. public or private).
- `isAbstract`: a `Boolean`, initially `false`, indicating whether the changed class is defined as *abstract* or not. An abstract class can not be instantiated.

Two subclasses of `ClassChange` are discussed in the following two subsections.

4.3.2.2 AddClass

What The `AddClass` class represents the creation of a new empty class within a certain package which is maintained by the inherited `belongsTo` relationship of `EntityChange`.

How & Where To apply an `AddClass` the following actions are required:

- *Add class C to package P*
- *For each superclass S of C, add an inheritance definition I between class C and class S*

This results in the addition of a new `Class` C and the creation of inheritance definitions between C and its superclasses. Furthermore the existing `Package` P is influenced.

When

- A `Class` with `name` C does not exist
- A `Package` with `name` P exists
- There is at least one `superclass`
- Each specified superclass of class C exists

4.3.2.3 RemoveClass

What The `RemoveClass` class expresses the removal of an empty class from a certain package. That package is maintained by the inherited `belongsTo` relationship of `EntityChange`.

How & Where To apply a `RemoveClass` the following actions need to be performed:.

- *Remove class C from package P*
- *For each superclass S of C: remove the inheritance definition I between class C and class S*

This results in the removal of an existing `Class` C and the deletion of all inheritance definitions between C and its superclasses. Furthermore the existing `Package` P is influenced.

When

- A `Class` with `name` C exists
- A `Package` with `name` P exists
- `Package` P contains `Class` C
- No methods are defined in `Class` C
- No attributes are defined in `Class` C

4.4 Applications for support in software evolution

This section discusses some applications that may benefit from using first-class changes. First-class changes offer some general advantages with respect to software development (e.g. *program generation*). They are also useful for reasoning about the evolution of one or more programs (e.g. *program exploration*). Researchers can give their analysis results to developers in order to improve the quality of the studied software system.

4.4.1 Program generation

Language independent representation of first-class changes enables semi-automatic *program generation*. Language independent representation involves the exchange of system histories between change management systems and a central unit. That program takes as input changes stored according to an intermediary format used by all change management systems (e.g. the system history of system A implemented in Java). That central unit can exchange the obtained changes with one specific change management system (e.g. the one integrated with Smalltalk). In figure 4.6, this is denoted by the double arrow between the central unit and the two change management systems each integrated into a programming environment. Each change management system can be extended in order to read exchanged change information and to instantiate the correct change objects based on that information. Therefore it needs its internal model of first-class changes and language specific elements require the intervenience of developers (e.g. `this` in Java

vs `self` in Smalltalk). Each change management system defines for each type of change how its instances must be applied or undone in the concerned programming language. This allows developers to construct a program in one language and to generate the same program with the same evolutionary history in another language. In Figure 4.6, systems A and B are equivalent to each other.

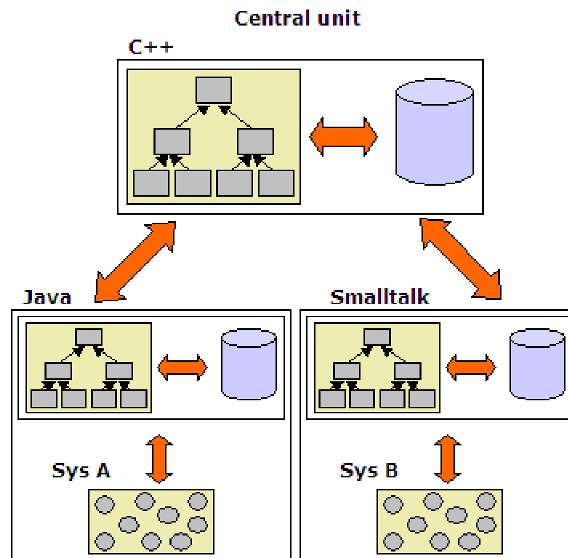

Figure 4.6: Application - Program generation

4.4.2 Merging and conflict detection

Developers who work in parallel on the same project want to update their code-base with each other's changes. This update process is called *merging* [32]. By exchanging their system history (or a snippet of it) programmers can easily merge the two code-bases. This is done by combining the exchanged changes with the local system history, ordering them and then sequentially applying the contained changes. Figure 4.7 shows an example of two developers who are working on the same system and who want to exchange their histories with each other in order to update both programs. Note that the language independent representation of changes enables the possibility of developers to use different programming languages. In this example, merging would lead to a *conflict* because both developers renamed the variable `name` differently. Such conflicts could lead to errors and a significant increase in debugging-time. In order to fix such a conflict, it has to be *detected* and *resolved* before the change histories are being applied. Change-based reasoning allows querying both histories for conflicting changes.

Conflicting changes can easily be detected by adding post-conditions to each type of change [22]. Those conditions say what the consequences would be of applying the change object easing the search for conflicting changes. Change objects can also be compared based on their timestamp and their extensive amount of information (e.g. the what information). In most of the cases, it is possible to resolve the conflict by asking the developer(s) which change should count as valid and which one should be discarded. By doing extra bookkeeping like holding all the usages of the variable, conflicts can be resolved very easily. *Conflict resolution* improves *testing* and *debugging* of software systems [12].

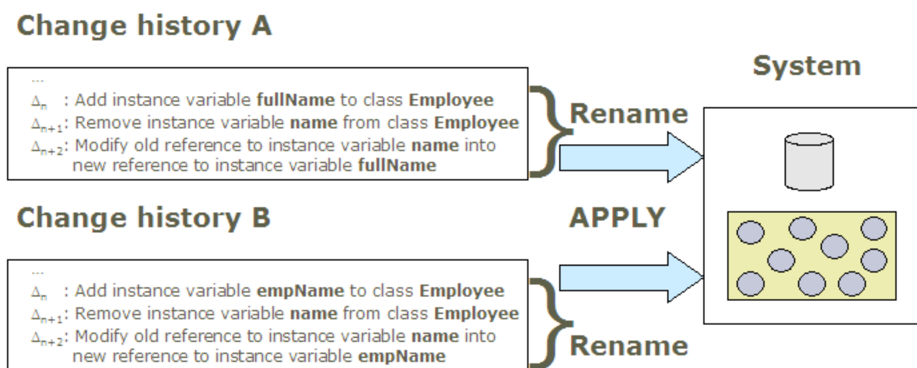

Figure 4.7: Application - Merging and conflict detection

4.4.3 Program exploration

First-class changes offer *complete information* i.e. no information is *degraded* or *lost*. The available information reflects with greater accuracy the way developers think about the structure, implementation and maintenance of a system which increases *program understandability*. The incremental capturing of modifications to the system also *eases its understanding* [43, 44, 45]. According to [12], first-class changes can be used to capture the intention of the developer when making certain changes. Grouping and documenting them also eases their understanding. Increasing understandability facilitates software maintainability which encourages component reuse.

Searching the extension for *meaningful intensions* (e.g. a rename method refactoring) allows software evolution researchers to reason thoroughly about the evolutionary history of a software system (e.g. by querying the extension). The recovery of meaningful intensions takes into account the order of changes and the information they hold. This is the technique asserted to recover the *programmer's*

intention and is very useful when trying to understand the purpose of some code. Knowing what the programmer intended to do with some changes improves the understanding of the involved code. In general, talking about intentions rather than code is far more natural and intuitive hence this reduces misunderstandings. For accomplishing that, it is necessary to recover the applied intensions (e.g. all rename method refactorings) and then map those intensions to the intention(s) of the developer(s) (e.g. improving program maintainability).

Thanks to the language independent representation, evolutionary information of software systems can be analyzed independent from the used programming languages. Researchers can also reason about merged system histories of different programs by using that language independent representation. All changes applied to the different systems are centralized in one single system history providing even more information to the researchers. Researchers may find useful information concerning two or more programs (e.g. common functionality or classes). That information can then be given to the developers responsible for maintenance.

4.4.4 Debugging

The extra level of abstraction introduced by the first-class changes improves the debugging process. By grouping changes into intensions, more information can be provided to the evolution researchers. A refactoring can be seen as a group of changes which can be described by an *intension query*. Figure 4.8 clarifies this: suppose a developer has the intention of improving the maintainability of a particular program by applying the *Pull up Method refactoring* of the `getName` method. Instead, he only pulls up the method from `Bookkeeper` and `Salesman`. Since the `getName` method still remains in the `Engineer` class, the refactoring was not carried out completely while the intention was to apply a refactoring. Badly executed refactorings could lead to errors while executing the program and an increased debugging time.

Figure 4.9 shows a snippet of the change history that corresponds to the example from above. After mining for the *Pull up Method refactoring intension*, we find that these seven changes only correspond for 66.66% to the pattern of the intension query. The `getName` method is implemented by all three subclasses but only pulled up from the `Bookkeeper` and `Salesman` classes. A correction can now be suggested to the developer for assisting the debugging process.

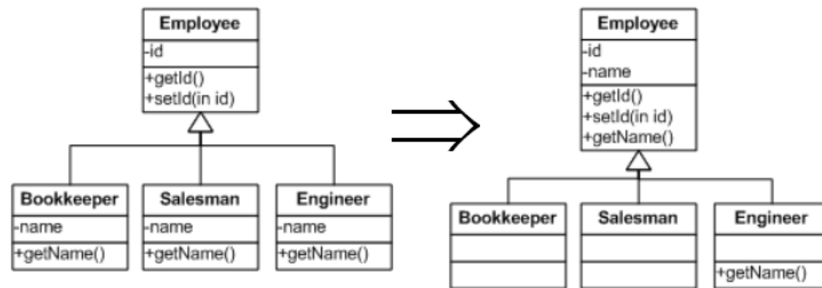

Figure 4.8: Application - Pull Up Method

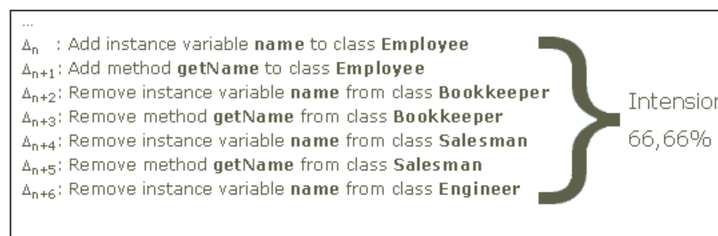

Figure 4.9: Application - Debugging

4.5 Conclusion

This chapter starts with introducing *first-class changes*. They are captured by an *incremental and entity-based* change management system which provides *more accurate* evolutionary information about a software system than the changes captured and maintained by the file-based or snapshot-based systems. For example, the incremental nature of the deployed change management system allows to *preserve the order* in which changes were performed. It is feasible that a first-class change object can answer *what* it represents, *why* it exists, *when* it applies, *where* it applies and *how* it applies. The different types of first-class changes are derived from the FAMIX meta-model which is language independent. Thus first-class change objects may represent changes applied to software systems implemented in *different programming languages* as long as those languages adhere to that meta-model. Different levels of abstraction are introduced for reasoning about changes: *extension*, *intension* and *intention*. The extension refers to the complete history of a software system while an intension is a description of (an extract of) the extension. The programmer's intention denotes his motivations for changing the program.

Next, a change hierarchy is established based on the FAMIX meta-model in order to obtain a *language independent class hierarchy of first-class changes*. The

available types of changes can be extended by *subclassing* classes of the change hierarchy. Also, each type of change can be extended by specifying *extra parameters* in order to provide more information. A small extract of the class hierarchy is studied where each type of change is explained by studying its detailed information (e.g. what or how).

The final section focusses on some applications that may benefit from using first-class changes. Thanks to the *language independent representation* of first-class changes, developers can construct a program in one language and *generate the same program* with the same evolutionary history in another language. Developers who work in parallel on the same project can *merge* their code-bases assisted by *conflict detection* and/or *resolution* which improves *testing* and *debugging* of software systems. Researchers have the possibility of reasoning about merged system histories independent from the used implementation languages. *Querying the extension* for *meaningful intensions* is the technique asserted to recover the *programmer's intention* and is very useful when trying to understand the purpose of some code. Knowing what the programmer intended to do with some changes improves the *understanding* of the involved code.

Chapter 5

Implementation

5.1 Introduction

This chapter deals with the implementation of *first-class changes* and *change management* which were discussed in the previous chapters. Those changes must be acquired *incrementally* and must be entity-based i.e. containing information about changed program entities. Section 5.2 deals with the search for an adequate programming language and development environment followed by a brief exploration of the two chosen alternatives. The third section concentrates on the implementation of the first-class changes: how is the *conceptual model* implemented? How are changes structured? How are they maintained? That section also reveals how the changes' *preconditions* are implemented. Section 5.4 focusses on *change management*: *creation*, *instrumentation*, *application* and *undoing* of changes. First-class change objects can be created in many different ways (e.g. *manually*, *interactive* or *automatically*). Instrumentation deals with acquiring changes and providing them with the information that specifies them. Last but not least, the application and undoing of changes is closely studied.

5.2 Environment

The previous chapter discusses a conceptual hierarchy of first-class changes designed to assist software evolution researchers. As a proof of concept that model needs to be implemented together with a change management system. A requirement for that change management system is the adoption of an *incremental* and *entity-based* approach.

An entity-based change management system stores changes about *program entities*. Since the conceptual hierarchy is based on an *object-oriented* meta-model, only program entities of object-oriented programming languages are considered. An incremental change management system captures changes as they are applied to the concerned program. This is possible by integrating the change management system into the programming environment. When capturing changes, the change management system may need to reason about the state of the monitored program. This can be accomplished by using a *reflective system* which is defined by Maes as

> "a system which incorporates structures representing (aspects of) itself. This representation makes it possible for the system to answer questions about itself (*introspection*) and support actions on itself (*intercession*)." [34, 35]

Introspection is the ability of a program to reason about its own state whereas intercession is the ability of a program to alter its execution state.

In order to keep the code-base consistent, each change class is equipped with its own *preconditions*. It might come in handy to reason about satisfied and unsatisfied constraints (e.g. developers want to know why their modification is not valid). There is thus a need for a mechanism expressing those preconditions as first-class functions or methods (e.g. *lambda functions*). Lambda functions are anonymous functions that take a number of arguments and execute their body.

5.2.1 Smalltalk

Smalltalk is a widely used *object-oriented*, *dynamically typed*, *reflective* programming language formerly released as *Smalltalk-80* [19]. An object is the most important concept in object-oriented programming but according to Alan Kay[1], *messaging* is the most important concept in Smalltalk. A dynamically typed language does not require the developer to explicitly type variables. Type checking happens at run-time and types of variables are determined by the values assigned to them.

The main reasons for choosing Smalltalk as implementation language are its support for *reflection* and *anonymous functions* (called *block closures*). An additional benefit of choosing Smalltalk over other programming languages is that Smalltalk is an easy to learn language with a higher efficiency regarding development speed

[1] one of the inventors of the Smalltalk programming language and one of the fathers of the idea of Object-Oriented Programming

than other programming languages. This allows to solve a certain problem quicker and with less code. It also has fewer defects over other programming languages [46].

5.2.2 VisualWorks for Smalltalk

VisualWorks for Smalltalk is an enterprise application development environment included in the Cincom Smalltalk Suite. It contains many useful components for developing Smalltalk and GUI applications (e.g. database connectivity or web applications). VisualWorks is instantly portable across a wide range of platforms (e.g. Windows or Linux) [52].

VisualWorks contains a *basic change framework* used for generating changes maintained in the *ChangeList* tool. That framework offers a basic functionality for the change management integration and is the main reason for choosing VisualWorks as our programming environment.

5.3 First-class changes in Smalltalk

5.3.1 Change classes

The previous chapter discusses a conceptual model of first-class changes. Each discussed conceptual change maps to a class of the implemented change hierarchy, expressing that type of change. The root class of that hierarchy is the abstract `Change` class and is depicted in Figure 5.1. The `Change` class is explained in the previous chapter and is therefore not explained in this text.

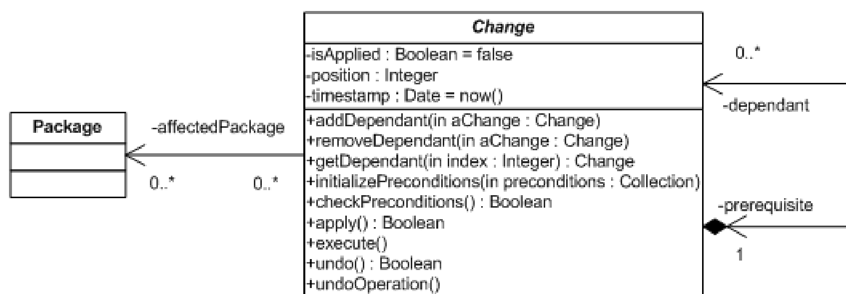

Figure 5.1: Change class

A first-class change object can be created by sending an *instantiation message* to the proper change class. Each change class defines a specific method accepting a number of parameters for instantiating itself. Those parameters are used for initializing the attributes of the new change object. Each first-class change object owns a private *state* which may be altered by sending messages to it. The *type* of a first-class change (which change class) together with its *state* answers the following question: *what* does the change represent?

5.3.2 Preconditions

Each programming language imposes *system invariants* which are used to preserve the system's consistency enabling to derive *preconditions* for the first-class changes. Whenever a change is applied, its preconditions are checked ensuring that the system invariants will not be violated by that change. The `Change` class defines an `initializePreconditions` method which returns a list of preconditions, initially empty. Each class in the change hierarchy has the possibility to overload the `initializePreconditions` method in order to:

- create a new list and forget the preconditions defined by its ancestors.
- create additional preconditions by first calling that initialization method on the superclass and then adding new preconditions.

Preconditions are expressed by using Smalltalk's block closure mechanism. Algorithm 2 depicts the `initializePreconditions` method of the `AddClass` class. Firstly, the preconditions specified by the superclass are initialized. Afterwards, five extra preconditions are specified.

Algorithm 2 Application - AddClassChange initializePreconditions

```
super initializePreconditions
precond add: [self name asQualifiedReference isDefined not]
precond add: [(Store.Registry packageNamed: (self packageName)) isNil not.]
precond add: [(self superclassName isNil) not.].
precond add: [(self superclassName = ”) not.].
precond add: [self superclassName asQualifiedReference isDefined]
```

1. A class with the provided name does not exist.
2. Classes belong to packages, the package held by the `AddClass` object must exist.
3. The name of the superclass of the `AddClass` object is not `nil`.
4. The name of the superclass of the `AddClass` object is not empty.

5. The provided superclass must exist.

The `Change` class implements a method `checkPreconditions` which evaluates either to *true* (all preconditions are met), to *false* (at least one precondition is not satisfied) or throws an *exception*. A thrown exception is an indication of a failed precondition. When invoked, the `checkPreconditions` method evaluates all specified preconditions sequentially. Note that *introspection* may be used for checking certain preconditions. This is for example the case when checking the first precondition shown in Algorithm 2. When executed, the block closure of that precondition reasons about the system's state by asking the system whether or not a class exists with the provided name.

5.4 Change management

5.4.1 ChangeLogger

A central class `ChangeLogger` is responsible for managing and collecting the captured changes. Changes are added via the `addChange` method and removed via the `removeChange` method. Furthermore `ChangeLogger` provides two methods for retrieving collected changes: `getChange`, which returns a change at a certain position, and `changes` which returns the change history. The `clear` method erases the entire system history. Note that at any time only one system history may exist thus also one instance of the `ChangeLogger` class. This is enforced by using the *Singleton Design Pattern*: it restricts the instantiation of a certain class to a specified number of objects (mostly one) [16]. `ChangeLogger` defines the `instance` method and the first time it is called, it creates a new instance of `ChangeLogger` which is then assigned to the `uniqueInstance` attribute. From then on, it is that unique instance of `ChangeLogger` that is returned by the `instance` method and used for registering changes.

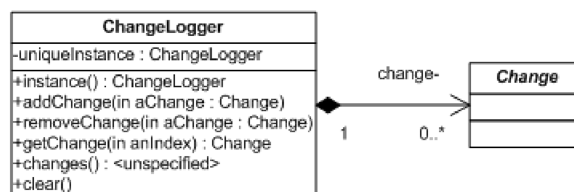

Figure 5.2: Singleton Design Pattern - ChangeLogger

5.4.2 Creation of change objects

There are several possibilities for creating first-class change objects. One way to create them is by manually invoking the *instantiation method* on the proper change class. A second approach for creating first-class change objects is by interacting with the IDE and create changes on the fly via *interactive dialogs* asking the user for particular change information. Another approach is by *loading change information* stored in a file or database. A fourth way of creating changes is by *capturing all information* via the developer's IDE. This enables automatic logging of the changes applied to the system, incrementing the system history change by change.

5.4.3 Logging of changes

This section deals with capturing changes and providing them with the information that specifies them. Thanks to Smalltalk's powerful reflective capabilities, that information can easily be provided to the first-class change objects. It is important to capture changes at the right place: the change information must be easily accessible and duplication of logging facilities must be avoided.

Whenever a developer applies some change to the source code (e.g. creating a new class), the VisualWorks IDE is responsible for carrying it out properly. This is done by performing the following two steps. Firstly, VisualWorks checks what kind of modification was performed and creates a `RefactoryObjectChange` object representing the developer's action. The `AddClassChange` class for example is provided in the basic change framework and inherits from the `RefactoryObjectChange` class. Its subclasses define an `execute` method describing the necessary actions to complete the developer's action (e.g. creating a class). Secondly, the VisualWorks IDE invokes the `execute` method on the concerned `RefactoryObjectChange` object.

The `execute` method serves as an excellent hook for integrating the logging facilities. We have provided each concrete `RefactoryObjectChange` subclass with the `prepareLogging` and `register` methods. Those methods are invoked in the `execute` method of the `RefactoryObjectChange` class. Algorithm 3 describes the three basic phases of that `execute` method. Note that developers have the possibility of switching the logging functionality on and off. That status is stored in the `loggingStatus` attribute of the `ChangeLogger` class. The following two subsections explain both preparation and registration steps.

Algorithm 3 Instrumentation - RefactoryObjectChange execute

```
if loggingStatus then
    self prepareLogging
end if
self primitiveExecute
if loggingStatus then
    self register
end if
```

5.4.3.1 Preparation

One RefactoryObjectChange class may represent different developer actions. AddClassChange for example represents the creation of a new class as well as the creation of new attributes. Therefore during the preparation phase, the exact type of performed change is checked by examining the system's state (introspection) and the change to be applied. Each RefactoryObjectChange subclass is extended by providing extra attributes that can be used during the preparation phase. That extra information can then be retrieved during the registration phase in order to instantiate the correct Change subclass defined in our change hierarchy.

The preparation phase is illustrated for the AddClassChange subclass. That subclass contains the following extra attributes:

- isNewClass: a Boolean indicating if the expressed class is a new class or an existing one.
- isNewInstanceVar: a Boolean representing whether or not new attributes (at instance-level) will be added to the represented class.
- newInstanceVars: a List holding the instance-level attributes to be added.
- isRemInstanceVar: a Boolean that indicates whether or not existing instance attributes will be removed from the represented class.
- remInstanceVars: a List holding the attributes that will be removed.
- oldInstanceVars: a List holding all instance-level attributes of the class that will be changed.
- hasModifiedSuperclass: a Boolean indicating if the inheritance definition of the class to be modified will be changed.
- oldSuperclass: a Class holding the current superclass before the change is applied.

- `instances`: an `Array` holding all current instances of the defined class.

Note that the `AddClassChange` class does not contain any attributes to keep information regarding class-level attributes, in Smalltalk called *shared variables*. The creation and removal of a shared variable are respectively caught in the `AddSharedVariableChange` and `RemoveSharedVariableChange` classes. The `prepareLogging` method of the `AddClassChange` class is described by Algorithm 4. First, it checks if a class with the provided name exists. If so, the `AddClassChange` object represents a change to an *existing* class and some extra information is gathered to know which changes are being performed. The existing class is retrieved from the code-base and the instance/class variables are fetched. Then there are two statements checking if there are instance-level attributes to be added or removed. Finally, the algorithm detects if the inheritance of the concerned class will be changed. All `RefactoryObjectChange` subclasses define a `prepareLogging` method similar to Algorithm 4.

Algorithm 4 Instrumentation - AddClassChange prepareLogging

```
isNewClass ← self name asQualifiedReference isDefined not
if not isNewClass then
    class ← self name asQualifiedReference value
    oldInstanceVars ← class instVarNames
    oldClassVars ← class classVarNames.
    self checkNewAttributes
    self checkRemovedAttributes
    self checkModifiedSuperclass
end if
```

5.4.3.2 Registration

Depending on the information provided during preparation, the `register` method takes actions. Algorithm 5 shows the `register` method of the `AddClassChange` class. If the represented class is a new class (expressed by the `isNewClass boolean`), an `AddClass` object needs to be registered into the system history. This is done by invoking the `logAddClass` method on the concerned `RefactoryObjectChange` object. In the other case, it is possible that several changes need to be logged:

- Each new attribute (added attribute) is logged as an `AddAttribute` change.
- Each removed attribute is logged as a `RemoveAttribute` change.
- If the inheritance definition between the concerned class and its superclass has changed, then a `ModifyInheritanceDefinition` needs to be logged.

Algorithm 5 Instrumentation - AddClassChange register

```
if isNewClass then
    self logAddClass
else

    if isNewInstanceVar then
        for all var in newInstanceVars do
            self logAddAttribute: var
        end for
    end if
    if isRemInstanceVar then
        for all var in remInstanceVars do
            self logRemoveAttribute: var
        end for
    end if
    if hasModifiedSuperclass then
        self logModifiedSuperclass
    end if

end if
```

The `logAddClass` method is expressed by Algorithm 6, the other log methods shown in Algorithm 5 are similar to Algorithm 6 and are not explained in this text. Algorithm 6 shows that a new `AddClass` object is created followed by assigning some values to its attributes (e.g. the created class, its name and package). After that, the change object is added to the system history by invoking the `addChange` method on the unique `ChangeLogger`. Finally, all attributes of the new class are logged as `AddAttribute` changes.

Algorithm 6 Instrumentation - AddClassChange logAddClass

```
change ← AddClass new
change class ← self definedObject
change name ← self name
change definition ← self definition
change belongsTo ← self package
change superclass ← self superclass
ChangeLogger addChange: change
for all var in (change class instVarNames) do
    self logAddAttribute: var

end for
```

5.4.4 Logging of changes at statement level

A method body contains a lot of useful and important information: it reveals links between two or more entities (e.g. an access to a variable or an invocation on an object). Thus the capturing of an added or removed method involves more than just creating an `AddMethod` or `RemoveMethod` object. Those objects respectively represent the creation or removal of an *empty* method. Temporary vari-

ables and all statements of the concerned method need to be mapped to the corresponding changes (e.g. `AddLocalVariable` or `RemoveAccess`). Those changes are referred to as *dependent* changes while the addition or removal of the method is referred to as the *prerequisite* change. Local variables are easy to detect: they are enclosed by the "|" symbol. Capturing the other changes is done by using a Smalltalk `Compiler` instance which parses the provided method body. The outcome is a parse tree composed of *program nodes* (e.g. `MessageNode` or `VariableNode`). An example clarifies this (see Algorithm 7): it depicts the `register` method of a `MailingList` class expecting a `Person` object. Firstly, a temporary variable `record` stores the email address of the given person. Secondly, that `record` is added to the mailing list on which the register method is invoked. Figure 5.3 expresses the method's parse tree that a `Compiler` would return. As figure 5.3 shows, the method comprises of two statements: an assignment and a message each one expressed by a `ProgramNode`. The first statement is held in an `AssignmentNode` which is composed of a variable and a value respectively represented by a `VariableNode` and a `MessageNode`. That `MessageNode` is composed of a receiver and a selector (message) respectively expressed by a `VariableNode` and `String`. The second statement is an invocation which is represented by a `MessageNode` and below that we find a receiver, selector and an argument.

Algorithm 7 Example - register: aPerson

record ← aPerson email

self add: record

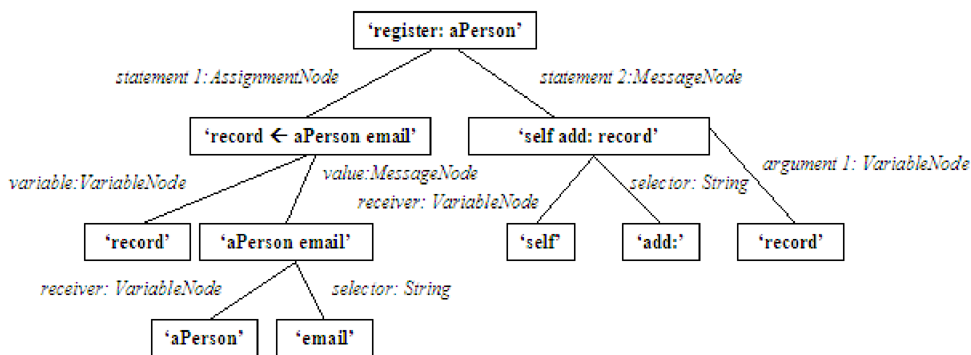

Figure 5.3: Example - Parse tree

The mapping of such a parse tree onto the available first-class changes happens by using the *Visitor Design Pattern*. The Visitor Design Pattern is an abstraction mechanism in order to separate an algorithm or operation from an object structure [16]. Figure 5.4 shows the actors involved in the design pattern. The

`ProgramNodeVisitor` class acts as *abstract visitor*: it defines an abstract visit method for each concrete program node. The *concrete visitor*, represented by the `ChangeVisitor` class, inherits from `ProgramNodeVisitor` and overrides one or more visit methods. The `ProgramNode` class acts as *abstract node* and defines an abstract `accept` method. Each subclass of `ProgramNode` is a *concrete node* element and overrides the `accept` method: each subclass implements it in another way. There are more concrete nodes than depicted in Figure 5.4.

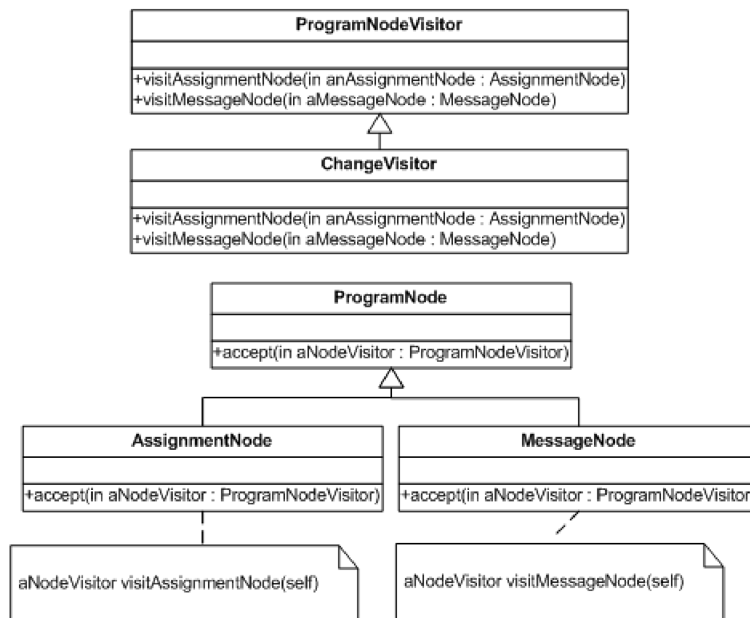

Figure 5.4: Visitor Design Pattern - ProgramNodeVisitor

Each visit method focusses on one single type of node and specifies which and how changes must be instantiated. The `visitVariableNode` for example creates either an `AddAccess` or `RemoveAccess` change. It provides general information such as the package, the initiator (who defines that access), the accessed structural entity, ... Additionally, the `VariableNode` knows whether the variable is read and/or written to and provides this information to the change object. All changes detected by the `ChangeVisitor` are collected and added to the system history after the `Add/RemoveMethod` and `Add/RemoveLocalVariable` change objects. The logging of the explored example would result in the following changes (which in their turn may contain dependent changes):

1. Add method “register: aPerson” (`AddMethod`)
2. Add local variable “record” (`AddLocalVariable`)

3. Add write access: write value of “aPerson email” (`AddInvocation`) to local variable “record” (`AddAccess`)
4. Add invocation (`AddInvocation`): call message “add:” (`String`) on “self‘” (`AddAccess`) with argument “record” (`AddAccessArgument`)

5.4.5 Application of changes

The `Change` class implements an `apply` method that incorporates the structure of the application methodology (see Algorithm 19). The apply method answers the following questions: *how* is a change applied and *where* is it applied (what artifacts are affected by its application). That method is inherited by all `Change` subclasses and may not be overloaded. When invoked on a particular change object, the specified preconditions are checked resulting in a `Boolean`. If not all preconditions are met, `false` is returned. In the other case, the logging status is stored in a temporary variable `lstatus`. Changes being applied via the `apply` method are not logged hence the logging status is toggled off. Next the concerned change is executed by calling the `executeOperation` method. Afterwards, the logging status is restored to the value it had before the execution. Successful application results in toggling the `isApplied` flag on.

Algorithm 8 Application - Change apply

```
if self checkPreconditions then
    lstatus = loggingStatus
    if loggingStatus then
        ChangeLogger toggleLoggingOff
    end if
    self executeOperation
    ChangeLogger loggingStatus: lstatus
    self isApplied ← true
    return true
else
    return false
end if
```

Each concrete `Change` subclass implements the `executeOperation` method differently yet all implementations have a common methodology. They all use the classes present in the basic change framework (e.g. `AddClassChange`) which are intercepted by the change management system when developers perform changes. Those classes can be instantiated by providing all information necessary to perform them correctly. After instantiation, the `execute` method can be invoked on such an object to apply it to the current code-base. Due to the switched off logging status, it is not recorded in the system history. Algorithm 9 shows the `executeOperation` method of the `AddClass` class. Firstly, a

new AddClassChange object is created by providing the definition held by the AddClass object. Secondly, that definition stores also instance and class variables hence these are removed so that they would not be created (AddClass only expresses the addition of an empty class). Thirdly, the package to which the class must be added is set and finally that AddClassChange object is executed in order to create a new class based on the information held by the concerning AddClass object.

Algorithm 9 Application - AddClass executeOperation

```
addClass ← AddClassChange definition: self definition
addClass resetVariables.
addClass package ← self belongsTo

addClass execute
```

5.4.6 Undoing of changes

The Change class implements an undo method to undo the change on which it is invoked (see Algorithm 10). This method is inherited by all Change subclasses and may not be overloaded. Firstly, the logging status is stored in a temporary variable lstatus. Changes being undone via the undo method are not logged hence the logging status is toggled off. Next the concerned change is undone by calling the undoOperation method. Afterwards, the logging status is restored to the value it had before the undoing. A successful undo of a change results in toggling the isApplied flag off.

Algorithm 10 Application - Change undo

```
lstatus = loggingStatus
if loggingStatus then
    ChangeLogger toggleLoggingOff
end if
success ← self undoOperation
if lstatus then
    ChangeLogger toggleLoggingOn
end if
if success then
    self isApplied ← false
end if

return  success
```

Each concrete Change subclass implements the undoOperation method differently yet all implementations have a common methodology. As opposed to the executeOperation method, the undoOperation uses *compensating actions*. Instead of using the existing classes of the basic change framework, change classes of the opposite type are used (e.g. the undo of an AddClass can be

compensated by the application of a `RemoveClass`). Algorithm 11 shows the `undoOperation` method of the `AddClass` class. It creates a new instance of the `RemoveClass` class by copying all information necessary to carry out the undo action. Then the `apply` method is invoked on that object and all steps explained in the previous section are performed.

Algorithm 11 Application - AddClass undoOperation

```
undo ← RemoveClass copyFrom: self
undo apply
```

Please note that undone change objects can be re-applied by invoking the `apply` method on it. First-class objects keep the information that specifies them regardless of their `isApplied` status. This makes experimenting with change objects easier for developers.

5.5 Conclusion

This chapter discusses the implementation of *first-class changes* and *change management* which were discussed in the previous chapters. Changes are *incrementally* captured and contain information about changed *program entities*. Section 5.2 seeks an adequate programming language and development environment. Incremental acquisition of changes is possible by integrating the change management system into the programming environment. Gathering change information may require some knowledge about the systems' state at the point of logging. This can be accomplished by using a *reflective system* which incorporates structures representing (aspects of) itself. This makes it possible for the system to answer questions about itself (*introspection*) and support actions on itself (*intercession*). Each first-class change object is equipped with *preconditions* in order to preserve the systems' state by not violating its *invariants*. It may be useful to treat those preconditions as first-class objects (e.g. *lambda functions*) in order to reason about satisfied and unsatisfied constraints. Lambda functions are anonymous functions that take a number of arguments and execute their body. *Smalltalk* is chosen as implementation language and *VisualWorks for Smalltalk* as development environment and both are briefly explained.

The next section concentrates on the implementation of the first-class changes and seeks answers for the following questions. How is the *conceptual model* implemented? How are changes structured and maintained? It also explains how the changes' preconditions are implemented.

Section 5.4 focusses on *change management*: *creating*, *instrumentating*, *applying* and *undoing* changes. First-class change objects can be created in many different ways (e.g. *manually*, *interactive* or *automatically*). The instrumentation of changes involves the acquisition of changes and then provide them with the information that specifies them. That section explores into detail the instrumentation process and clarifies it with an example. That process takes into account that a method body reveals links between two or more entities (e.g. an access to a variable or an invocation on an object). Thus the capturing of an added or removed method involves more than just creating an `AddMethod` or `RemoveMethod` change object. Temporary variables and statements of the method body need to be mapped to the matching change classes. This is done by parsing the methods' body which generates a *parse tree* composed of *program nodes*. These nodes can then be visited by using the *Visitor Design Pattern* which allows separation of an algorithm or operation from an object structure. Once registered into the system history, first-class change objects may be applied, undone and re-applied which is possible due to the detailed information they possess. The apply and undo functionalities are explained thoroughly and clarified with a small example.

Chapter 6

Validation

6.1 Introduction

This chapter validates if the incremental and entity-based change management system with first-class changes really offers *accurate and useful information* for reasoning about the evolution of a software program. This chapter starts with explaining the infrastructure for reasoning about software evolution. Section 6.2 explains why the *Smalltalk Open Unification Language* (SOUL) is chosen as reasoning language. Section 6.3 explains how intensions can be recovered from program histories by using SOUL queries. *HotDraw* is taken as a case monitored by an incremental and entity-based change management system. Section 6.4 firstly explains what HotDraw is and why it is chosen for validating purposes. Secondly, it explains two evolution scenarios applied to that case. The next section reasons about the evolving HotDraw system: First at a basic level studying *basic evolutionary information* (e.g. the total number of changes) followed by a more advanced study by searching patterns of changes (e.g. refactorings).

6.2 Environment

The change management system is used to capture the system history of an evolving program resulting in a repository of first-class change objects expressing that history. The information in a system history can be consulted by searching for *patterns*. Such a pattern can be expressed by a *set of rules* to which a group of changes must comply. This is the technique asserted in *declarative programming languages* which are high-level languages describing a problem rather than

defining a solution for solving it [33]. A declarative programming language is thus perfectly suited for describing change patterns. It also offers the advantage of having to write less "code" than when using the traditional programming languages. It is feasible to use a declarative programming language compatible with Smalltalk for easy integration with the implemented change management system.

Wuyts et. al. have implemented *SOUL*, a logic-based declarative programming language which is integrated into Smalltalk environments [54]. SOUL stands for *Smalltalk Open Unification Language* and is designed for *declarative meta programming* [54]. Declarative meta programming is a programming approach where declarative programming languages are used for writing *meta-programs* which process other programs. For more information about declarative meta programming, we refer the reader to [36, 54]. SOUL is similar to *PROLOG* [24, 25] but it offers the possibility of using Smalltalk code in its language. Additionally SOUL can process programs implemented in Smalltalk, Java and C. Therefore it respectively needs the LiCoR, Irish and Zombie libraries.

In SOUL, *logic variables* are denoted with question marks and the comma is used for the boolean *and*. Terms between square brackets are *Smalltalk terms* and they contain Smalltalk expressions which can refer to logic variables. A predicate can have different specifications called *rules*: each rule describes a possible solution. The boolean *or* is introduced by allowing different rules for the same predicate.

6.3 Intension queries

This section introduces some queries which can be used to recover programmer's intensions. The following two sections respectively deal with basic and advanced intension queries.

6.3.1 Basic queries

Basic intension queries are small queries used to build more complex ones. Algorithm 12 shows the most basic predicate `change(?CH)` described by only one *rule*. The `member(?EL,?COL)` predicate gets a member `?EL` of a collection `?COL`, in this case the available system history. That member is then *bound* (assigned to) to the variable `?CH`. Concretely in Algorithm 12, `?CH` gets bound to every member of the collection denoted by `[ChangeLogger allChanges]`. This is a Smalltalk block which evaluates to the collection of all changes.

Algorithm 12 change(?CH)

```
member(?CH,[ChangeLogger allChanges])
```

Queries can be executed in SOUL's *Query Browser*. That browser allows a user defined order of the logic variables used in the query and browsing through the results one by one. Figure 6.1 shows a screenshot of that browser containing the first fifteen results for the `change(?CH)` query.

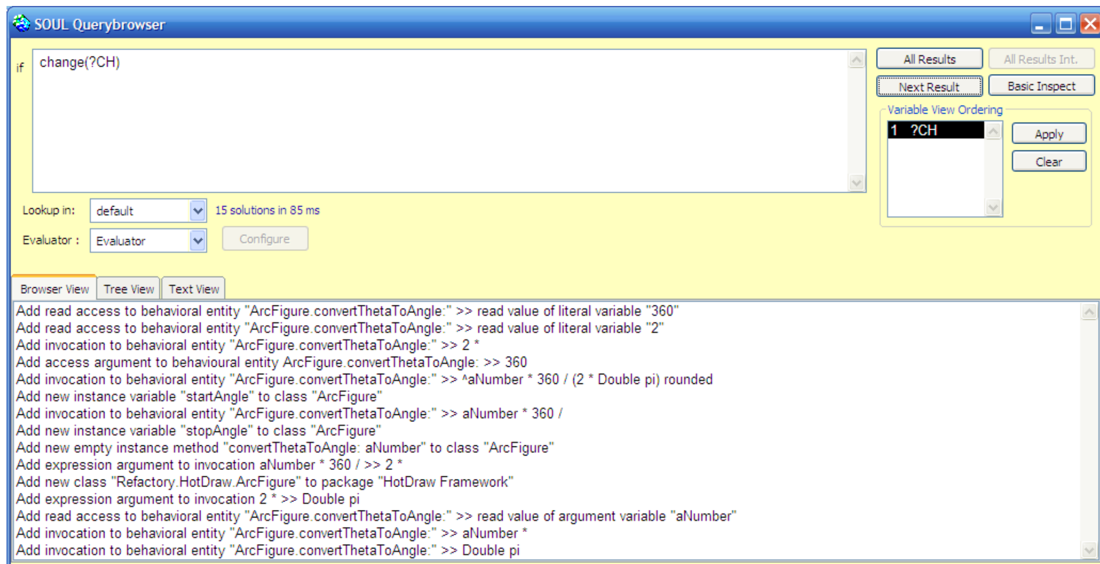

Figure 6.1: SOUL Query Browser - An extract of the results for change(?CH)

The `change(?CH)` predicate can now be used to build bigger blocks in order to find specific types of changes (e.g. an `AddClass` or `RemoveMethod` change object). For each concrete `Change` subclass, a rule was developed describing change objects of that type. Algorithm 13 depicts the `addMethod(?C,?M,?CH)` predicate. A result for that query is a change `?CH` of type `AddMethod` expressing the creation of a method with name `?M` in a class with name `?C`.

Algorithm 13 addMethod(?C,?M,?CH)

```
change(?CH),
isAddMethod(?CH),
equals(?M,[?CH name]),
equals(?C,[?CH className ])
```

6.3.2 Advanced queries

Specific change predicates (e.g. `addMethod(?C,?M,?CH)`) can now be used to describe more complex patterns of changes. Instead of asking which methods were added during the evolution of the concerned program, users can restrict their question to methods returning or setting the content of an attribute (getters and setters). The `addGetter(?C,?F,?M,?CH)` predicate is described in Algorithm 14. A result for that query is returned if there exists an `AddMethod` change object `?CH` expressing the addition of a method `?M` in a class `?C` followed by the addition of a returning read access of field `?F` in that method. The result is not complete if the added method does not have the same name as the added field.

Algorithm 14 addGetter(?C,?F,?M,?CH)

```
addMethod(?C,?M,?CH),
addReadAccess(?C,?F,?M,?RA),
after(?RA,?CH),
equals(?M,?F),
isReturnValue(?RA)
```

By combining different smaller change patterns, *high-level change patterns* can be described. One specific category of higher-level patterns concerns refactorings. Fowler defines a refactoring as a

> "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [15]."

He describes each refactoring as a sequence of steps in order to apply the refactoring in a safe way. Based on those mechanics, rules were developed to describe the following refactorings: *Self Encapsulate Field*, *Pull Up Field*, *Push Down Field*, *Move Field*, *Rename Method*, *Add Parameter* and *Remove Parameter*. Each refactoring query is equipped with a basic metric indicating how well the detected refactoring was performed (e.g. How many invocations of a method were adjusted when renaming it?) [9, 49].

The Self Encapsulate Field refactoring for example is described by two rules that are respectively shown in Algorithms 15 and 16. A result for the first rule is returned if a field `?F` has been encapsulated by adding a getter `?G` and a setter `?S` to class `?C` and if there were direct read or write accesses to field `?F`. The variable `?NR` is bound to a collection of encapsulated references to field `?F`. The metric is calculated by dividing the number of encapsulated references by the total number of direct references. A result for the second rule is returned if that same field `?F` has been encapsulated by adding a getter `?G` and a setter `?S` to class `?C` and if

there were no direct read or write accesses to field `?F`. In that case the metric is automatically set to 100%.

Algorithm 15 SelfEncapsulateField(?C,?F,?G,?S,?OR,?NR,?M)

```
encapsulateField(?F,?C,?G,?S),
referencesToBeEncapsulated(?C,?F,?OR),
not(emptyList(?OR)),
encapsulateReferencesToField(?C,?F,?NR),
length(?OR,?l1),
length(?NR,?l2),
procentualDivide(?l2,?l1,?M)
```

Algorithm 16 SelfEncapsulateField(?C,?F,?G,?S,?OR,?NR,?M)

```
encapsulateField(?F,?C,?G,?S),
referencesToBeEncapsulated(?C,?F,?OR),
emptyList(?OR),
equals(?M,100),
equals(?NR,?OR)
```

6.4 Case study: HotDraw

HotDraw is a *framework*, implemented in VisualWorks/Smalltalk, for building two-dimensional graphical editors (e.g. a simple painting program) [6]. A framework can be seen as a set of cooperating classes that together form a *reusable design* for a specific application. HotDraw was originally developed by Kent Beck and Ward Cunningham but has been reimplemented many times since then. It is well known for its pattern style implementation and its users are supposed to *subclass* framework classes in order to reuse the design. The HotDraw framework is selected as case study because:

- it's source code is publicly *accessible* creating the opportunity of fast experimenting with the case.
- it is a widely accepted framework and has been used for many validations. As such HotDraw can be considered a *representative* case.
- it is implemented in *Smalltalk*, the same programming language used for implementing the incremental and entity-based change management system.

Our entity-based change management system has never been used for developing real software systems. As such, for validating our system, we must somewhat

simulate how the history of a real case would look like in our change management system. This is done by reconstructing the evolutionary history as first-class changes.

Concretely, version 1.7 of HotDraw is loaded into a new image with an empty system history. That image is monitored by the implemented change management system in order to capture changes when loading HotDraw. The resulting system history mainly contains additions since all program entities of the loaded package are added to the system. Developers release new versions as they add new functionalities (e.g. a slide presentation tool) and update source code (e.g. bug fixing or optimization) resulting in version 1.25. In order to recover applied changes between versions 1.7 and 1.25, a complete system history is reconstructed by completing the following five steps:

1. The system history of HotDraw 1.7 is temporarily saved in a "backup" history *A*. The original system history is cleared.
2. HotDraw 1.25 is then loaded into the same image while being monitored by the change management system. This results in a system history consisting of almost all additions which is temporarily saved in a second "backup" history *B*.
3. All *added* program entities recorded in *A* but not in *B* are no longer valid and considered as removals. Those removals are collected in a temporary change list *R*.
4. All *added* program entities recorded in *B* but not in *A* are new program entities and they are collected in a temporary change list *N*.
5. Finally, the complete system history of HotDraw 1.25 is reconstructed by sequentially adding the changes contained in respectively history *A*, change list *R* and change list *N* to an empty system history.

Afterwards the source code of HotDraw 1.25 is cleaned up resulting in version 1.26. During the cleaning up phase, changes are captured and registered into the reconstructed system history.

6.5 Reasoning

6.5.1 Basic reasoning

This section reports on the basic reasoning we did on the HotDraw case by querying its system history for some basic information. The registered first-class changes are listed in a table according to their type. The following types are considered: addition/removal of a(n) *Package* (P), *Class* (C), *Instance Attribute* (IV), *Class Attribute* (CV), *Instance Method* (IM), *Class Method* (CM), *Function* (F) and *Local variable* (L). More fine-grained changes, obtained by dissecting a method's body, reveal useful information with respect to existing/removed links between entities. Therefore the following fine-grained types are recorded: addition/removal of an *Invocation* (I), *Read Access* of a variable (RA) and *Write Access* to a variable (WA). The results are then analyzed within the context of the evolving case.

The system history of HotDraw 1.26 contains *37719* first-class changes. Table 6.1 gives an overview of the most important types of changes. During the evolution of the HotDraw case, 73 classes were added and none of those were ever removed. In total, 1250 methods have been created while 169 were removed during HotDraw's lifetime. When developers modify a method, the change management system registers a removal of that method followed by an addition of the "new" one. This implies that only a *maximum* of 169 methods have been modified out of a total of 1250 HotDraw methods and existing methods defined in the concerning image. This is only a ratio of 13,52% or less indicating that a lot of methods were added without ever modifying. As Table 6.1 shows, a large number of read accesses and invocations were registered respectively 10974 and 9458 while only 1684 read accesses and 1678 invocations were removed. This table gives some insights to the size of HotDraw's program structure however based on only this information, it is very hard to say what the intention of the developer(s) was when evolving the software.

Type/Entity	**P**	**C**	**IV**	**CV**	**IM**	**CM**	**F**	**L**	**I**	**RA**	**WA**
Addition	8	73	158	17	1041	209	1193	753	9458	10974	969
Removal	0	0	3	0	135	34	185	331	1678	1684	96

Table 6.1: HotDraw - Basic reasoning

To support developers in locating (possible) bottlenecks, change objects can be grouped according to the package or class they affect. In this way, developers can see which packages or classes have been changed the most. This indicates that further analysis of that entity is beneficial for improving the quality of the

program structure [4]. A large number of additions to one class for example, may indicate that the class is too big and that some functionalities must be moved to other classes. Algorithms 17 and 18 show the developed rules for retrieving that kind of information. Table 6.2 shows the results for the query described in algorithm 17 and as such, it presents for each package of the HotDraw case the total number of changes affecting it. That table reveals that 76,50% of all changes affect the “HotDraw Framework” package. A closer look at that package learns that it contains the most classes of all packages justifying its large number of changes.

Table 6.3 shows the ten largest classes, with respect to the number of changes, returned by the query described in algorithm 18. The `Tool` class takes the first place in the table with a ratio of 19,36%. Almost a fifth of all registered changes affect that one class while the other 80,64% affects the other 72 classes. This high ratio for the `Tool` class may encourage developers to analyze it thoroughly. The other nine presented classes vary from 1398 to 383 changes indicating a decrease of the class size. This kind of information may also serve as an indicator for setting standards. For example, once a class has been changed over 2000 times, developers must inspect it and try to reduce its size.

Algorithm 17 changesOnPackage(?P,?COL,?T)

```
addPackage(?P,?),
findall(?CH,and(change(?CH),equals(?P,[?CH packageName])),?COL),
length(?COL,?T)
```

Algorithm 18 changesOnClass(?C,?COL,?T)

```
addClass(?C,?),
findall(?CH,and(change(?CH),equals(?P,[?CH className])),?COL),
length(?COL,?T)
```

	Package	Changes
1	HotDraw Framework	28856
2	HotDraw Tool Development	2859
3	HotDraw Animated Examples	2127
4	HotDraw Drawing Inspector	1296
5	HotDraw Slides	1075
6	HotDraw HotPaint	1003
7	HotDraw PERT Chart	426
8	HotDraw Animation Framework	77

Table 6.2: HotDraw - Changes per package

	Class	Changes
1	Tool	7303
2	LineFigure	1398
3	Drawing	1278
4	Figure	964
5	TextFigure	870
6	ObjectFigure	823
7	DrawingEditor	740
8	SlideFigure	512
9	ObjectWorldFigureModel	386
10	PolyLineFigure	383

Table 6.3: HotDraw - Changes per class

6.5.2 Advanced reasoning

This section reasons at an advanced level about the HotDraw case by querying its system history for change patterns. More specifically, we launch the described refactoring intensions from Section 6.3: *Self Encapsulate Field*, *Pull Up Field*, *Push Down Field*, *Move Field*, *Rename Method*, *Add Parameter* and *Remove Parameter*. Table 6.4 gives a summary of the number of detected refactorings in the system history representing the evolution until HotDraw 1.26. Reasoning about these refactorings reveals information about asserted coding conventions, irregularities in design, debugging opportunities and recovered programmer's intentions.

Refactoring	Total
Self Encapsulate Field	50
Pull Up Field	2
Push Down Field	0
Move Field	0
Rename Method	24
Add Parameter	2
Remove Parameter	0

Table 6.4: HotDraw - Advanced reasoning

Coding conventions As shown in Table 6.4, fifty Self Encapsulate Field refactorings were detected. The first ten results are shown in Table 6.5 which has the following columns: *class*, *field*, *getter*, *setter*, *old refs*, *new refs* and *metric* which are explained by using an example. As seen in Table 6.5, the `Handle` class has an `owner` field returned by the `owner` method and set by the `owner:` method. The query also returns information about old and new references respectively to the field and to one of its accessing methods. For the `owner` field of the `Handle` class, there is one direct access found in the `postCopy` method of the `Handle` class. In this case the query did not return any new references meaning that the direct access has never been replaced by an invocation to the `owner:` method. Therefore this refactoring has a 0% metric indicating that the applied refactoring was not fully completed. Actually all detected Self Encapsulate Field refactorings generate a 0% metric. A closer look at the case study reveals however that other classes do use the provided getter and setter methods. This indicates that developers probably have followed the following *coding convention*: within the defining class of a field, direct accesses are allowed. Outside that class, getters and setters must be used.

Irregularities Table 6.1 reveals an *irregularity*: 175 fields were added to the HotDraw case while only 50 of them were encapsulated by using the Self Encapsulate Field refactoring (Table 6.4). This shows that not all fields are properly encapsulated by using pure getters and setters. Renaming a method is necessary when the name of that method does not reveal its intention. Table 6.4 reveals that twenty four Rename Method Refactorings were detected in the system history of HotDraw. The first seven are shown in Table 6.6 which owns following column headers: *class*, *old name*, *new name*, *old refs*, *new refs* and *metric*. The different column headers are explained by using an example. As seen in Table 6.6, the `drawing` method has been renamed to `currentDrawing` in all classes implementing that method. The same counts for the `figureAt:` method which has been renamed to `figureAtPoint:`. The query returns information about old and new invocations to respectively the old and new name of the concerned method. In the case of the renamed `drawing` method, there were 38 possible invocations of the old method regardless its containing class. All 38 invocations were modified in order to invoke the renamed method resulting in a 100% metric for all found results. This shows that no irregularities were found with respect to this refactoring. Probably, those refactorings were applied by means of an automated refactoring tool.

Class	Field	Getter	Setter	Old refs	New refs	Metric
ToolbarView	horizontal	horizontal	horizontal:	1	0	0%
LineAnnotation	isFilled	isFilled	isFilled:	1	0	0%
CircleAnnotation	radius	radius	radius:	1	0	0%
ViewAdapterFigure	component	component	component:	2	0	0%
Figure	model	model	model:	1	0	0%
ArrowAnnotation	width	width	width:	1	0	0%
ArrowAnnotation	length	length	length:	1	0	0%
DrawingEditor	drawing	drawing	drawing:	2	0	0%
Handle	toolState	toolState	toolState:	1	0	0%
Handle	owner	owner	owner:	1	0	0%

Table 6.5: HotDraw - Performed Self Encapsulate Field Refactorings

Class	Old name	New name	Old refs	New refs	Metric
Figure	drawing	currentDrawing	38	38	100%
SlideFigure	drawing	currentDrawing	38	38	100%
Drawing	drawing	currentDrawing	38	38	100%
DrawingEditor	drawing	currentDrawing	38	38	100%
Tool	drawing	currentDrawing	38	38	100%
Figure	figureAt:	figureAtPoint:	4	4	100%
Drawing	figureAt:	figureAtPoint:	4	4	100%

Table 6.6: HotDraw - Performed Rename Method Refactorings

Debugging information As shown in Table 6.4, two Pull Up Field refactorings were detected. Table 6.7 contains the details about those refactorings. This refactoring pulls up a field from a set of subclasses defining that field. The "Pulled up from" contains the actual classes of which the field is pulled up. This is used for calculating the metric. The first result of this query indicates that the index field was defined in two subclasses `IndexedTrackHandle` and `TentativePositionHandle`. The found result has a 50% metric indicating that not all fields were removed in the set of Pull up from classes. The detected refactoring shows from which class it has been pulled up, in this particular case from the `IndexedTrackHandle` class. Based on this information, developers can recover why the refactoring was not completely performed easing its correction.

Field	Pull up from	Pull up to	Pulled up from	Metric
index	IndexedTrackHandle TentativePositionHandle	TrackHandle	IndexedTrackHandle	50%
owner	Handle	Figure	Handle	100%

Table 6.7: HotDraw - Performed Pull up Field Refactorings

Intentions The results for the other refactoring queries are similar to those explained and further details are omitted in this text. Based on the nature of the applied refactorings and the results obtained by the queries, developers probably wanted to *improve the understandability* (e.g. Rename Method or Add Parameter) of the code, to *decrease the amount of duplicated code* (e.g. Pull Up Field) and to *raise the level of abstraction* (e.g. Self Encapsulate Field). These results align with the idea behind refactorings: improving the quality of source code without modifying the program's behavior.

6.6 Conclusion

This chapter validates if the incremental and entity-based change management system really offers *accurate and useful information* for reasoning about the evolution of a software system. That validation is done by trying to recover the intention a programmer had when developing a software system.

The system history of an evolving program is captured by using an incremental and entity-based change management system. That history is expressed by a repository retaining *first-class change objects*. Information held in that repository can be consulted by looking for *change patterns*. A change pattern is a set of rules to which a group of changes adheres in order to form a solution for that pattern. This is the technique asserted in *declarative programming languages* which describe a problem rather than defining a solution for solving it. Section 6.2 elaborates on the *SOUL* language and why it is chosen as reasoning language.

Section 6.3 introduces queries describing patterns for recovering intensions from a system history. Basic intension queries are small queries that can be used to build more complex ones. The `change(?CH)` predicate is an example of such a query, it returns one or more changes contained in the available system history. Bigger blocks can be built in order to describe patterns expressing specific types of changes (e.g. `addMethod(?C,?M,?CH)` which returns one or more `AddMethod` change objects). Those bigger blocks can be used to describe more complex patterns (e.g. one that returns all added getter methods) which in their turn can be combined to describe *higher-level change patterns*. One specific category of higher-level patterns concerns refactorings which improve the quality of a system's internal structure without modifying its behavior.

HotDraw is chosen as case and explained in Section 6.4. That section describes two evolution scenarios applied to the HotDraw case. The first scenario concerns adding new functionalities and updating source code. The complete system his-

tory is reconstructed by starting from two preliminary versions 1.7 and 1.25 and recovering the applied changes between those versions. How this is done, is explained in section 6.4. Afterwards the source code of HotDraw is cleaned up.

Section 6.5 presents examples of both basic and advanced reasoning about the evolution history of HotDraw. Basic reasoning reveals details about the size of the program as well as which are its crucial parts. Advanced reasoning allowed the detection of coding conventions, irregularities, debugging information and the recovering of some programmers intentions.

This validation shows that the implemented change management system offers a lot of evolutionary information that can be used for reasoning about the monitored program. In practice however, it would be better to use that change management system from the moment developers start with a project. The system history would then contain the system's entire evolution allowing researchers to draw more precise conclusions and to predict evolution with greater accuracy.

Chapter 7

Conclusion and future work

7.1 Conclusion

The main goal of this research was to support *reasoning about software evolution*. In order to reason about the evolution of software systems, researchers require *evolutionary information*. One way to acquire that information is by inspecting and analyzing the artifacts stored at the *repository* of a *change management system*. Such a repository is a centralized library which is maintained by that change management system (e.g. files containing source code). The thesis of this dissertation is that *an incremental and entity-based change management system with first-class changes based on a meta-model offers accurate evolutionary information and supports reasoning about software evolution*. This dissertation provides four contributions which are discussed below:

1. Chapter 2 defines a ***taxonomy of change management systems*** in the context of software evolution. The categorization is based on two dimensions: *time* (when are changes stored) and *structure* (how are changes stored). The different categories are evaluated with respect to the following criteria: *complete information*, *language independence for monitoring changes and reasoning*, *developer independence*, *order preservation* and *hooks for extensibility for monitoring changes and reasoning*. Incremental and entity-based change management systems were found to be best suited for our purpose. An incremental change management system continuously captures and stores changes as they are applied by the developers. Entity-based change management systems store change information about program entities. A program entity is a "building block" of a programming language

(e.g. a class or method). A *meta-model* for programming languages describes their available building blocks.

2. Chapter 3 establishes a ***taxonomy for meta-models*** which are capable of modeling building blocks of class-based object-oriented software. The exploration of alternative meta-models is supported by the following criteria: *support for multiple object-oriented languages*, *extensibility hooks*, *derivable system invariants* and *ease of information exchange*. With respect to the evaluated criteria, *FAMIX*, a language independent meta-model for modeling object-oriented software, turns out to be most suited.

3. The FAMIX meta-model is ***extended*** in order to model (a) the *dynamic state* of a program and (b) *Smalltalk specific features* enabling to derive first-class changes with respect to those aspects. An example concerning the dynamic state is capturing the creation of an instance. A language specific feature of Smalltalk is its block closure mechanism allowing anonymous functions. Changes with respect to block closures can be captured by the change management system.

4. In chapter 4, a ***conceptual model of changes*** is established based on the FAMIX meta-model. Changes to a program are expressed as *first-class objects* which may be stored in a variable or data structure, may be passed as an argument to a function and may be returned as the value of a function. The conceptual model is implemented as a *class hierarchy* of which each class represents a type of change. Each change class is designed so that a change object can answer *what* it represents, *why* it exists, *when* it applies, *where* it applies and *how* it applies. The language independence of FAMIX implies that the conceptual model of first-class changes is also *language independent* enabling *language independent reasoning*.

The change management system as well as the obtained change hierarchy are implemented in Smalltalk. The functionality for capturing changes is implemented by integrating it into a *basic change framework* provided by VisualWorks for Smalltalk. Once the change management system is deployed in order to monitor an evolving program, each change is registered into the system history as a first-class change. As such, program histories can be composed in an incremental and entity-based way. The change information can be consulted by exploring the system history for *change patterns*. Such a pattern is represented by a set of rules to which a group of changes adheres. *SOUL*, a *declarative programming language* was used to reason about histories. Intensions can be recovered from program histories by using SOUL queries. Such queries can be composed to build bigger blocks and more complex patterns eventually leading to *high-level change patterns*. One example of high-level patterns concerns *refactorings* which

are used to improve the quality of a systems internal structure without modifying its behavior.

We validate our approach by reasoning about the evolution history of a real-life system: HotDraw. The validation process reveals basic information about the size of the system as well as which are its crucial parts. More advanced reasoning offers information about asserted coding conventions, irregularities in the design, debugging opportunities and recovered programmer's intentions.

7.2 Future work

Different applications of support for software evolution may benefit from our incremental and entity-based change management system. We believe that an implementation of a (semi-automatic) *program generator* would open many doors as developers sometimes need to convert software systems to another implementation language. It would allow to take as input an entire system history, to process it for converting language specific aspects (e.g. `self` vs `this`) and to output it to the programming environment of interest. Its change management system can then take as input the processed changes and sequentially apply them to create the same program as the original one. Another application would be the *merging* of two system histories resulting into one single history however when merging, conflicts can arise which could introduce a lot of errors in the program increasing debugging time. Therefore the merging process can be supported by *conflict detection* and/or *resolution* reducing programmers' effort for locating and solving errors.

Currently, the number of implemented change patterns is limited. For this research work, seven refactorings are described by using SOUL rules and queries. Providing more refactoring queries would increase the chance of recovering more intensions from program histories. We also see opportunities in other high-level changes such as *composite*, *domain-specific* or *intensional* changes which all raise the level of abstraction. A composite change is a high-level change composing atomic and/or composite changes. Just like atomic changes, they are available to developers and researchers for querying, applying or undoing itself. A domain-specific change is a composite change with an abstract goal within a particular domain. An intensional change is a description of a group of changes which can be evaluated.

This research work focusses on first-class changes expressing changes to program entities of *object-oriented languages*. It might be interesting to experiment

with the idea of first-class changes for support in software evolution in the context of other programming paradigms (e.g. component-oriented, functional or logical programming languages). This work is situated in the field of software evolution where lot of systems are developed in a component-oriented way hence component-oriented development is an interesting area to investigate.

A final track of future work concerns thorough experimentation with our change management system. One idea of a complete and independent validation is to assign students or employees with the development of a small tool in the VisualWorks for Smalltalk environment. In this way, researchers have no idea what developers have been doing and have to recover programmer's intentions independently from the developed tool.

Bibliography

[1] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Proceedings of Annual Reliability and Maintainability Symposium*. IEEE, 2002.

[2] S. Ambler. *The Object Primer Third Edition Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2004.

[3] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 73–87. ACM Press, 2000.

[4] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman. Does god class decomposition affect comprehensibility? In *Proceedings of the IASTED International Conference on Software Engineering*, pages 346–355. IASTED/ACTA Press, February 2006.

[5] G. Booch, I. Jacobson, and J. Rumbaugh. The unified modelling language for object-oriented development. Documentation set, version 0.9, Rational Software Corporation, 1996.

[6] J. M. Brant. Hotdraw. Master's thesis, University of Illinois, 1995.

[7] P. Cederqvist. *Version Management with CVS*. Free Software Foundation, 2004.

[8] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly & Associates, 2004.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–177. ACM Press, 2000.

[10] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne, 1999.

[11] S. Ducasse and S. Demeyer. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, 1999.

[12] P. Ebraert, E. V. Paesschen, and T. DHondt. Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel, 2007.

[13] J. Estublier. Software configuration management: a roadmap. In *ICSE - Future of Software Engineering Track*, pages 279–289, 2000.

[14] G. Florijn. RevJava: Design critiques and architectural conformance checking for java software. *SERC*, 2002.

[15] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal Of Software Maintenance And Evolution: Research And Practice*, 2006.

[18] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 117–119. ACM Press, 2002.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addision-Wesley, 1983.

[20] O. M. Group. Unified modeling language 1.3. Technical report, Rational Software Corporation, June 1999.

[21] B. Henderson-Sellers. Some problems with the UML 1.3 meta-model. In *Proceedings of the 34th Hawaii International Conference on System Sciences*. IEEE Computer Society, 2001.

[22] A. Herranz-Nieva and J. J. Moreno-Navarro. Generation of and debugging with logical pre and post conditions. *Automated and Algorithmic Debugging*, 2000.

[23] L. Hochstein and M. Lindvall. Diagnosing architectural degeneration. In *Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop*, 2003.

[24] R. A. Kowalski. *Logic for Problem Solving*. Elsevier North Holland, 1979.

[25] R. A. Kowalski. Prolog as a logic programming language. Technical report, Department of Computing, Imperial College, 1981.

[26] M. M. Lehman and B. Belady. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[27] M. M. Lehman and J. F. Ramil. Evolution in software and related areas. In *Proceedings of the fourth International Workshop on Principles of Software Evolution*. ACM Press, 2001.

[28] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 2001.

[29] M. M. Lehman and J. F. Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1-4):275–309, 2002.

[30] M. M. Lehman and J. F. Ramil. Software evolution: background, theory, practice. *Information Processing Letters*, 2003.

[31] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software. In *Proceedings of the Eighth International Symposium on Software Metrics*, pages 77–86. IEEE Computer Society, 2002.

[32] E. Lippe and N. V. Oosterom. Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, volume 17, pages 78–87. ACM Press, 1992.

[33] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, second edition, 1987.

[34] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[35] P. Maes. Concepts and experirnents in computational reflection. In *Proceedings of the second Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155. ACM Press, 1987.

[36] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[37] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic notes in Theoretical Computer Science*, 72(2):12, 2002.

[38] R. Mittermeir. Facets of software evolution, 2006.

[39] Mogware. Filehamster - a personal revision control solution for content creators. `http://www.mogware.com/FileHamster/`, 2006. [Last accessed 2 June 2007].

[40] G. Reggio and R. Wieringa. Thirty one problems in the semantics of UML 1.3 dynamics. 1999.

[41] R. Robbes. Mining a change-based software repository. In *Proceedings of Fourth International Workshop on Mining Software Repositories*, 2007.

[42] R. Robbes and M. Lanza. Versioning systems for evolution research. In *Proceedings of Eighth International Workshop on Principles of Software Evolution*, pages 155–164. IEEE Computer Society, 2005.

[43] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, 2007.

[44] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *Proceedings of Tenth International Conference on Fundamental Approaches to Software Engineering*, pages 27–41, 2007.

[45] R. Robbes and M.Lanza. Change-based software evolution. In *Proceedings of EVOL 2006 (First International ERCIM Workshop on Software Evolution)*, pages 159–164, 2006.

[46] J. A. Robertson. Why smalltalk? Could smalltalk be the next big business language. *PC AI*, 15(2):24–26, 2001.

[47] A. Rosdal. Empirical study of software evolution and architecture in open source software projects. Technical report, Norwegian University of Science and Technology, 2005.

[48] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, pages 126–130. IEEE Computer Society, 2003.

[49] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001.

[50] A. Software. AJC active backup. `http://www.ajcsoft.com/AJCActBk.php`, 2007. [Last accessed 2 June 2007].

[51] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of FAST 2003: second USENIX Conference on File and Storage Technologies*, 2003.

[52] C. Systems. *VisualWorks - Application Developer's Guide*. 1993-2005.

[53] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

[54] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society, 1998.

Appendix A

Invariants for the FAMIX meta-model and its extensions

A.1 FAMIX meta-model

Object

- Each `Object` maintains maximum one `sourceAnchor`.
- Each `Object` has maximum one `commentsAt`.

Property

- Each `Property` maintains exactly one `name`.
- The `name` of a `Property` is unique for all properties of a single `Object`.
- Each `Property` has exactly one `value`.

Entity

- Each `Entity` maintains exactly one `name`.
- Each `Entity` maintains exactly one `uniqueName`.
- The `uniqueName` of an `Entity` is unique for all entities in the model.

Package

- Each `Package` belongs to maximum one `Package`.

Class

- Each `Class` maintains maximum one `isAbstract`.
- Each `Class` belongs to maximum one `Package`.

BehaviouralEntity

- Each `BehaviouralEntity` has maximum one `accessControlQualifier`.
- Each `BehaviouralEntity` maintains exactly one `signature`.
- The concatenation of `signature` and `belongsTo` uniquely distinguishes a `BehaviouralEntity`.
- A `signature` includes the `name` of the `BehaviouralEntity`.
- Each `BehaviouralEntity` owns maximum one `isPureAccessor`.
- Each `BehaviouralEntity` maintains maximum one `declaredReturnType`.
- Each `BehaviouralEntity` declares maximum one `returnClass`.

Method

- Each `Method` belongs to exactly one `Class`.
- Each `Method` has maximum one `hasClassScope`.
- Each `Method` owns maximum one `isAbstract`.
- Each `Method` maintains maximum one `isConstructor`.

Function

- Each `Function` belongs to maximum one `Package`.
- The concatenation of the `name` of the containing `Package` and the `name` of the `Function` is unique within the model.

StructuralEntity

- Each `StructuralEntity` has maximum one `declaredType`.
- Each `StructuralEntity` is declared as maximum one `Class`.

Attribute

- Each `Attribute` belongs to exactly one `Class`.
- Each `Attribute` maintains maximum one `accessControlQualifier`.
- Each `Attribute` maintains maximum one `hasClassScope`.

GlobalVariable

- Each `GlobalVariable` belongs to maximum one `Package`.
- The concatenation of the `name` of the containing `Package` and the `name` of the `GlobalVariable` is unique within the model.

ImplicitVariable

- Each `ImplicitVariable` maintains maximum one `belongsToContext`.
- The concatenation of `belongsToContext` and the `name` of the variable is unique within the model.

LocalVariable

- Each `LocalVariable` belongs to exactly one `BehaviouralEntity`.

FormalParameter

- Each `FormalParameter` belongs to exactly one `BehaviouralEntity`.
- Each `FormalParameter` owns exactly one `position`.
- Each `position` is unique within the `BehaviouralEntity`'s parameter list.

InheritanceDefinition

- Each `InheritanceDefinition` has exactly one `subclass`.
- Each `InheritanceDefinition` has exactly one `superclass`.
- Each `InheritanceDefinition` maintains maximum one `accessControlQualifier`.
- Each `InheritanceDefinition` owns maximum one `index`.
- If provided, `index` is unique within the list of superclasses of a particular class.
- `index` is `null` when multiple inheritance is not allowed.

Access

- Each `Access` accesses exactly one `StructuralEntity`.
- Each `Access` is accessed in exactly one `BehaviouralEntity`.
- Each `Access` owns maximum one `isAccessLValue`

Invocation

- Each `Invocation` is invoked by exactly one `BehaviouralEntity`.
- Each `Invocation` refers to minimal one `candidate` `BehaviouralEntity`.
- Each `Invocation` refers to maximum one `base` `Entity`.
- The concatenation of `signature` and the `name` of `base` forms a unique name of the invoked behavioral entity.

Argument

- Each `Argument` maintains exactly one `position`.
- `position` is unique within the corresponding list of arguments.
- Each `Argument` has exactly one `isReceiver`.

AccessArgument

- Each `AccessArgument` maintains exactly one `Access`.

ExpressionArgument

- Each `ExpressionArgument` maintains exactly one `Invocation`.

A.2 Extension for dynamic state

Instance

- Each `Instance` is instantiated of exactly one `Class`.
- Each `Instance` has maximum as many attribute values as attributes of the `Class` of which the instance is instantiated.

AttributeValue

- Each `AttributeValue` maintains exactly one `valueFor`.
- Each `AttributeValue` has exactly one `Instance` as `value`.

GlobalVariable

- Each `GlobalVariable` maintains exactly one `valueFor`.
- Each `GlobalVariable` has exactly one `Instance` as `value`.

A.3 Extension for Smalltalk

Class

- Each `Class` has exactly one `isMetaClass`.

BehavioralEntity

- Each `BehaviouralEntity` stores the "Object" `Qualifier` as value for `declaredReturnType`.
- Each `BehaviouralEntity` returns the `Object Entity`.

Method

- Each `Method` belongs to exactly one `Package`.
- Each `Method` belongs to exactly one `Protocol`.

StructuralEntity

- Each `StructuralEntity` stores the "Object" `Qualifier` as value for `declaredType`.
- Each `StructuralEntity` is declared as the `Object Entity`.

Attribute

- Each `Attribute` maintains maximum one `initializationValue`.
- Each `Attribute` stores the "protected" `Qualifier` as value for `accessControlQualifier`.

GlobalVariable

- Each `GlobalVariable` maintains maximum one `initializationValue`.

LocalVariable

- Each `LocalVariable` maintains exactly one `position`.
- `position` is unique within the corresponding list of temporary arguments.

InheritanceDefinition

- Each `subclass` inherits from exactly one `superclass`.

Appendix B

Design of first-class changes

B.1 Base model

This section describes the change hierarchy directly derived from the FAMIX meta-model as explained in chapter 3. Some change classes contain extra information (e.g. `MethodChange` has a `body` attribute) even when that information can be retrieved from the system history. That extra information however can be useful for speeding up the reasoning process about the evolution of a system.

B.1.1 Change

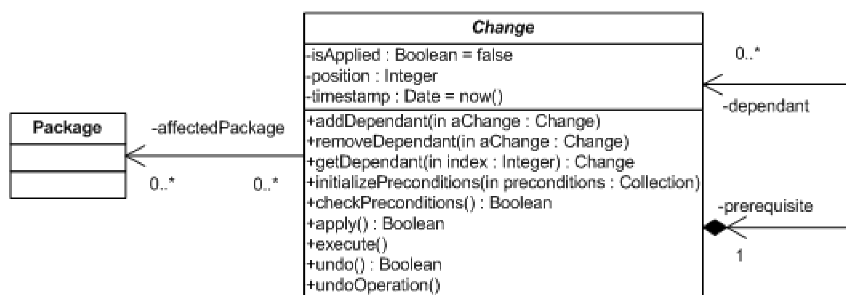

Figure B.1: Design - Change

What Figure B.1 shows the abstract `Change` class which is the root class of the change hierarchy. Each change object (a `Change` instance) influences one or more `Entity` objects which may be contained in packages. In order to maintain

clarity among the system history, changes own an `affectedPackage` association. This allows developers or researchers to group changes according to the package they affect. A first-class change also knows date and time of its creation, represented by the `timestamp` attribute. All changes in the hierarchy are *atomic* meaning they can not be split up however some changes contain information that can be analyzed in order to obtain more useful information (e.g. the arguments of an invocation). Therefore it is allowed that such a change object (`prerequisite`) contains changes (`dependant`) expressing that information. To support that, the `Change` class provides following (abstract) methods:

- `addDependant`: adds a `Change` object as `dependant`.
- `removeDependant`: removes an existing `Change` `dependant`.
- `getDependant`: retrieves the `Change` `dependant` kept at a certain `index`.

Thus each `Change` subclass overriding those methods, allows its instances to add or remove dependant changes and to manage them. Dependants maintain a reference to their `prerequisite` change object and their `position` in their `prerequisite`'s collection of dependants.

How & Where The `Change` class can not be instantiated because it is an abstract class. As such it can not be applied or undone even if it provides the apply/undo mechanism to its subclasses. The `initializePreconditions` method is defined in the `Change` class and it initializes an empty list of preconditions. Each subclass may override that method to add preconditions. The `apply` method of the `Change` class specifies the necessary actions in order to complete the (re-)application of a change. Algorithm 19 shows the general steps of that `apply` method. Firstly, the `apply` method checks if all specified preconditions are satisfied. If so, the `execute` method is called: each change is applied differently and may override the standard `execute` method provided by the `Change` class. Afterwards the `isApplied` flag is switched on and is returned. If one or more preconditions are not met by the system, the apply functionality returns `false`. The undo mechanism provides a `compensating action` for undoing changes and is similar to algorithm 19. A simple example reveals how compensating actions can be used. To undo the creation of a method, that method needs to be removed from the system (the *compensating action* of a method creation). The `isApplied` attribute is a `Boolean` indicating whether or not the change is applied to the current state of the program structure.

Algorithm 19 Generic apply method

```
if checkPreconditions then
    execute
    isApplied ← true
end if
return isApplied
```

B.1.2 EntityChange

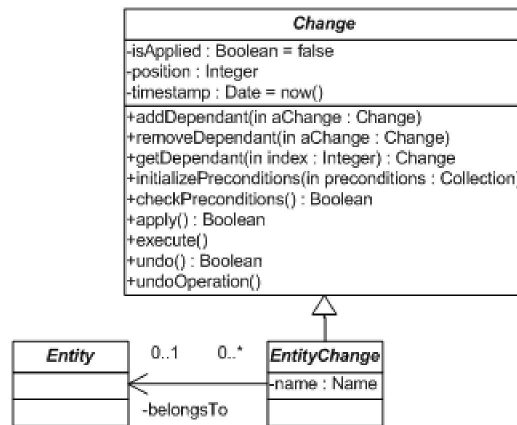

Figure B.2: Design - EntityChange

What Figure B.2 shows the abstract `EntityChange` class that inherits from the `Change` class. It expresses a change to some entity (e.g. a `class` or `method`) and stores the `name` of the involved `Entity`. It also maintains a reference to the *containing entity* of the changed entity. An example clarifies this: a class is defined in some package P and that class may be changed during its lifetime. Then any change to that class is expressed by an `EntityChange` object and `belongsTo` refers to package P.

When

- `name` is not empty
- Each `EntityChange` belongs to maximum one `Entity`

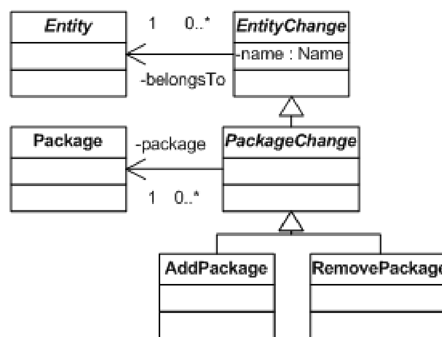

Figure B.3: Design - PackageChange

B.1.2.1 PackageChange

What Figure B.3 depicts the abstract `PackageChange` class and its subclasses. As seen in the figure, the `PackageChange` class inherits from the `EntityChange` class. A `PackageChange` represents a change with regard to some `Package` (e.g. the creation of a new package) which is expressed by the `package` association. Packages may belong to other packages denoted by the inherited `belongsTo` association of the `EntityChange` class. The subclasses of `PackageChange` are discussed in the following two sections.

B.1.2.2 AddPackage

What The `AddPackage` class is used to represent the addition of a new empty package.

How & Where There are two possible scenarios to add a new package to a program structure. First of all, a new package can be added to the root thus not to another package. To accomplish this, the following action is taken: *Add package P*. Secondly, a new package can be added to another package by performing the next action: *Add a package P to package E.*

When

- A `Package` with `name` P does not exist (Scen. 1 & 2)
- A `Package` with `name` E exists (Scen. 2)

B.1.2.3 RemovePackage

What The RemovePackage class is used to represent the removal of an existing empty package.

How & Where There are two possible scenarios to remove an existing package from a program structure. First of all, an existing package can be removed from the root, it does not belong to another package. This requires the next step to be executed: *Remove package P*. Secondly, an existing package can be removed from another package by performing the following action: *Remove a package P from package E*.

When

- A Package with name P exists (Scen. 1 & 2)
- Package P is empty, it does not contain any entities (Scen. 1 & 2)
- A Package with name E exists (Scen. 2)
- Package E contains package P (Scen. 2)

B.1.2.4 ClassChange

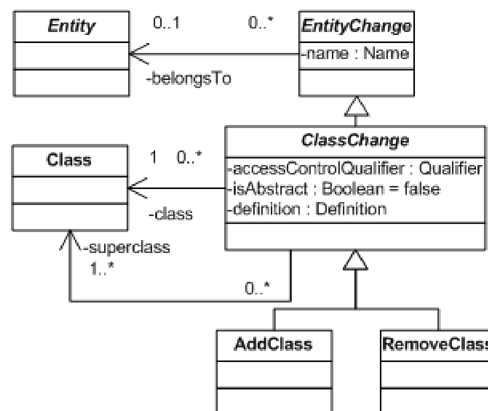

Figure B.4: Design - ClassChange

What The abstract `ClassChange` class (depicted in figure B.4) represents a change with respect to class entities (thus no other entities such as methods or functions). As seen in figure B.4, there is a `class` association between the `Class` and `ClassChange` classes. That `class` association refers to the `Class` entity manipulated by the `ClassChange` object. The `superclass` association of `ClassChange` refers to the superclass(es) of the changed class.

The `ClassChange` class owns following attributes:

- `accessControlQualifier`: the access control qualifier of the changed class (e.g. public or private).
- `definition`: the full class specification.
- `isAbstract`: a `Boolean`, initially `false`, indicating whether the changed class is defined as *abstract* or not. An abstract class can not be instantiated.

The subclasses of `ClassChange` are discussed in the following two sections.

B.1.2.5 AddClass

What The `AddClass` class represents the creation of a new class within a certain package which is maintained by the inherited `belongsTo` relationship of `EntityChange`.

How & Where To apply an `AddClass` the following actions are required:

- *Add class C to package P*
- *For each superclass S of C, add an inheritance definition I between class C and class S*

When

- A `Class` with `name` C does not exist
- A `Package` with `name` P exists
- There is at least one `superclass`
- Each specified superclass of class C exists

B.1.2.6 RemoveClass

What The `RemoveClass` class expresses the removal of an empty class from a certain package. That package is maintained by the inherited `belongsTo` relationship of `EntityChange`.

How & Where To apply a `RemoveClass` the following actions need to be performed:.

- *Remove class C from package P*
- *For each superclass S of C: remove the inheritance definition I between class C and class S*

When

- A `Class` with `name` C exists
- A `Package` with `name` P exists
- `Package` P contains `Class` C
- No methods are defined in `Class` C
- No attributes are defined in `Class` C

B.1.2.7 BehavioralEntityChange

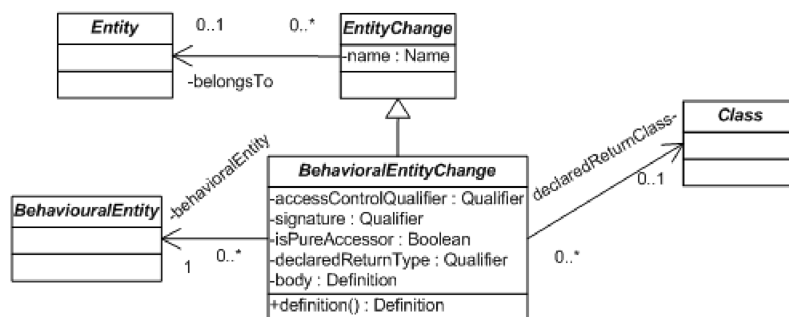

Figure B.5: Design - BehaviouralEntityChange

What Figure B.5 depicts the `BehavioralEntityChange` class which denotes a change manipulating a certain behavioral entity (`behavioralEntity` association). Each `BehavioralEntityChange` object stores the following data:

- the `signature` of the manipulated behavioral entity
- the defined access control qualifier of the manipulated behavioral entity: `accessControlQualifier`
- a `Boolean isPureAccessor` indicating if the behavioral entity is a pure accessor.

Additionally the `body` attribute stores the behavioral entity's body. Each behavioral entity returns an object thus if possible `BehavioralEntityChange` stores the concerned information in the following attribute and assocation:

- `declaredReturnType`: the type of the returned object. Typically this will be a class, a reference to an object (*pointer*) or a primitive type.
- `declaredReturnClass`: the class implicitly defined in `declaredReturnType`.

When

- `signature` is not empty

B.1.2.8 MethodChange

What As figure B.6 shows, the abstract `MethodChange` class inherits from `BehaviouralEntityChange`. It represents a change with regard to a `Method` entity and refers to that entity via the inherited `behavioralEntity` method. The `MethodChange` class owns the following attributes:

- `hasClassScope`: a `Boolean` indicating if the concerned method is defined at instance or class level.
- `isAbstract`: a `Boolean` indicating whether the modified method is defined as *abstract* or not. An abstract method can not contain a body.
- `isConstructor`: a `Boolean` expressing whether or not the method creates and initializes new instances of its containing class.

The subclasses of `MethodChange` are discussed in the following two sections.

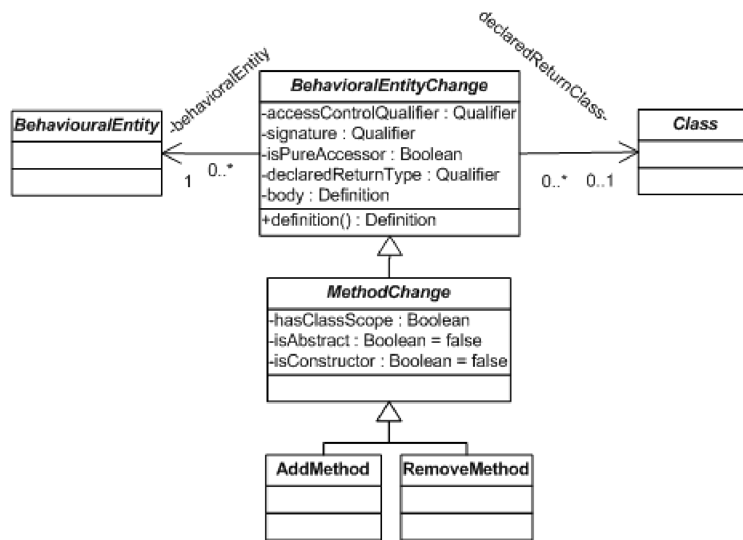

Figure B.6: Design - MethodChange

When

- `hasClassScope` is either `true` or `false`

B.1.2.9 AddMethod

What The `AddMethod` class is used to express the creation of a new method within a certain class, this method has an empty body. The concerned class is maintained by the inherited `belongsTo` relationship of the `EntityChange` class.

How & Where The application of an `AddMethod` involves the following action: *Add method M to class C.*

When

- A `Class` with `name` C exists
- `Class` C does not define a `Method` with `signature` M

B.1.2.10 RemoveMethod

What The `RemoveMethod` class represents the removal of an empty method from a class. The concerned class is maintained by the inherited `belongsTo` relationship of the `EntityChange` class.

How & Where To apply a `RemoveMethod` the following step needs to be executed: *Remove method M from class C.*

When

- A `Class` with `name` C exists
- `Class` C defines a `Method` with `signature` M
- `Method` M has an empty body

B.1.2.11 FunctionChange

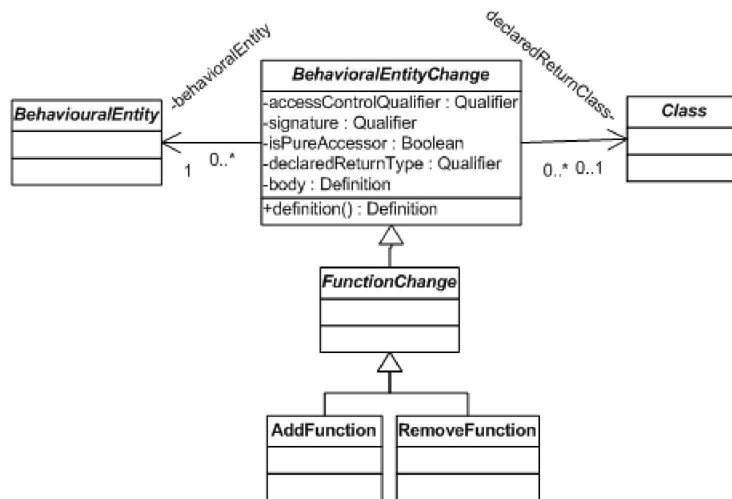

Figure B.7: Design - FunctionChange

What Figure B.7 shows that the `FunctionChange` class inherits from the `BehaviouralEntityChange` class. It represents a change with respect to a `Function` entity, its subclasses are explored in the following two sections.

B.1.2.12 AddFunction

What The `AddFunction` class expresses the creation of a new function within a certain package, the added function has an empty body. The concerned package is maintained by the inherited `belongsTo` association.

How & Where There are two possible scenarios for adding a function. First of all, a function can be added to the root directory of the system, this requires the following step: *Add function F*. Secondly, a function can be added to an existing package by performing the following action: *Add function F to package P*.

When

- A `Function` with `signature` F does not exist (Scen. 1 & 2)
- A `Package` with `name` P exists (Scen. 2)

B.1.2.13 RemoveFunction

What The `RemoveFunction` class is used to represent the removal of an empty function from a package. The concerned package is maintained by the inherited `belongsTo` association.

How & Where There are two possible scenarios for removing a function. Firstly, a function can be removed from the root of the program structure, this requires the following action: *Remove function F*. Secondly, a function can be removed from a package by performing the following step: *Remove function F from package P*.

When

- A `Function` with `signature` F exists (Scen. 1 & 2)
- A `Package` with `name` P exists (Scen. 2)
- `Package` P contains `Function` F (Scen. 2)

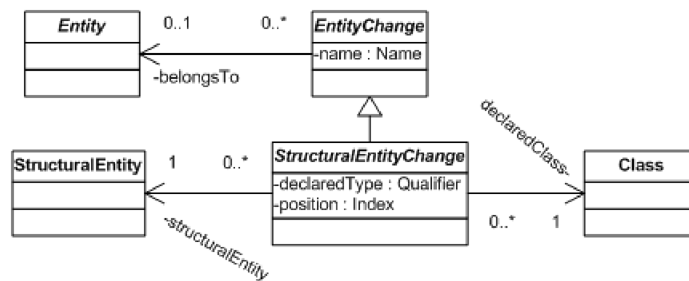

Figure B.8: Design - StructuralEntityChange

B.1.2.14 StructuralEntityChange

What Figure B.8 depicts the abstract `StructuralEntityChange` class, expressing a change related to a particular structural entity denoted by the `structuralEntity` association. The figure shows that the `StructuralEntityChange` class inherits from the `EntityChange` class. It maintains the following attributes and associations:

- `declaredType`: the type of the concerned structural entity. Typically this will be a class, a pointer or a primitive type.
- `declaredClass`: the class implicitly defined in `declaredType`.
- `position`: indicates the structural entity's position in the defining entity.

B.1.2.15 AttributeChange

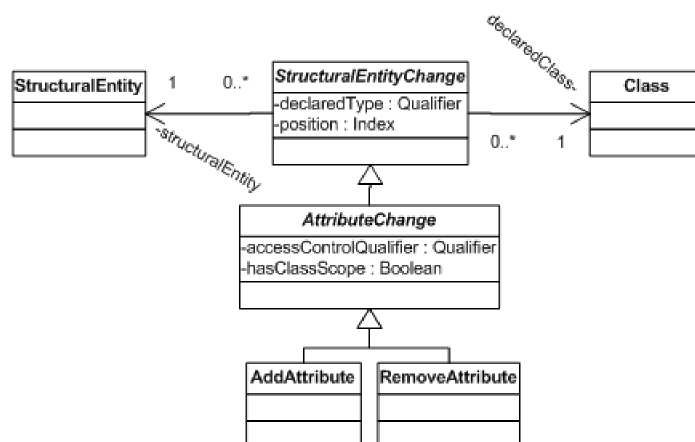

Figure B.9: Design - AttributeChange

What The abstract AttributeChange class is depicted in figure B.9. It inherits from StructuralEntityChange and expresses a change influencing a class field. The inherited structuralEntity association refers to the manipulated field. The AttributeChange class owns the following fields:

- accessControlQualifier: represents the accessor assigned to the concerned structural entity.
- hasClassScope: indicates whether the concerned field is defined at instance-level or class-level.

As figure B.9 shows, AttributeChange acts as a superclass for the classes discussed in the following two sections.

When

- hasClassScope is either true or false

B.1.2.16 AddAttribute

What The AddAttribute class is used to express the addition of an attribute to an existing class.

How & Where To accomplish the application of an AddAttribute the following step is required: *Add attribute A to class C*.

When

- A Class with name C exists
- Class C does not define an Attribute with name A

B.1.2.17 RemoveAttribute

What The RemoveAttribute class represents the removal of an attribute from an existing class.

How & Where To accomplish the application of a RemoveAttribute the following action is required: *Remove attribute A from class C*.

When

- A `Class` with `name` C exists
- `Class` C defines an `Attribute` with `name` A

B.1.2.18 GlobalVariableChange

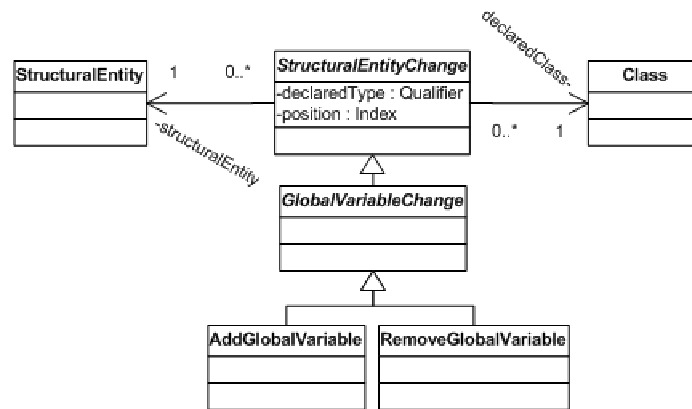

Figure B.10: Design - GlobalVariableChange

What The abstract `GlobalVariableChange` class is depicted in figure B.10. It inherits from `StructuralEntityChange` and expresses a change influencing a global variable. The manipulated variable is maintained by the inherited `structuralEntity` relationship. The `GlobalVariableChange` class owns no fields or relationships. As figure B.10 shows, `GlobalVariableChange` has subclasses which are discussed in the following two sections.

B.1.2.19 AddGlobalVariable

What The `AddGlobalVariable` class is used to express the addition of a global variable to a program structure. If it is added to a package, then that package is referred to via the inherited `belongsTo` association of `EntityChange`.

How & Where There are two possible scenarios to add a new global variable to a program structure. Firstly, a new global variable can be added to the root thus not to any package. To accomplish this, the following action is undertaken: *Add*

global variable G. Secondly, a new global variable can be added to a package by performing the next action: *Add global variable G to package P*.

When

- A `GlobalVariable` with `name` G does not exist (Scen. 1 & 2)
- A `Package` with `name` P exists (Scen. 2)

B.1.2.20 RemoveGlobalVariable

What The `RemoveGlobalVariable` class is used to express the deletion of a global variable from a program structure. If there is a containing package, then this is maintained via the inherited `belongsTo` relationship of `EntityChange`.

How & Where There are two possible scenarios to remove an existing global variable from a system. First of all, an existing global variable can be removed from the root thus not from any package. To accomplish this, the following action is undertaken: *Remove global variable G*. Secondly, a global variable can be removed from a package by performing the next action: *Remove global variable G from package P*.

When

- A `GlobalVariable` with `name` G exists (Scen. 1 & 2)
- A `Package` with `name` P exists (Scen. 2)
- Package P contains global variable G (Scen. 2)

B.1.2.21 FormalParameterChange

What Figure B.11 depicts the abstract `FormalParameterChange` class. It inherits from `StructuralEntityChange` and expresses a change influencing a formal parameter defined by a behavioral entity. The manipulated parameter is maintained by the inherited `structuralEntity` relationship. The `FormalParameterChange` class has no fields or relationships. As figure B.11 shows, `FormalParameterChange` acts as a superclass for the classes discussed in the following two sections.

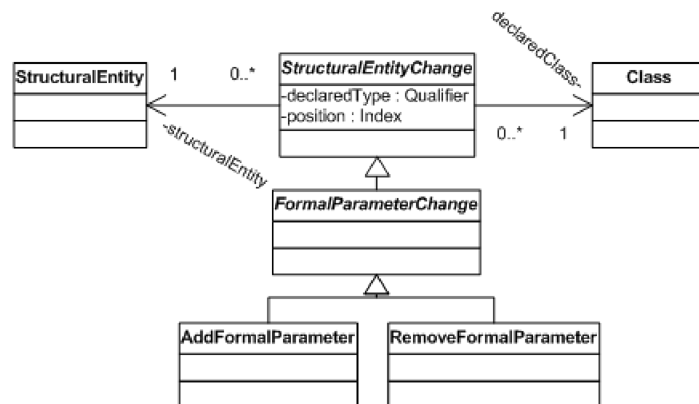

Figure B.11: Design - FormalParameterChange

B.1.2.22 AddFormalParameter

What The `AddFormalParameter` class is used to express the addition of a formal parameter to an existing behavioral entity. This behavioral entity is maintained by the inherited `belongsTo` relationship of `EntityChange`.

How & Where The application of an `AddFormalParameter` requires the following action: *Add formal parameter F to behavioral entity B.*

When

- A `Class` with `name` C defines a `BehaviouralEntity` with `name` B
- `BehaviouralEntity` B does not define a `FormalParameter` with `name` F

B.1.2.23 RemoveFormalParameter

What The `RemoveFormalParameter` class is used to express the deletion of a formal parameter from a behavioral entity. This behavioral entity is maintained by the inherited `belongsTo` association of `EntityChange`.

How & Where The application of a `RemoveFormalParameter` requires the following action: *Remove formal parameter F from behavioral entity B.*

When

- A `Class` with `name` C defines a `BehaviouralEntity` with `name` B
- `BehaviouralEntity` B defines a `FormalParameter` with `name` F

B.1.2.24 LocalVariableChange

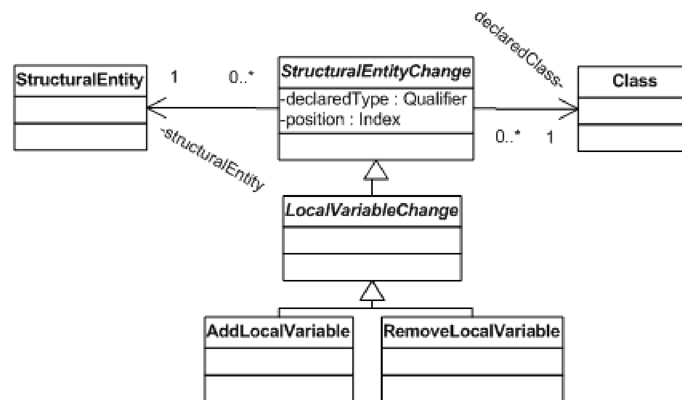

Figure B.12: Design - LocalVariableChange

What The abstract `LocalVariableChange` class is shown in figure B.12. `LocalVariableChange` inherits from `StructuralEntityChange` and represents a change influencing a local variable defined in a behavioral entity. That behaviorial entity is referred to via the inherited `belongsTo` association. The `LocalVariableChange` class has no fields or relationships. Its subclasses are discussed in the following two sections.

B.1.2.25 AddLocalVariable

What The `AddLocalVariable` class represents the addition of a local variable to the list of temporary variables defined in an existing behavioral entity.

How & Where To apply an `AddLocalVariable` the following action needs to be performed: *Add local variable L to behavioral entity B*.

When

- A `Class` with `name` C defines a `BehaviouralEntity` with `name` B
- `BehaviouralEntity` B does not define a `LocalVariable` with `name` L

B.1.2.26 RemoveLocalVariable

What The `RemoveLocalVariable` class represents the removal of a local variable from the list of temporary variables defined in an existing behavioral entity.

How & Where To apply a `RemoveLocalVariable` the following action needs to be performed: *Remove local variable L from behavioral entity B*.

When

- A `Class` with `name` C defines a `BehaviouralEntity` with `name` B
- `BehaviouralEntity` B defines a `LocalVariable` with `name` L

B.1.3 AssociationChange

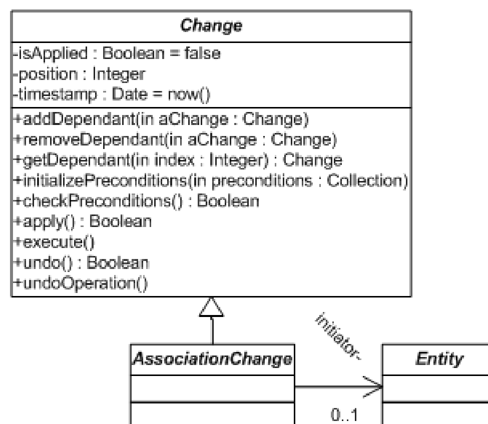

Figure B.13: Design - AssociationChange

What The `AssociationChange` class represents a change with respect to associations between two entities (e.g. inheritance definition) and is shown in figure B.13. As the figure shows, it inherits from `Change` and maintains a reference to the entity setting up the association (`initiator` relationship).

B.1.3.1 InheritanceDefinitionChange

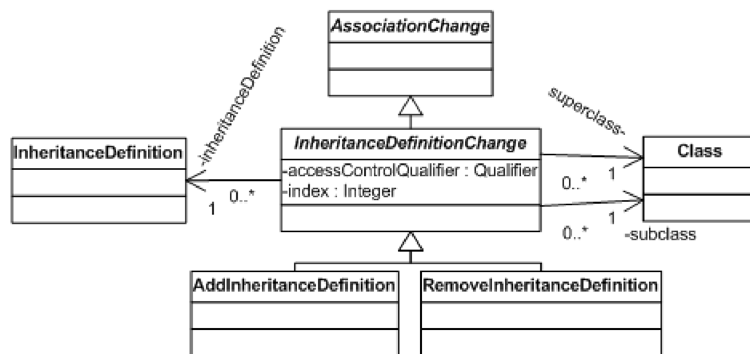

Figure B.14: Design - InheritanceDefinitionChange

What Figure B.14 shows the `InheritanceDefinitionChange` class and its subclasses. The figure also shows that `InheritanceDefinitionChange` inherits from `AssociationChange`. The `InheritanceDefinitionChange` class expresses the addition or removal of an inheritance association between two classes. One class then plays the role of the `superclass`, the other plays the role of the `subclass`. Furthermore the `InheritanceDefinitionChange` class owns following attributes and relations:

- `accessControlQualifier`: determines how subclasses access their superclasses.
- `index`: to support multiple inheritance, subclasses maintain lists with their superclasses. The `index` attribute of `InheritanceDefinitionChange` refers to the position of the superclass in such a list.
- `inheritanceDefinition`: a reference to the added or removed inheritance definition.

The inherited `initiator` association does not apply to inheritance definitions. The two subclasses of `InheritanceDefinitionChange` are explored in the following two sections.

B.1.3.2 AddInheritanceDefinition

What The `AddInheritanceDefinition` class represents the addition of an inheritance definition between two classes. The added inheritance definition is maintained by the inherited `inheritanceDefinition` association.

How & Where To apply an `AddInheritanceDefinition` the following action needs to be performed: *Add an inheritance definition between subclass and superclass*.

When

- `subclass` exists
- `superclass` exists

B.1.3.3 RemoveInheritanceDefinition

What The `RemoveInheritanceDefinition` class represents the removal of an inheritance definition between two classes. The removed inheritance definition is maintained by the inherited `inheritanceDefinition` association.

How & Where To apply a `RemoveInheritanceDefinition` the following action needs to be performed: *Remove the inheritance definition between subclass and superclass*.

When

- `subclass` exists
- `superclass` exists
- `subclass` inherits from `superclass`

B.1.3.4 InvocationChange

What The `InvocationChange` class (see figure B.15) expresses the addition or the removal of an invocation to/from a behavioral entity. As shown in figure B.15, the `InvocationChange` class inherits from the `AssociationChange`

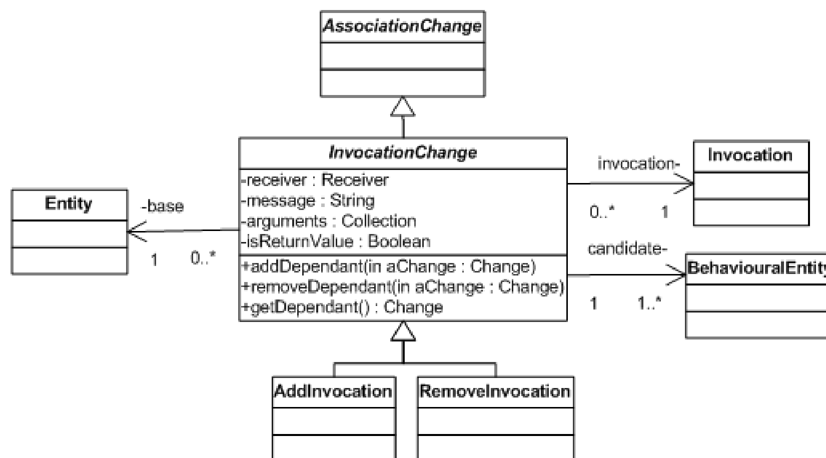

Figure B.15: Design - InvocationChange

class. An `InvocationChange` maintains a reference to the added or removed `Invocation` via the `invocation` relationship. Furthermore it owns following attributes and relations:

- `base`: the entity that defines the invoked behavioral entity.
- `receiver`: the `Entity`, `Argument` or `Invocation` on which the specified behavioral entity is called on.
- `message`: a `String` expressing the invoked message.
- `arguments`: a collection of passed entities.
- `isReturnValue`: a `Boolean` indicating if the invocation is the return statement of the behavioral entity where it is specified.
- `candidate`: refers to a behavioral entity possibly invoked. All candidates have the same signature.

The inherited `initiator` association refers to the behavioral entity that specifies the concerned invocation. An `InvocationChange` can contain dependant changes revealing more information about the invocation. These dependants are managed by the inherited `addDependant`, `removeDependant` and `getDependant` methods and provide detailed change information about the receiver and arguments of the invocation. The subclasses of `InvocationChange` are discussed in the following two sections.

When

- `initiator` is not empty
- `base` is not empty
- `receiver` is not empty
- `message` is not empty
- `isReturnValue` is not empty
- There is at least one `candidate`

B.1.3.5 AddInvocation

What The `AddInvocation` class expresses the addition of an invocation to a behavioral entity. The added invocation is maintained by the inherited `invocation` relationship of `InvocationChange`.

How & Where To apply an `AddInvocation`, the invocation can be reconstructed by sequentially combining its dependants and the following action is taken: *Add invocation I to behavioral entity B*.

When

- A `BehaviouralEntity` with `name` B exists
- A `Class` `base` defines a `BehaviouralEntity` with `name` `message`

B.1.3.6 RemoveInvocation

What The `RemoveInvocation` class expresses the removal of an invocation from a behavioral entity. The removed invocation is maintained by the inherited `invocation` association of `InvocationChange`.

How & Where To apply a `RemoveInvocation`, the invocation can be reconstructed by sequentially combining its dependants and the following action is taken: *Remove invocation I from behavioral entity B*.

When

- A `BehaviouralEntity` with `name` B exists
- A `Class` `base` defines a `BehaviouralEntity` M with `name` `message`
- `BehaviouralEntity` M is invoked by `initiator` B on `receiver`.

B.1.3.7 AccessChange

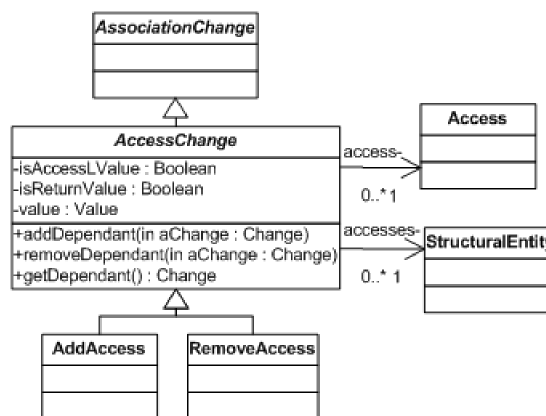

Figure B.16: Design - AccessChange

What The `AccessChange` class is depicted in figure B.16 and expresses the addition or removal of an access to a behavioral entity. As shown in figure B.16, `AccessChange` inherits from `AssociationChange`. An `AccessChange` maintains a reference to the added or removed `Access` by the `access` association. Furthermore it owns following attributes and relations:

- `isAccessLValue`: a `Boolean` indicating whether the concerned access corresponds to a getter action (which returns the value of a certain structural entity) or a setter action (which assigns a value to a certain structural entity). A `true` value indicates that it is a setter.
- `value`: in case of a setting action, this attribute maintains the `Entity`, `Argument` or `Association` assigned to the structural entity of this access.
- `isReturnValue`: a `Boolean` indicating if the access is the return value of the behavioral entity where it is specified.

- `accesses`: refers to the accessed structural entity.

The inherited `initiator` association refers to the behavioral entity where the concerned access is specified. The assignment side of a setting access statement can be complex and dissected in smaller changes which are managed by the `dependant` methods provided by the `Change` class. As figure B.16 shows, `AccessChange` acts as a superclass for the `AddAccessChange` and `RemoveAccessChange` classes which do not maintain any attributes or associations.

When

- `initiator` is not empty
- `accesses` is not empty
- `isAccessLValue` is not empty
- `isReturnValue` is not empty
- if `isAccessLValue` is true then `isReturnValue` is false
- if `isAccessLValue` is true then `value` is not empty

B.1.3.8 AddAccess

What The `AddAccess` class expresses the addition of an access statement to a behavioral entity. The added access is maintained by the inherited `access` relationship of `AccessChange`.

How & Where To apply an `AddAccess`, the access statement can be reconstructed by sequentially combining its dependants. To add an access, the following action is taken: *Add access A to behavioral entity B*.

When

- A `BehaviouralEntity` `initiator` exists
- A `StructuralEntity` `accesses` exists

B.1.3.9 RemoveAccess

What The `RemoveAccess` class expresses the removal of an access statement from a behavioral entity. The removed access is maintained by the inherited `access` relationship of `AccessChange`.

How & Where To apply a `RemoveAccess`, the access statement can be reconstructed by sequentially combining its dependants and the following action is taken: *Remove access A from behavioral entity B.*

When

- A `BehaviouralEntity initiator` exists
- A `StructuralEntity accesses` exists
- `accesses` is accessed in `initiator`

B.1.4 ArgumentChange

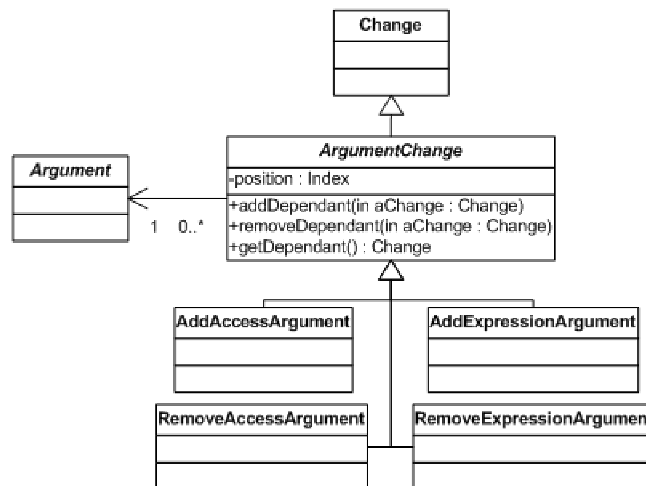

Figure B.17: Design - ArgumentChange

What Figure B.17 depicts the `ArgumentChange` class which represents the addition or removal of an argument in a method or function call. An argument can either be an access to some kind of variable (`AccessArgumentChange`)

or a complex expression (ExpressionArgumentChange). The index attribute represents the position of the argument in the complete argument list of the called BehaviouralEntity. The ArgumentChange class contains dependant changes expressing the argument information. As seen in figure B.17, the ArgumentChange class has four subclasses:

- AddAccessArgument: expresses the addition of an access as argument (e.g. access to a local variable).
- RemoveAccessArgument: is used to represent the removal of an access as argument (e.g. access to a local variable).
- AddExpressionArgument: represents the addition a complex expression as argument (e.g. an invocation).
- RemoveExpressionArgument: is used to express the removal of a complex expression as argument (e.g. an invocation).

B.2 Extensions

This section describes two extensions to the change hierarchy established in the previous section. The first one focusses on changes to the dynamic state of a running software system (e.g. living instances of a class) while the last extension aims at Smalltalks language features (Smalltalk single inheritance vs FAMIX multiple inheritance).

B.2.1 Extension for dynamic state

B.2.1.1 InstanceChange

What Figure B.18 depicts the abstract InstanceChange class which represents the addition or removal of a class instance. As shown in figure B.18, InstanceChange inherits from the Change class. The InstanceChange class maintains a reference to the concerned instance via the instance relationship. Furthermore, an InstanceChange stores of which class the expressed instance is instantiated. The following two sections respectively discuss the AddInstance and RemoveInstance subclasses.

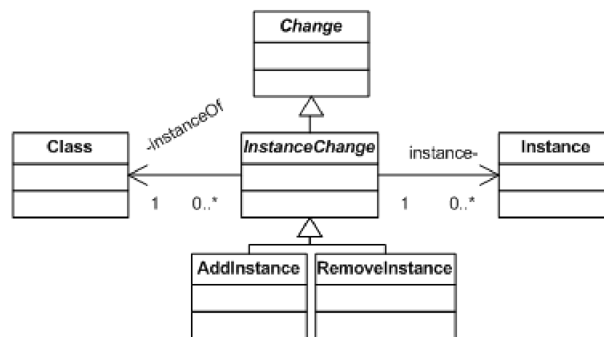

Figure B.18: Design - InstanceChange

B.2.1.2 AddInstance

What The `AddInstance` class expresses the addition of a class instance. The added instance is referred to via the inherited `instance` relationship of `InstanceChange`.

How & Where The application of an `AddInstance` requires the following action: *Add instance I of class C*.

When

- A `Class` C exists
- `Class` C is not an abstract class

B.2.1.3 RemoveInstance

What The `RemoveInstance` class expresses the deletion of a class instance. The deleted instance is referred to via the inherited `instance` relationship of `InstanceChange`.

How & Where The application of a `RemoveInstance` requires the following action: *Remove instance I of class C*.

When

- A `Class` C exists

- `Class` C is not an abstract class
- `Instance` I exists.

B.2.1.4 AttributeValueChange

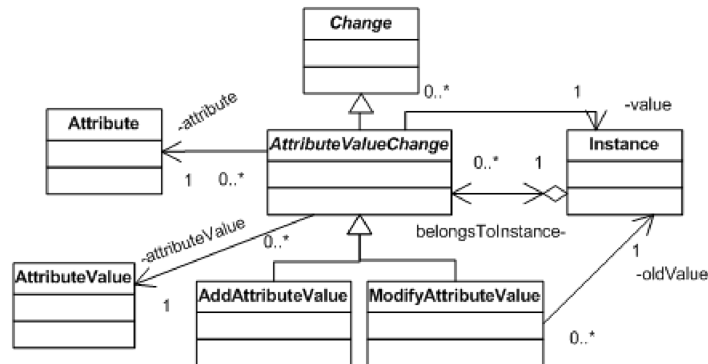

Figure B.19: Design - AttributeValueChange

What Figure B.19 shows the `AttributeValueChange` class which expresses the addition or modification of an attribute value. As shown in figure B.19, `AttributeValueChange` inherits from the `Change` class.
An `AttributeValueChange` maintains a reference to the `Attribute` of which the value has changed (`attribute` association). Furthermore it maintains following associations:

- `attributeValue`: the added or modified `AttributeValue`
- `value`: the `Instance` held as a value for a certain `Attribute`
- `belongsToInstance`: the attribute value belongs to to a certain `Instance`.

B.2.1.5 AddAttributeValue

What The `AddAttributeValue` class expresses the addition of an attribute value to an instance (`belongsToInstance`) for a certain attribute (`attribute`).

How & Where To apply an `AddAttributeValue` the following action needs to be executed: *Add a value V for an attribute A to an instance I.* This results in the creation of a new `AttributeValue` AV.

When

- AttributeValue AV does not exist
- A Class C defines an Attribute with name A
- Instance I exists
- V is an Instance

B.2.1.6 ModifyAttributeValue

What Since attributes always have a value (e.g. nil), values can not just be removed. Therefore the ModifyAttributeValue class exists: it represents the modification of an attribute value. The ModifyAttributeValue class refers to the value the attribute had before modification (oldValue association).

How & Where To apply a ModifyAttributeValue the following action needs to be executed: *Modify a value V to W for an attribute A of an instance I.* This results in the modification of an existing AttributeValue AV.

When

- AttributeValue AV exists
- A Class C defines an Attribute with name A
- Instance I exists
- V is an Instance
- W is an Instance

B.2.1.7 RemoveClass

Taking into account dynamic state raises another precondition for the RemoveClass class: *No instances of class C exist.*

B.2.1.8 GlobalVariableValueChange

What Figure B.20 shows the abstract GlobalVariableValueChange class which expresses the addition or modification of a value for a global variable.

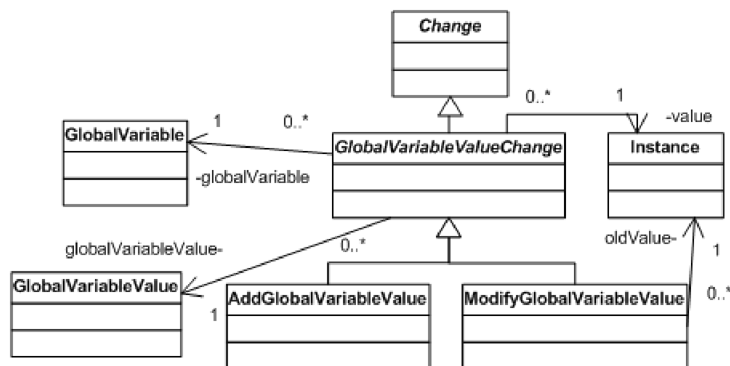

Figure B.20: Design - GlobalVariableValueChange

As shown in figure B.20, `GlobalVariableValueChange` inherits from the `Change` class. A `GlobalVariableValueChange` maintains a reference to the `GlobalVariable` of which the value has changed (`globalVariable` association). Furthermore it maintains following associations:

- `globalVariableValue`: the added or modified value for the concerned global variable
- `value`: the `Instance` held as a value for a certain `GlobalVariable`

B.2.1.9 AddGlobalVariableValue

What The `AddGlobalVariableValue` class expresses the addition of a value for a certain global variable denoted by the `globalVariable` association.

How & Where To apply an `AddGlobalVariableValue` the following action needs to be executed: *Add a value V for a global variable G*. This results in the creation of a new `GlobalVariableValue` GV.

When

- `GlobalVariableValue` GV does not exist
- A `GlobalVariableValue` with `name` G exists
- V is an `Instance`

B.2.1.10 ModifyAttributeValue

What The `ModifyAttributeValue` class expresses the modification of a value for a global variable expressed by the `globalVariable` association.

How & Where To apply a `ModifyAttributeValue` the following action needs to be executed: *Modify a value V to W for a global variable G*. This results in the modification of an existing `GlobalVariableValue` GV.

When

- `GlobalVariableValue` GV exists
- A `GlobalVariableValue` with `name` G exists
- V is an `Instance`
- W is an `Instance`

B.2.2 Extension for Smalltalk

B.2.2.1 AddClass

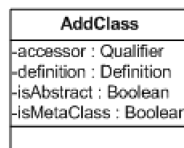

Figure B.21: Design - AddClass

What Figure B.21 shows that the `isMetaClass` attribute has been added to the `AddClass` class. This is a `Boolean` indicating whether or not the added class represents a Smalltalk metaclass..

When

- `isMetaClass` is either `true` or `false`

B.2.2.2 BehaviouralEntityChange

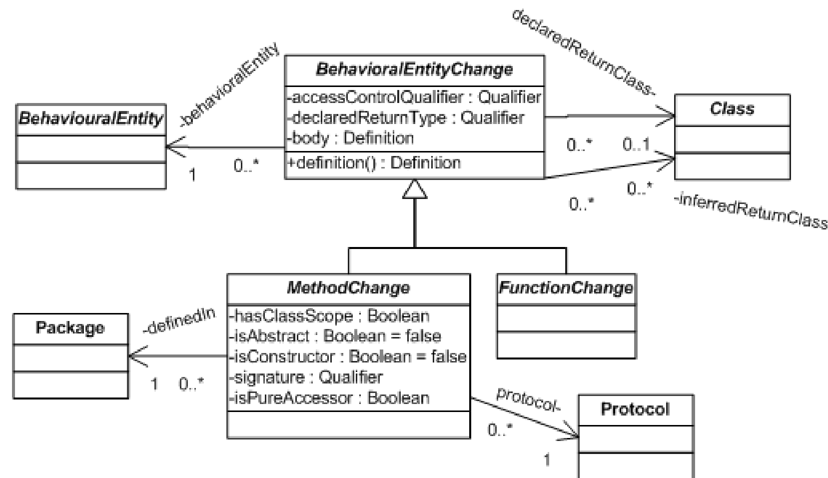

Figure B.22: Design - BehaviouralEntityChange

What Figure B.22 shows that the `inferredReturnClass` association has been added to the `BehavioralEntityChange` class which refers to all possible candidates for the return type of the concerned behavioral entity. Following attributes have been pushed down to the `MethodChange` entity:

- `signature`: each method has a unique signature. Obviously anonymous functions do not have any signature.
- `isPureAccessor`: a `Boolean` indicating whether or not the added or removed method is a pure getter or setter.

Figure B.22 also shows that the `protocol` association has been added between the `MethodChange` and `Protocol` classes. This relationship refers to the protocol according to which protocol the added or removed method is organized. The `definedIn` association between the `MethodChange` and `Package` classes denotes the fact that a method belongs to exactly one package in Smalltalk.

B.2.2.3 StructuralEntityChange

What Figure B.23 reveals that the `inferredClass` association has been added to the `StructuralEntityChange` class. This association refers to all possible candidates for the declared type.

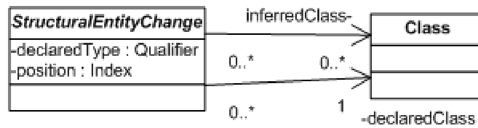

Figure B.23: Design - StructuralEntityChange

B.2.2.4 AddAttribute

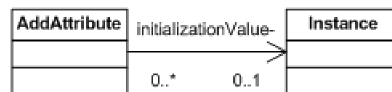

Figure B.24: Design - AddAttribute

What The `initializationValue` association has been added to the `AddAttribute` and `GlobalVariable` classes. Smalltalk allows initialization of attributes and global variables.

B.2.2.5 InheritanceDefinition

The FAMIX model allows multiple inheritance while Smalltalk does not. The following precondition has been added to the `AddInheritanceDefinition` class in order to impose single inheritance: *`subclass` does not inherit from any superclass*. Note however that the added precondition is in conflict with the following invariant stated in appendix A: *Each `subclass` inherits from exactly one `superclass`*. Thus `AddInheritanceDefinition` will only be used when creating a new subclass.

The application of `RemoveInheritanceDefinition` in Smalltalk results in a class not inheriting from any superclass. Note that this is also in conflict with the invariant *Each `subclass` inherits from exactly one `superclass`*. Thus `RemoveInheritanceDefinition` will only be used when removing a subclass.

B.2.2.6 ModifyInheritanceDefinition

What The `ModifyInheritanceDefinition` class has been introduced to change the inheritance between two classes. It maintains two attributes and two associations:

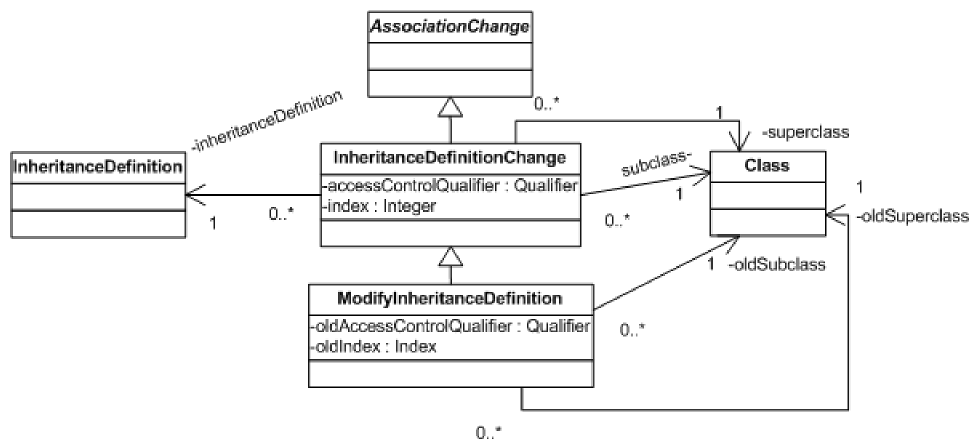

Figure B.25: Design - ModifyInheritanceDefinition

- `oldAccessControlQualifier`: determines how subclasses accessed their superclasses before the change was applied.
- `index`: to support multiple inheritance, subclasses maintain lists with their superclasses. The index attribute of `InheritanceDefinitionChange` refers to the old position of the superclass in such a list.
- `oldSubclass`: the class playing the role of `subclass` before the change was applied
- `oldSuperclass`: the class playing the role of `superclass` before the change was applied

When

- `subclass` exists
- `oldSubclass` exists
- `superclass` exists
- `oldSuperclass` exists
- `oldSubclass` inherits from `oldSuperclass`