Composite Domain-Specific Language Design and Development using Aspect-Oriented Weaving

Master thesis
60 credits

Henning Berg
[hennb@ifi.uio.no]

1st February 2010
Preface

This is a master thesis submitted to the Department of Informatics, Faculty of Mathematics and Natural Sciences at the University of Oslo as part of the requirements for Master of Science (MSc Informatics). The work has been carried out in the research group for Object orientation, Modelling and Language (OMS) under the supervision of Dr Birger Møller-Pedersen (professor) during the time period from January 2008 to January 2010.

The context of this thesis has been design and development of composite Domain-Specific Languages using aspect-oriented weaving and model transformations. The results of the work include elaboration on concepts of language design as verified by a fully-working language design and development platform.

I truly believe that some of my best moments in life have occurred working with this thesis and I am grateful for all experiences.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor Dr Birger Møller-Pedersen for his insight and the many interesting discussions during the course of this project. His knowledge and focus on correctness have undoubtedly been helpful in reaching my goals. Other people deserving recognition are

Mickael Clavreul (INRIA / IRISA, France) for his support and in-depth information on Kompose.

Mansur Ali Abbasi (Schlumberger WesternGeco, Norway) for sharing his knowledge on meta modelling and Kermeta. I will also use the opportunity to thank Mr Abbasi for inspiring me to seek a high degree of professionalism.

A thought also goes to Jon Oldevik (SINTEF, Norway) and Rune Flobakk (BEKK, Norway) for their inputs regarding EMF and Java, respectively. Finally, I thank my family and friends for their encouragement and patience during my studies. Especially, I thank my father for outstanding support throughout the years.
# Table of contents

1 Introduction .......................................................................................................................... 1  
  1.1 CONTEXT OF WORK ..................................................................................................... 1  
  1.2 MOTIVATION ............................................................................................................... 2  
  1.3 RESEARCH GOALS, SCOPE AND METHODS.................................................................... 3  
    1.3.1 GOALS ....................................................................................................................... 3  
    1.3.2 SCOPE ..................................................................................................................... 3  
    1.3.3 METHOD .................................................................................................................. 4  
  1.4 NOTES FOR THE READER ............................................................................................ 5  
    1.4.1 ACCOMMODATED READING PATTERNS ............................................................... 5  
    1.4.2 NOTATIONS USED .................................................................................................... 5  
    1.4.3 INTERNAL REFERENCES ......................................................................................... 5  

PART I  BACKGROUND AND CONTEXT .............................................................................. 6  

2 Methodologies and approaches ............................................................................................ 7  
  2.1 SEPARATION OF CONCERNS ..................................................................................... 7  
  2.2 ASPECT ORIENTATION ............................................................................................... 8  
  2.3 MODEL-DRIVEN ENGINEERING .................................................................................. 8  
  2.4 LANGUAGE DRIVEN DEVELOPMENT ......................................................................... 9  

3 Traditional software development ....................................................................................... 10  
  3.1 PHASES OF SOFTWARE DEVELOPMENT .................................................................. 10  
  3.2 ANALYSIS .................................................................................................................. 10  
  3.3 APPLICATION MODELLING ......................................................................................... 11  
  3.4 IMPLEMENTATION ....................................................................................................... 12  

4 Language design .................................................................................................................... 13  
  4.1 META MODELLING ........................................................................................................ 13  
    4.1.1 ABSTRACT SYNTAX AND WELL-FORMEDNESS RULES ...................................... 14  
    4.1.2 CONCRETE SYNTAX ............................................................................................... 17  
    4.1.3 SEMANTICS .............................................................................................................. 17  
    4.1.4 MAPPINGS .............................................................................................................. 17  

5 Domain-Specific Languages .................................................................................................. 19  
  5.1 INCEPTION AND HISTORICAL CONTEXT ................................................................ 19  
  5.2 CHARACTERISTICS OF DOMAIN-SPECIFIC LANGUAGES ....................................... 20  
  5.3 ADOPTION OF DOMAIN-SPECIFIC LANGUAGES ...................................................... 20  
  5.4 WHEN TO DEVELOP A DSL ......................................................................................... 21  

6 Software evolution .................................................................................................................. 23  
  6.1 THE NATURE OF SOFTWARE EVOLUTION ................................................................ 23  
  6.2 IMPACTS AND IMPACT ANALYSIS ........................................................................... 24  
    6.2.1 HORIZONTAL IMPACTS ........................................................................................... 25  
    6.2.2 VERTICAL IMPACTS ............................................................................................... 25  
  6.3 CO-EVOLUTION ............................................................................................................. 25  

PART II  TECHNOLOGY FOUNDATION .................................................................................. 26  

7 Technology foundation .......................................................................................................... 27
11 The concept of composite DSLs .................................................................................. 48
  11.1 MOTIVATION ........................................................................................................ 48
  11.2 COMPOSITE DOMAIN-SPECIFIC LANGUAGES ..................................................... 49
  11.3 COMPONENTS OF A COMPOSITE DSL ................................................................. 51
    11.3.1 LANGUAGE META MODEL ........................................................................... 51
    11.3.2 SEMANTICS MODULE .............................................................................. 51
  11.4 ASPECT-ORIENTED LANGUAGE WEAVING ......................................................... 52
  11.5 COMPOSITE DOMAINS ....................................................................................... 53
  11.6 REFINEMENT OF A COMPOSITE DSL ................................................................. 54
    11.6.1 REFINEMENT OF ABSTRACT SYNTAX ...................................................... 54
    11.6.2 REFINEMENT OF SEMANTICS .................................................................. 55

PART III CONTRIBUTION .......................................................................................... 30

8 Overview of the contribution ..................................................................................... 31
  8.1 THE ECOMMERCE FRAMEWORK ........................................................................ 31
  8.2 AN ATOMIC DOMAIN-SPECIFIC LANGUAGE ......................................................... 31
  8.3 THE LDE PLATFORM .............................................................................................. 32
  8.4 A SET OF COMPOSITE DOMAIN-SPECIFIC LANGUAGES .................................... 32

9 The Ecommerce framework ....................................................................................... 33
  9.1 AN OVERVIEW OF THE FRAMEWORK ................................................................ 33
    9.1.1 CONTROL LOGIC ......................................................................................... 34
    9.1.2 DOMAIN MODEL ......................................................................................... 35
    9.1.3 GUI AND PRESENTATION ............................................................................ 35
    9.1.4 SERVICES AND COMPUTATION MODEL .................................................... 36
    9.1.5 DATA PERSISTENCE .................................................................................... 37
    9.1.6 UTILITIES ..................................................................................................... 38
  9.2 FRAMEWORK INSTANTIATION ............................................................................. 38
    9.2.1 COMBINING PRE-DEFINED ENTITIES ......................................................... 39
    9.2.2 REDEFINING FRAMEWORK CONCEPTS ..................................................... 41
    9.2.3 CONFIGURATION OF DATA SOURCES ....................................................... 41
  9.3 THE PURPOSE OF THE FRAMEWORK ................................................................. 42

10 Designing a traditional DSL .................................................................................... 43
  10.1 A DSL FOR MODELLING OF CARS .................................................................. 43
  10.2 CREATING THE LANGUAGE META MODEL ..................................................... 43
    10.2.1 THE GENERATED META MODEL CODE ................................................... 44
    10.2.2 THE CAR META MODEL EDITOR .............................................................. 44
  10.3 READING A MODEL AT RUNTIME ..................................................................... 45
  10.4 EXTENDING THE META MODEL ..................................................................... 46
  10.5 CHANGING THE SEMANTICS .......................................................................... 46
  10.6 POSSIBILITIES AND LIMITATIONS USING EMF ............................................. 47

11 The concept of composite DSLs .............................................................................. 48
  11.1 MOTIVATION ........................................................................................................ 48
  11.2 COMPOSITE DOMAIN-SPECIFIC LANGUAGES ..................................................... 49
  11.3 COMPONENTS OF A COMPOSITE DSL ................................................................. 51
    11.3.1 LANGUAGE META MODEL ........................................................................... 51
    11.3.2 SEMANTICS MODULE .............................................................................. 51
  11.4 ASPECT-ORIENTED LANGUAGE WEAVING ......................................................... 52
  11.5 COMPOSITE DOMAINS ....................................................................................... 53
  11.6 REFINEMENT OF A COMPOSITE DSL ................................................................. 54
    11.6.1 REFINEMENT OF ABSTRACT SYNTAX ...................................................... 54
    11.6.2 REFINEMENT OF SEMANTICS .................................................................. 55
## 16.1 The Importance of Decentralised Development .............................................. 142
## 16.2 A Decentralised Language Development Process .......................................... 143
## 16.3 Arbitrary Language Integration ........................................................................ 143

### 17 Frameworks versus DSLs .................................................................................... 148

#### 17.1 Frameworks versus DSLs .............................................................................. 148
#### 17.2 Traditional DSLs versus Composite DSLs ....................................................... 150
#### 17.3 Differences in Development Methodologies ..................................................... 151
#### 17.4 Core Properties .............................................................................................. 151

### 18 Meta components .............................................................................................. 155

#### 18.1 Traditional Component Mechanisms ............................................................. 155
#### 18.2 Meta Model Nesting ....................................................................................... 156
#### 18.3 Meta Component Definition .......................................................................... 158
#### 18.4 Designing a Meta Component ........................................................................ 160
#### 18.5 Meta Modelling using Components ............................................................... 165
#### 18.6 Practical Meta Modelling using Components ................................................ 168
#### 18.7 Component Hierarchy .................................................................................... 170
#### 18.8 Weaving of Components ................................................................................ 170

### 19 Related work ....................................................................................................... 171

#### 19.1 Case 1: MOF 2.0 Meta Model Components .................................................. 171
#### 19.2 Case 2: Techniques for Meta Model Composition ........................................ 171
#### 19.3 Case 3: When and How to Develop DSLs ....................................................... 172
#### 19.4 Case 4: Modelling with Multiple DSLs ......................................................... 172
#### 19.5 Case 5: Polymorphic Embedding of DSLs ..................................................... 172

### 20 Conclusions ....................................................................................................... 174

#### 20.1 Summary ....................................................................................................... 174
    - 20.1.1 Composite Language Design and Development .................................... 174
    - 20.1.2 Frameworks and DSLs ............................................................................. 175
    - 20.1.3 Meta Modelling using Components .................................................... 176
#### 20.2 Weaknesses .................................................................................................... 176
#### 20.3 Contributions ............................................................................................... 176
#### 20.4 Critical Assessment ...................................................................................... 177

### 21 Future work ....................................................................................................... 178

#### 21.1 Solidification of Work .................................................................................... 178
#### 21.2 LDE Platform 2.0 ....................................................................................... 178
#### 21.3 Increasing Integrability .................................................................................. 178

---

**Bibliography** ............................................................................................................ 179

- Books ....................................................................................................................... 179
- Scientific articles, specifications and manuals ....................................................... 179
- Web resources .......................................................................................................... 181

**Appendices** .......................................................................................................... 182

- Delivered material / CD contents .......................................................................... 183
    - A.1 Delivered Material ....................................................................................... 183
    - A.2 Contents on accompanying CD ................................................................. 183
1 Introduction

This chapter gives an introduction to the work of this thesis. It is divided in six sections comprising information on context of work, motivation and research goals. This includes definition of scope and an overview of methods used to pursue the goals.

1.1 CONTEXT OF WORK

In the last 50 years, development of general languages has been the major trend in language design. This has resulted in the creation of a large number of General Purpose Languages (GPLs) whose common goal is to be as general as possible to support modelling and computing within arbitrary problem domains. This is achieved by providing a set of general first-class language constructs. The conception among developers has been that one language can solve all problems. In recent years, another type of language, known as Domain-Specific Languages (DSLs), has gained popularity. Contrary to general languages, these languages are custom-made to support modelling and computing related to a specific, confined problem domain. Consequently, they are designed to solve a domain-specific kind of problems.

Domain-Specific Languages are not a new invention. In fact, these languages have been used in parallel with General Purpose Languages for half a century. The recent popularity is the result of a new approach in software development that focuses on language design, as opposed to using a general language, in order to solve a given problem. This approach, known as Language Driven Development (LDD) [1], is directly related to the Model-Driven Engineering (MDE) [2] methodology and Aspect Orientation (AO) [3]. The main driving force behind this new approach is the ability to (rapidly) tailor a language for a special type of problems that, as a consequence, can be solved in an efficient and pragmatic manner.

Another popular approach in software development is the use of domain-specific frameworks. Frameworks provide pre-defined concepts and components that can be combined. Currently there are numerous frameworks that have been developed over several years. Framework usability and instantiation, however, have been the focus of discussion. Using frameworks, also known as framework instantiation, has in many situations proved to be challenging. Furthermore, it may require in-depth knowledge of the framework’s architecture and internal class hierarchy. Frameworks can also have complex interrelationships between concepts and components that must be maintained. Consequently, it can be difficult to write proper instantiation code. All these issues impose the need of detailed, up-to-date documentation. It may be argued that frameworks represent the counterpart to Domain-Specific Languages. Diverse research has been performed to investigate how Domain-Specific Languages can be integrated with frameworks. One result of this work is a description of how Domain-Specific Languages can be used to instantiate frameworks [4].

Software evolves continuously. This is a consequence of new requirements and ultimately a changing world. Clearly, Software Evolution [5] results in issues that have to be explicitly addressed. Such issues include co-evolution phenomena between software entities at different abstraction levels. For instance, languages and their models are likely to evolve at different rates. A consequence of this is issues related to forwards and
backwards compatibility.

1.2 Motivation
There are several ways of defining a Domain-Specific Language. This includes development of a complete language from scratch, utilisation of one or more frameworks as semantics foundation and creation of an embedded language, known as Embedded Domain Specific Language (EDSL) [6]. Obviously, there are no unifying factors between arbitrary DSLs since there are numerous ways to design and implement these. As previous described, a new popular trend has emerged where language design is in focus. This approach opens up for many interesting possibilities. However, one particular aspect of this new approach is questionable. Clearly, as more and more developers follow this approach a new breed of languages are created. In the long run this implies potentially a huge number of languages. These languages are proprietary and expressive only in a narrow field. Moreover, it is also likely that many languages will support overlapping problem domains. In fact, there may be languages that are quite similar, but have a slightly different structure. Keeping an overview of all these languages may prove to be very challenging. It can also be difficult to find an existing language that suits a specific problem, because of the variations. To date, this issue has not been encountered in a large scale within language development. It can, however, be related to the principles of Information Overload. Language development should follow a certain set of methodologies to ensure that languages are designed according to the best practices and design patterns available. Otherwise, the consequence can be reinvention of already well-proved concepts and mechanisms. Furthermore, methodologies are required to ensure that language duplicates are avoided.

As pointed out; software evolves. This is primarily caused by new requirements and changes in problem domains. As might be expected, a DSL is vulnerable to changes in the problem domain, because such changes have to be reflected directly in the language’s abstract syntax and semantics. This is not the case with GPLs which, from their inception, provide a set of generic constructs that may be used to reflect arbitrary problem domain concepts. As can be seen, DSLs are subject to software evolution at a higher rate than what is the case with GPLs. Hence a central issue in this regard is how to support new requirements and changing problem domains in a straightforward, intuitive and efficient manner.

Some work has been performed on meta model composition and weaving [7][8]. This work has focused on generic meta model merging techniques, principles and theory. Moreover, a proposed component-oriented version of the Meta Object Facility (MOF) is discussed in [9]. However, it is of interest to investigate a more pragmatic approach to meta model and language weaving where focus is on language tailoring. That is, investigation of mechanisms that support refinement of a language to suit a specific problem domain. Clearly, this has to be considered in the light of software evolution. Please refer the chapter Related work for more information on existing research.
1.3 RESEARCH GOALS, SCOPE AND METHODS

1.3.1 GOALS
As motivated in the previous section, the main goal of this thesis is to address aspects of Domain-Specific Language design with the intention of creating a platform for language development and execution of corresponding models. This platform should provide mechanisms for customisation and extension of DSLs to meet the challenges of evolving requirements and changing problem domains (tools, editors and environments). The platform should provide a fully-working runtime environment for execution of models / programs in DSLs (both atomic and composite DSLs). The purpose of the platform can be divided in two parts. Firstly, it should constitute a proof-of-concept / prototype application(s) that exemplifies a complete solution for composite DSL development and model execution. Secondly, it should serve as basis for reasoning about concepts and principles regarding language design. Important principles and points in this regard are evolving languages, standardisation of language components, decentralised language development and reuse of DSLs.

Sub goals include development of a complete framework (domain-specific abstractions and structures) that can be used to illustrate aspects of software development using frameworks and how DSLs can be integrated with third-party software. This includes investigation of similarities and difficulties between frameworks and DSLs.

Traditional software modelling technology comprises components, templates, state machines, use cases and more. On the contrary, meta modelling mostly deals with flat class diagrams and associated semantics. In the light of this thesis it is interesting to investigate the possibilities of component-based meta modelling. Such a component mechanism would provide means of grouping domain-specific meta model concepts closely together. This would potentially provide a new dimension to the art of language design and development. Some work on this has been carried out in [9] where components with a traditional flat structure are promoted. On the contrary, we will focus on meta model nesting.

1.3.2 SCOPE
In this thesis, language design is in focus. Language design is a rich art and comprehensive discipline. Naturally, discussing all aspects of language design is out of scope of this thesis. Consequently, focus has been put on certain points and objectives. These relate to customisation, extension and reuse of languages with support of decentralised language development. To a large extent, these topics have been selected based on potential scientific value. There are, however, some closely related spin-off topics of the initial objectives that are discussed briefly. These are included for the sake of completeness.

Some parts of the technical solution deals with dynamic entities that may have several states and configurations. Supporting every possible configuration and variation are in some cases redundant with regard to the goals and objectives of this thesis. Such cases are explicitly stated.

Investigation of additional mechanisms and principles of meta modelling is of current interest, especially composition and the use of templates. However, due to time
constraints, focus has been put on one modelling principle: components and composition. This will be investigated and discussed briefly.

1.3.3 METHOD

The method and approach used to realise the work of this thesis comply with the genres of Software Engineering, Language Driven Development (LDD), Model-Driven Engineering (MDE) and Aspect Orientation (AO). Specifically, the applied research strategy is based on development of artefacts that are used to illustrate and prove aspects of software and language design. This strategy can be described in the following steps:

**Step 1 – Acquiring knowledge**

An important initial step is to acquire a fundamental understanding and knowledge of concepts, principles and existing technology related to the goals and objectives of the work to be carried out. Clearly, it is essential to gain an overview of related work to be able to put things in perspective and build further on proved results.

**Step 2 – Creating a generic framework**

Creating the framework prior to the language development platform is a natural choice since frameworks play an essential role in modern software development. Acquiring in-depth knowledge of frameworks and framework instantiation early on is therefore preferred. There are has been a lot of work performed with respect to frameworks. Some essential principles and approaches are discussed in [10]. Key points in the design process of the framework are focus on flexibility and usability, and providing a sound internal architecture.

**Step 3 – Developing the language runtime environment**

One of the most challenging tasks is to create a language runtime environment that supports execution of models in an arbitrary DSL. Furthermore, the runtime environment should support composite DSLs. That is, a language comprising several DSLs working together as a logical whole. It is advantageous to create the runtime environment prior to the development tools in order to start testing early.

**Step 4 – Creation of language development tools**

Several development tools have been identified as critical for the achievement of the identified goals and objectives. This includes tools for meta model weaving, code generation and model transformations. Model weaving is a popular research area. A generic approach for automatic model composition is identified in [11]. The weaver of the platform will be based on this work. Techniques for meta model composition is proposed in [7].

**Step 5 – Making a set of disjunctive examples**

An important step of any work is verification and validation of its correctness. In this case, this can be achieved by creating a set of DSLs of different characters that test the functions of the runtime environment and development tools. These DSLs have two purposes; to test the proof-of-concept applications and to constitute examples that illustrate the mechanisms of the proposed language design approach. The framework logic can be verified in a similar manner.
1.4 NOTES FOR THE READER

1.4.1 ACCOMMODATED READING PATTERNS
This document has been designed to accommodate two manners of reading. Specifically, the concepts and ideas introduced in code snippets are not used in the main text. Consequently, it is possible to omit delving into the details on how mechanisms and implementations are realised. It is, however, suggested to include these details in order to acquire a better understanding of the subjects.

Information Design principles are used to relate chapters of the Discussion part to relevant contribution and current thesis objectives being addressed. This increases readability by pointing to relevant additional reading and puts the discussion in the proper context. Also refer Appendix A.

1.4.2 NOTATIONS USED
Instantiation of classes is a central issue in this thesis. A distinct notation is used occasionally for class instances in order to create more smooth text. As an example, the instances of the fictive classes Function and Attribute may be written ‘function’ and ‘attribute’, respectively. It will be clear from the context if other notations are used.

In the context of meta model architectures, instantiation is by definition the process of creating a realisation (instance) of a class. Additionally, we will sometimes refer to instantiations of language constructs and concepts (abstractions) to increase readability.

1.4.3 INTERNAL REFERENCES
There are three types of internal references: references to chapters, sections and sub sections. References to sub sections are described using the following concise format:

Section ➤ Sub section of chapter Chapter.
PART I
BACKGROUND
AND
CONTEXT
2 Methodologies and approaches

There are several software methodologies and approaches. Here we will briefly describe the approaches that create the foundation for the work presented in this thesis. A section on Separation of Concerns is included due to its importance.

2.1 Separation of Concerns

Separation of Concerns (SoC) [12] is an important design principle. It advocates that a system or model should be separated in distinct parts that correspond to each concern addressed by the system or model. This is imperative in order to comprehend the system and deal with its complexity. Concerns are closely related to the formal requirements of a system.

Unfortunately, Separation of Concerns is not always possible to realise. For instance, in modelling and programming, tangling and scattering are two well-known phenomena [13]. Tangling occurs when an entity of a system, for instance a class in a model, addresses multiple requirements. Scattering, on the other hand, is the result of using multiple entities to represent an individual requirement. Consequences of tangling and scattering are increased complexity, impaired traceability between requirements and implementation, and difficulties in determining how changes to an entity impact other entities and the behaviour of the system. In fact, one small change can have severe unforeseen impacts on other parts and aspects of the system. The ability to analyse and optimise a system in order to improve performance is an important aspect of software development. This can be challenging if the system is not well-structured.

One of the reason tangling and scattering occur is how requirements are implemented using generic representations. For instance, according to the object-orientation paradigm, requirements are realised using classes, interfaces and methods. These units of abstraction, and their composition, do not necessarily provide the appropriate structure to represent requirements properly.

There are also aspects of systems that are difficult to realise using traditional language mechanisms. These kinds of aspects, known as cross-cutting concerns, affect other concerns or parts of a system semantically or performance-wise [13]. Examples of such concerns are security, tracing and logging. These concerns often correspond to non-functional requirements. In many cases cross-cutting code is tangled with basic functionality of a system. This is often the result of lack of coherent (first-class) abstractions for the cross-cutting concern. Moreover, this compromises the reusability of software and makes maintenance difficult.

Different techniques can be used to achieve a higher degree of Separation of Concerns. Some of these are decomposition, encapsulation and focus on modularity. Refer the section Aspect orientation.
2.2 ASPECT ORIENTATION

Aspect Orientation (AO) [3] is a term used to describe systems and artefacts that provide means of separation of concerns. Specifically, aspect orientation addresses cross-cutting concerns by treating aspects as first-class entities. There are several approaches and technologies that support aspect orientation. The most popular approach is Aspect-Oriented Programming (AOP) [3]. AOP is used to encapsulate concerns using suited functionality. Hence, aspects are treated separately from the core business logic of applications. This avoids tangled and scattered code. An example of a common used AOP tool is AspectJ. AspectJ is an extension to Java that provides a distinct aspect construct. This construct is a special class that can reflect a certain concern. Two important mechanisms of an aspect are join points and inter-type declaration. These can be used to weave aspect code with base code and add fields, methods and interfaces to classes. Other research areas are Aspect-Oriented Modelling (AOM) [14], Aspect-Oriented Software Development (AOSD) [15] and aspect weaving strategies [16]. Naturally, some concepts of these technologies and methodologies overlap. Note, however, that some aspects cannot be decomposed due to their cross-cutting nature.

Aspect orientation should also be used in other phases of software development in addition to the implementation phase. For instance, during requirements specification, problem domain analysis and software design / modelling. In fact, each phase can be considered an aspect of the intended software.

2.3 MODEL-DRIVEN ENGINEERING

As the name suggests, Model-Driven Engineering (MDE) [2] is a methodology that is centred on creation and use of models in the software development process. In essence, models represent the information and domain knowledge associated with a project or problem, as acquired from the project’s stakeholders. An important property of models is conformity with a given problem domain. Hence, the models abstract over different implementation platforms which increase focus on problems and ensures compatibility between systems. Architecture design and code generation are two aspects of MDE.

An approach within the MDE methodology is the Object Management Group’s (OMG) Model-Driven Architecture (MDA) [2]. MDA is a strategy and collection of standards for creating highly interoperable systems based on model manipulation and transformations. An important concept in this regard is separation of the specification of a system and the specification of the system on a given platform. Two types of models are identified to achieve this: Platform Independent Models (PIMs) and Platform Specific Models (PSMs), where PSMs can be created from PIMs using model transformations. Separation of specifications ensures that a PSM can be used for several implementation platforms. There are several advantages with this approach including easier model validation, support for system evolution, and increased interoperability by defining systems in platform-independent terms.

Domain-Specific Languages and the Eclipse Modelling Framework (EMF) [17] are examples of other approaches / technologies within the MDE methodology.
2.4 LANGUAGE DRIVEN DEVELOPMENT

Language Driven Development (LDD) [1] is an approach focusing on rapid development of supportive languages and tools in software engineering. The main purpose of LDD is to ensure a highly adaptable software development process where problem solving is not limited to a fixed set of languages. This includes creating semantically rich languages with appropriate abstractions, integration of languages and support for evolving languages. Hence, languages should be designed for a given purpose and problem space. LDD has properties in common with MDE and MDA. While MDE approaches primarily focus on models, LDD evaluates all language artefacts as important in the software development process. It is also argued that MDA is limited in scope of application compared to LDD. Meta modelling is the foundation of LDD. In the chapter Language design, we will go into details regarding meta modelling. Key points using LDD are:

- Rich languages provide abstractions that can deal with complexity
- Diversity can be managed by integrating appropriate languages
- Evolving languages support changes

The work of this thesis is closely related to principles of LDD.
3 Traditional software development

Software development has a long tradition. Numerous techniques and best-practices have emerged during the years. On the contrary, large-scale DSL development has become of special interest lately. Hence, there is still a need for adequate development strategies and methodologies. Nevertheless, the fundamentals in software development are still essential to achieve high quality languages. In this chapter, we will review some of the most relevant aspects of traditional software design and development.

3.1 Phases of software development

Traditional software development comprises several phases. The five major phases are concerned with requirements specification, problem domain analysis, application modelling, implementation, and testing. A simplified model of a much followed software development process is found in Figure 3.1.

Each phase produces a result which is used by the subsequent phase. (The result from the Testing phase can be understood as a complete and stable application.) An important characteristic of the model is the bi-directional relations that exist between the different phases. This implies that changes may propagate in both directions. In addition, it should be possible to trace implemented functionality back to preceding phases. Tracing is also important in quality assurance and process improvement.

Different development methodologies and strategies do exist. Some examples are iterative and incremental development, evolutionary development and Test Driven Development (TDD). In essence, the model abstracts over variations in the development process. In the continuation, focus will be on the Analysis, Application Modelling and Implementation phases.

3.2 Analysis

Analysis, also referred to as problem domain analysis, is one of the most important aspects of software engineering. In essence, problem domain analysis is the process of gaining a thorough understanding of the concepts and structures of a problem domain. Specifically, a problem domain analysis should result in descriptions and characterisations of important aspects and properties that need to be addressed by the intended software. These results are general and independent of technologies. Usually, the acquired information is described in a natural language together with a set of models, diagrams and tables. It is also possible to use formal languages to represent information. Reuse of information is promoted. There are many disciplines associated with problem domain analysis. This includes knowledge acquisition and representation, analysis methodologies, and characterisation of information according to best-practices (information management). In addition, social and cultural differences must be
considered. Many sources of information can be utilised in order to acquire knowledge of a problem domain. Examples of sources are problem-specific documents, organisational diagrams and maps, and empirical studies of behaviour and communication patterns. Clearly, the type of sources depends on the problem domain. Problem domain analysis can be justified with higher software quality and software reuse.

The problem domain analysis process translates to three major tasks:

- Identification of essential domain concepts and their interrelationships
- Capturing of the identified features using a selection of abstractions
- Organisation of the abstractions according to well-proved architectures

**Identification of essential domain concepts and their interrelationships**
The boundaries of a problem domain are often described in the requirements specification. Based on this information, it should be possible to confine relevant objects and relations.

**Capturing of the identified features using a selection of abstractions**
Choosing the appropriate abstractions requires experience. In some occasions it can also be considered an art. Generic abstractions are often preferred. Moreover, only the essential and critical concepts should be reflected by the abstractions.

**Organisation of the abstractions according to well-proved architectures**
There are many architectural models available that describe organisation and structuring of software. It is advantageous to organise the abstractions according to similar architectures. This ensures encapsulation of related concepts and separation of concerns.

Results from the problem domain analysis are used to construct the intentional software. Problem domain analysis is a continuous process where refinement is critical. Software evolves with time. Thus, changes have to be captured using problem domain analysis and eventually be implemented in software. A proper standardisation of analysis results and documentation increases the degree of integration between new observations and existing knowledge. Moreover, choosing the appropriate abstractions and generalisations early ensures that extensions can be implemented more easily. It is out of the scope of this thesis to address detailed problem domain analysis scenarios.

**3.3 APPLICATION MODELLING**
Application modelling is the art and discipline of designing and structuring an application. This includes finding the appropriate application architecture and definition of a domain model. As a result, the application model carefully describes the important aspects of an application. This includes reflection of concepts and structures from the problem domain.

**Modelling**
Design and verification of the application model is usually performed in a modelling language. *Unified Modelling Language (UML)* [18] is the currently most popular modelling language. UML is a General Purpose Language that includes a standardised collection of modelling technologies that incorporate knowledge on best-practices. Common tools available include class, use-case and state chart diagrams.
An application model embeds a domain model. This is an essential subset of the complete problem domain model as identified during the problem domain analysis. Note that we will refer to this subset whenever the term domain model is used. Clearly, an important design principle of the application model is to keep the domain model as a separate entity. Consequently, it is possible to differentiate between the parts of an application that are domain-specific and the parts that are domain-irrelevant. An illustration of this is found in Figure 3.2.

![Application Model](image)

*Figure 3.2 Overview of the domain model and its context*

The purpose of the domain model is to make problem domain computations or representations possible. A domain model is usually the result of an extensive problem domain analysis. As noted, several analysis approaches exist, however, they all have one common superior goal. That is, to identify and describe the concepts and structures of the problem domain in the best possible manner. A domain model usually consists of a class model, also known as an object model, which reflects the properties of the problem domain. A concept is represented as a class, while the relations are represented using a combination of generalisations, aggregations, compositions and associations. Integral concept properties are expressed using class attributes. The domain model represents the computation core of a software application. It defines the true purpose and extent of the application. That is, what kind of computations the application supports. Traditionally, the initial version of an application model is non-executable. However, tool integration has diminished the boundaries between non-executable and executable models.

### 3.4 IMPLEMENTATION

In the implementation phase, an executable version of the application model is expressed by means of a programming language. It is important to note that this implementation still represents the same application model and abstraction level as expressed in the modelling language. However, there are some differences worth mentioning. The most obvious difference is that concepts in a programming language often are described in a textual manner, whereas the same concepts in a modelling language are visualised using boxes, lines and symbols. This usually makes a modelling language better suited for design of the application model. Another prominent difference is that a programming language supports executable models and runtime computations.
4 Language design

Designing a language is quite different from creating a traditional software application. There are two major differences. Firstly, creating a language requires a higher degree of formalisation. Secondly, a language comprises concepts on a higher abstraction level than a traditional application. In this chapter, we will elaborate on these differences and see how languages can be formalised using a meta model architecture.

4.1 META MODELLING

Designing an application in the traditional sense using GPLs is mostly concerned with application modelling in a modelling language and subsequent implementation of the same application model in a programming language. Language design, on the other hand, implies working on a higher abstraction level, formally known as the architecture level or meta model level. We will base our reasoning on the traditional meta model architecture [1]. This architecture is based on the OMG Meta Object Facility (MOF) [19] standard which is regarded as the de facto standard for language and tool design in the software industry. It identifies four meta levels, known as M0, M1, M2 and M3. These are elaborated below.

- M0 represents the lowest and most concrete meta level in the MOF architecture. It deals with application data like instances of classes and data from database table rows. Note that this level may contain the runtime representation of a system.

- M1 deals with application modelling. In other words, how an application is represented and structured in some kind of language. This level is also known as the model level.

- M2 contains the meta model of the model / program found on meta level M1. The meta model defines the language by specifying its syntax and semantics (if a rich meta model is used). Another name for this level is meta model level.

- M3 defines the meta-meta model whose purpose is to describe a general meta model. In other words, the specification of the concepts a meta model can use and how these may be combined to form a legal meta model.

An overview of the MOF meta model architecture is found in Figure 4.1. Each successive step upwards in the meta model architecture increases the level of abstraction and generality. Specifically, each meta level contains instances of concepts defined at the above meta level. These concepts are expressed using classes. Consequently, this is
known as a classification hierarchy. That is, application data of M0 constitutes instances of the concepts found in the application model M1. This application model contains instances of the language meta model concepts defined on M2. Furthermore, the elements in the meta model are instantiated from the meta-meta model at level M3. In addition, the meta model architecture ensures that structural constraints are preserved. For instance, the application model found at M1 has to fulfil the structural requirements imposed by the meta model found on M2, and so on.

The aforementioned relationships between the different levels of the meta model, known as classifier relationships, are of great importance. It makes it possible to reason about all languages by means of the standardised meta model architecture. Languages which comply with the MOF standard share a common fundament. Thus, it is possible to develop language tools and metaware that apply to all languages. Another benefit is the possibility of combining several meta models to achieve one resulting meta model, and the ability to perform mappings between meta models. A mapping basically means to apply a relationship or perform a transformation between different models found on level M1. In other words, languages can be unified. Internal mappings also exist between the components of a language. An example is the relationship between the abstract and concrete syntaxes of a language.

As noted previously, an important part of traditional software development is creation of an application model. This application model resides on level M1 of the MOF architecture. On the contrary, designing a language is mostly concerned with meta level M2, and the creation of a meta model. A meta model defines the syntax and semantics of a language, and thus provides concepts for creation of different application models. Using a custom language will then correspond to creating an application model in the same manner as expressed in the traditional development process.

Creating a meta model for definition of a language is not a straightforward task. A rich meta model consists of five major parts known as:

- Abstract syntax
- Well-formedness rules (static semantics) and meta operations
- Concrete syntax
- Semantics
- Mappings

4.1.1 ABSTRACT SYNTAX AND WELL-FORMEDNESS RULES

A language consists of a collection of constructs. These constructs correspond to pre-defined concepts that are used to create models in the language. The purpose of the abstract syntax is to define the concepts of a language and express how these concepts are related. In addition, it contains information on how the concepts may be combined to form legal models. This information is known as well-formedness rules, or static semantics, and is usually expressed using the Object Constraint Language (OCL) [20]. What kinds of concepts included in a language depends on the type of language. A GPL will, as the name suggests, provide a set of general concepts. On the contrary, a DSL contains representations that directly reflect concepts found in its target problem domain.

We will exemplify the meta model characteristics and abstract syntax by focusing on two known concepts in programming known as if statements and classes. A simple meta model for a traditional if-statement is found in Figure 4.2. Only the abstract syntax of the
A meta model for if-statements is shown. 

![Figure 4.2 A meta model for if-statements](image)

An if-statement is always associated with one expression, and at least one statement. The abstract syntax does not express any semantics or meaning, even though some information may be extracted from naming and class attributes. In theory, this implies that any interpretation may be given to the concepts If, Expression and Statement. In this case, however, the purpose is to understand how a meta model can be used to express language constructs. Therefore, we will refer to the natural interpretation of the concepts as defined in the traditional if-statement semantics. The associated semantics basically says that:

Expressions always evaluate to a boolean value which determines if the associated statements are to be executed.

A syntax diagram and EBNF-grammar for the if-statement is included as reference. It is found in Figure 4.3.

![Figure 4.3 Syntax diagram and EBNF-grammar for if-statements](image)

The keyword if and the braces are terminals, while <expression> and <statement> are non-terminals.

We will consider another more comprehensive meta model. Figure 4.4 shows an extract of the Java meta model. Specifically, this meta model segment expresses class contents, as described in [21]. Most of the class attributes are excluded from the figure for the sake of clarity.
As can be seen in Figure 4.4, a JavaPackage can contain any number of Java classes. Each JavaClass may declare inner classes, and contain any number of Fields and Methods. A Method is associated with two types of parameters, specified as inputParameters and returnParameter. Furthermore, a Method is associated with one or more instances of JavaClass, denoted javaExceptions. It can also be seen from the diagram that arrays are specialised JavaClasses. Again, Figure 4.4 only refers to the syntactic structure of the language, and does not reveal any meaning. Nevertheless, in this case it is easy to acquire a mental interpretation of the meta model because of the familiar concept names and the known context of Java classes.

In many cases, it can be desirable to express additional constraints on the concepts of the meta model. A language like OCL expresses such constraints in a formal and unambiguous manner. Adding a constraint to the Java meta model segment, as visualised in Figure 4.4, is a straightforward task. For instance, it may be desirable to put a constraint on the objects of the Field concept. In this case, we want to specify a requirement for the name attribute of Field. Note that the Field concept inherits from a class known as NamedElement, and thereby inherits an attribute called name.

The constraint should be:

All field names should include the substring “_field”.

This requirement is expressed by the following OCL code:

```ocl
context Field
@constraint ValidFieldName
+fields.name->includes("_field")
end
```

As might be expected, this code adds a constraint to the meta model imposing that all fields in Java programs should have a name that includes the substring “_field”. Notice that the inherited attribute name is referred in the OCL code.
4.1.2 CONCRETE SYNTAX
Most languages provide some kind of notation which is mapped to the underlying abstract syntax (and semantics). This notation, formally known as concrete syntax, makes it possible to express and create models in the language. There are two main groups of concrete syntaxes: textual syntax and visual syntax. The former type is usually chosen for programming languages, while the latter is adapted by modelling languages. A distinct advantage with textual syntax is its ability to express complex models with a great amount of details. On the other hand, the syntax can be quite difficult to comprehend if models are large. Conversely, visual syntaxes use diagrammatical views which make it easy to maintain overview and manage large models. Nevertheless, too many details may clutter the model. Clearly, a combination of the two types of syntaxes offers the most advantageous approach. Examples of concrete syntaxes are depicted in Figure 4.5.

```java
public class Mapping
{
    private String name;
    private static int count;
    private Vector<Item> items;
    ...
    public void setName( String name )
    {
        this.name = name;
    }
}
```

*Figure 4.5 Examples of textual and visual syntaxes, in the form of Java code and a class diagram*

4.1.3 SEMANTICS
A language needs a precisely defined meaning of its concepts and sound interpretation. This is known as the language’s semantics. It consists of a detailed description of the concepts and how they can be used in a correct manner. This includes a precise definition of the language’s capabilities and limitations. An important characteristic of the semantics is that it should be fairly easy to understand. Naturally, this is critical in order to use the language properly. In some contexts it is appropriate to differentiate between the formal semantics of a language and the implemented semantics. Implemented semantics is usually expressed using some kind of modelling or programming language which supports computation.

4.1.4 MAPPINGS
It can be desirable to integrate different languages. This can be done using mappings. A mapping consists of relations which comply with three major types: translation, semantic equivalence and abstraction, as explained below.

- Translation means that a concept in one language is translated into a concept in another language
- Semantic equivalence exists if a concept defined in one language has the same semantic meaning as a concept defined in another language
- Abstraction expresses a relation between two languages at different levels of abstraction
An illustration of the different mapping types is found in Figure 4.6.

Figure 4.6 Illustration of the different mapping types

Figure 4.6 illustrates three languages with simplified application models and the relationships between them, known as mappings. Each coloured piece represents a concept in the enclosing language. As an example, the pieces may be understood as classes that represent concepts found in the languages’ problem domains. Note that the concept referred in the semantic equivalence relation exists independently in both language A and language B. We will later use a combination of the mapping types identified to weave languages together.
5 Domain-Specific Languages

Domain-Specific Languages are the main focus point of this thesis. Here, we will shed some light on this type of languages and describe their role in today’s software industry. For the most part, we will consider DSLs as programming languages.

5.1 Inception and Historical Context

_A special kind of beauty exists which is born in language, of language, and for language._

_Gaston Bachelard, French philosopher_

Human languages have evolved in tenth of thousands of years. The early languages were small with a limited degree of comprehensiveness. However, they fulfilled their purpose of being able to accurately describe and explain the artefacts of the ancient world. An example of such a language is the figurative language of the past Egyptians, known as the hieroglyphs. This written language, estimated to have its origin around 4000 B.C., comprised a set of highly expressive figures used to represent the concepts of the past world.

Even though the hieroglyphic language is long ago extinct, it still represents an analogy to a Domain-Specific Language in computer science. Both the hieroglyphic language and a Domain-Specific Language share some important characteristics. Firstly, both languages have a vocabulary and language constructs limited to a certain set of concerns or domain. Secondly, they are both highly expressive in describing concepts in this domain. (It may be argued that the domain of the hieroglyphic language was in fact the entire known world. Nevertheless, the hieroglyphic language still represents a DSL with regard to the rich and complex general languages of the present world.)

Domain-Specific Languages have become a popular research area the recent years. However, as the aforementioned historical epistle illustrates, describing domain-specific artefacts and concepts using dedicated languages is not a new invention. The notion of Domain-Specific Language is often associated with research performed during the last decade. On the other hand, languages designed for expressing specific concerns have already been around for half a century. An example of such a language / formalism is the well-known Backus Naur / Normal Form, also known as BNF. BNF was invented in 1959 by the computer scientist John Backus. It is still regarded as the standard for describing the syntaxes of programming languages [22].

DSLs can be defined this way:

_A Domain-Specific Language is a language tailored for computation of specific concerns in a confined problem domain. This is achievable using domain-specific programming constructs and notations that abstract the concepts and relations present in the target problem domain._

In essence, concepts of the problem domain are reflected in first-class language
constructs. These programming constructs represent the building blocks of the DSL and are combined to model the domain-specific concerns. A DSL usually only defines constructs that are related to its target domain. That is, no general constructs. Consequently, it is difficult if not impossible to represent concerns outside of this domain. On the other hand, a DSL is capable of expressing concerns in its associated domain much more accurately and efficiently than GPLs. In addition, DSLs are often easier to use and maintain.

5.2 CHARACTERISTICS OF DOMAIN-SPECIFIC LANGUAGES

One of the peculiarities with DSLs is the pre-defined set of domain-specific constructs used to create programs, which implies that concerns are expressed in terms of pre-defined abstraction levels. This is on the contrary to GPLs, where general programming constructs are combined to create problem domain representations on an arbitrary abstraction level. Furthermore, the number of possible programs that may be modelled using a DSL is limited due to the pre-defined set of constructs. Each construct has its distinct role with respect to other constructs of the language. Thus, there are clear constraints on how constructs can be combined to form legal programs.

An implication of this pre-defined set of domain-specific constructs is increased productivity [23]. Concerns in the problem domain are easier to model and express than in GPLs. In general, programs of DSLs are more easily understood. This increases the reliability and quality of models. Conversely, general programming constructs of GPLs may obscure the true intention of a program. Moreover, DSLs make it possible to validate models and perform error handling against incorporated problem domain knowledge.

5.3 ADOPTION OF DOMAIN-SPECIFIC LANGUAGES

GPLs, like Java, C++ and C#, are by far the most popular and acknowledged programming languages in today's software industry. Contrary to DSLs, these languages are capable of performing computations for any given problem domain. This is achievable by combining general programming constructs and using abstraction mechanisms like classes, enums and structs.

There are many reasons why GPLs are dominating the software industry. The most evident reason is difficulties concerned with development of high quality DSLs. Naturally, creating a DSL requires both skills in language development and domain expertise. This knowledge is often shared between different stakeholders in a project. Thus, user participation and good communication are required. Unfortunately, there are many examples of traditional software development projects where these requirements have not been fulfilled. As a result, the software applications may have been cumbersome to use and not been capable of reflecting the problem domain in an intuitive and productive way. Specifically, the root cause for this is the differences in perspectives used by the stakeholders. A software developer sees an application from a technical point of view, whereas a domain expert often has a more practical and simplified understanding of the matters in question. Communicating and combining these different points of views have to all appearance proved to be more difficult than it may initially seem.

Another prominent reason to the stronghold of GPLs is the availability of tools and metaware for large-scale software development, as pointed out in [24]. Development of high-quality tools requires a great deal of resources and effort. Consequently, it is
unprofitable to create tools for languages that are used only by a small group of people. Furthermore, GPLs have the added benefit of being embraced by the open-source communities. These communities have played a central role in software development the last decade. One reason for the success of open-source software is focus on a relatively small number of GPLs, which brings us to the third major reason why GPLs are more widespread than special-purpose languages.

Every programming language imposes certain requirements with regard to training, language support and standardisation facilities. Ideally, this is something which should co-exist and complement use and evolution of every programming language. However, DSLs often have small user groups which may result in suppression of these issues. Consequently, the level of language acceptance is reduced. A new language comprises a set of constructs that comply with certain user patterns and methodologies. Managing a new language to the degree of efficient usage usually takes a considerable amount of time. Moreover, only stakeholders directly associated with the application area of a new language may want to invest resources in it. Thus, an immediate result is the lack of reception in the industry and the diminution of possible usage contexts.

For all the aforementioned reasons, frameworks and program libraries are often chosen instead of DSLs. Many Domain-Specific Language projects end up as an Embedded Domain-Specific Language, which more closely resembles a framework than an independent language. Some projects are even postponed indefinitely. We will later analyse frameworks and DSLs with respect to a set of essential properties and elaborate on the similarities and differences between these approaches.

There are variations of DSLs including Domain-Specific Modelling Language (DSML) and Visual Domain-Specific Modelling Language (VDSML), etcetera. We will not differentiate between the variations. Note that EDSLs are not considered in the main content of this thesis. A discussion of embedded languages is found on the accompanying CD.

5.4 WHEN TO DEVELOP A DSL

A DSL can offer substantial benefits with regard to GPLs. Deciding whether to design and implement a DSL is, however, a difficult task. There are many factors that should be considered in order to make a correct decision. The most important criteria relate to improved economics, easier maintenance and improved user experience [23].

There is a usual comprehension in the industry that every business decision made should eventually result in improved economics. One of the most fundamental initiatives in this regard is to ensure that costs are minimised by increased efficiency and productivity. This can be done by identification and analysis of tasks that are often repeated. The goal is to reduce cost, and at the same time increase the quality of the concerned products or services. Naturally, this initiative also applies to software development. A collection of patterns for making the correct decisions about DSL development is identified by [23]. Some of the concerns that are important to evaluate in the decision process are related to:

- Analysis of programming tasks that share properties
- Identification of architecture
- Interaction and GUI
- Increased usability
Analysis of programming tasks that share properties
Using GPLs often implies programming of entities that share a set of properties. Identifying these entities and supporting automatic code generation will increase the productivity and reduce the number of tedious programming tasks. A DSL transformational system can support automatic code generation.

Identification of architecture
Many applications share a fundamental design and architecture. A DSL may offer means of architecture specification which simplifies the process of structuring applications. A similar application area is modelling of data structures, which often is complex in nature. Furthermore, a DSL may express how data structures are traversed in a better way than what is possible with a GPL.

Interaction and GUI
A DSL can make specification of complex or repetitive input to a target system more user-friendly. In addition, it can help specifying configuration and adaptation options of a system.

Increased usability
Communicating specification details to developers can introduce misunderstandings and inconsistencies. The reason for this is that the communication is informal which allows for interpretation and use of judgment. DSLs, however, make it possible to specify aspects of the problem domain in a formal way. In other ways, in the matter of pre-defined semantics which all stakeholders can agree on. Moreover, DSLs reduce the required programming expertise needed to create sound programs, which potentially implies a broader user base.

A DSL is strongly coupled with its associated domain. Hence, using a DSL to create an application model basically means creating a domain model. It is important to emphasise that a DSL meta model does not correspond do a domain model, but a set of constructs that can be used to create a set of different application / domain models. This distinction is of utter importance. An illustration of how a GPL and DSL adapt to the MOF architecture is found in Figure 5.1.

![Figure 5.1 GPLs and DSLs in the context of the MOF meta model architecture](image)

There are two variants of MOF know as Essential MOF (EMOF) and Complete MOF (CMOF). In this thesis, we will use Ecore. Ecore is the implemented meta model architecture (meta-meta model) of the Eclipse Modelling Framework. Ecore describes similar concepts, and complies with structuring and principles of EMOF. We will not elaborate on the differences in this thesis.
This chapter identifies and describes software evolution. First we will define the evolution concept and use some common scenarios to exemplify the practical consequences of software evolution. Then, we will elaborate on how software evolution influences languages and their models, and why it is important to consider software evolution when designing and implementing a custom language.

### 6.1 THE NATURE OF SOFTWARE EVOLUTION

A common understanding among many developers is that the notion software refers merely to the programs written in some kind of programming language. However, software is more than just programs. Seen from a general perspective, software constitutes both the model and its corresponding meta model. Even the meta-meta model is considered as part of the software. In other words; the notion of software includes the levels M1, M2 and M3 with respect to the MOF meta model architecture. Consequently, it is possible to reason about software in a much broader sense [5]. Even though this initially might seem a bit strange, it makes perfect sense. We will base our further discussion on traditional software development, and explain the notion of software from the perspective of software evolution. Java is used as a reference language.

A typical Java application consists of classes that represent problem domain concepts and program control logic of some kind. These classes are built using formalised Java programming concepts such as methods, statements and means of structured control, like while and for constructs. Most developers know that applications change with time. There are basically two reasons for this. Firstly, changes in the problem domain imply that the domain model, which is incorporated in the application, has to change as well. Secondly, application code is subject to change due to demands regarding efficiency, security and application architecture. All changes to the application are covered by a term known as model evolution. We will return to this term later. At this point it is sufficient to know that applications, or models in general, evolve with time.

Models are, however, not the only type of entities that evolve. Languages change with time as well. For instance, it may be desirable to include new constructs or perform refinement of existing language concepts. An example of the latter was the introduction of generics in Java 1.5. Clearly, this language update imposed certain requirements with respect to compatibility. As an example, applications written in an earlier version of the language should still compile using the new JVM and work as expected. Even though the introduction of generics has proved to be successful with regard to compatibility issues, this is not always the case. Incorporating new concepts into the meta model is difficult. Preserving backwards compatibility is even more challenging. This is also the case with metaware tools that are based on the meta model. Changes in language and metaware tools, as functions of time, are known as language evolution and tool evolution, respectively.

Meta-meta models do also change, even though the rate at which these change is considerably lower than what is the case with models and meta models. We may refer to these changes as meta-meta model evolution. There also exists evolution at the instance level of an application. This can be exemplified by a change in program state due to program execution. Instance level evolution is included in this context merely for the sake
of completeness.

A common characteristic of a meta-level architecture, like the MOF architecture, is its organisation in a classification hierarchy. A consequence of this structure is the forced dependencies between the different abstraction levels. For instance, a model instantiates concepts found in its meta model. The type of concepts the model can instantiate, and the interrelationships between these concepts, are entirely determined by the meta model. Changes in the meta model have to be propagated to the model in order for the model to be compatible with the meta model (preserving conformance). A similar dependency exists between the meta-meta model and the meta model. Consequently, models created in some kind of language depend directly on the meta model and indirectly on the meta-meta model. It clearly makes sense to include these higher abstraction levels as well when talking about software. Put in another way, software = appliware + metaware, as explained in [5]. Appliware can be seen as the part of the software which constitutes the actual application, while metaware can be seen as the software that makes the appliware realisable.

In practice, most of the MOF meta-level architecture is represented in software. M1, known as the appliware, consists of the application. M2 comprises metaware tools like compilers and debuggers, which are based on a meta model. M3 is also considered as metaware. However, concepts covered by M3 are usually only found implicit in software. For instance, tools that are based on a meta model have to conform to meta-meta model concepts, even though these concepts are not necessarily explicitly instantiated in code. Note that the instance level M0 is not represented in or considered as software. It comprises a model state, usually found in memory (or other forms of data storage).

Software evolves on several levels. Naturally, evolution of software should be treated as a multi-facetted phenomenon. The reason for this is simple. Software on each meta-level evolves asynchronously both with regard to rate of change and compatibility issues. This results in co-evolution issues and software erosion.

### 6.2 Impacts and Impact Analysis

Software consists of atomic entities that work together. Examples of such entities are functions, objects and components. While the abstraction level of these entities differs, they all share a common set of interaction patterns. These interaction patterns consist of relations that tie the software together. For instance, an object-oriented application may be understood as a complex network of communicating objects. This can be visualised as a graph where the nodes represent the objects, and edges constitute the relations. This is depicted in Figure 6.1.

![Figure 6.1 Network of communicating objects](image)

A similar graph can be created for all kinds of software regardless of the concepts that are employed. Naturally, there are several kinds of relations within software. Some of the most common relations are:
- Instantiation relationship (between class and instance)
- Inheritance (between two classes)
- Function invocation (between defined function and function call)

Making changes to entities of these relations may potentially introduce impacts. This includes impacts between different abstraction levels, as structured by a meta-level architecture. Naturally, the consequences of changing one part of a relation might introduce inconsistencies that can be difficult to predict. Changing and extending software may prove to be very difficult. Impact analysis is the process of illuminating impacts that may occur in software and assess the consequences. An impact may be understood as a conflict between two interacting software entities. Impacts are classified in two main groups. These are horizontal impacts and vertical impacts. Changing an entity may result in several impacts, both horizontal and vertical.

6.2.1 HORIZONTAL IMPACTS
The term horizontal impact is common for impacts that occur within the same abstraction level. An example of such an impact is a function invocation where the declaration and invocation of the function appear on the same meta-level. A horizontal impact leads to horizontal inconsistency. This type of inconsistency must in most cases be resolved in order for the software to operate normally.

6.2.2 VERTICAL IMPACTS
Vertical impacts, on the other hand, refer to impacts that cross abstraction levels. A relevant example of vertical impacts is the inconsistency that may appear when an old compiler of a language is used to compile a program written in a newer version of the language. This type of inconsistency is known as vertical inconsistency. On the contrary to horizontal inconsistency, this type of consistency does not usually cause any momentary issues that must be sorted out in order for the software to work. (An earlier version of the compiler could be used in the above example.) Vertical inconsistency is the result of co-evolution.

6.3 CO-EVOLUTION
Co-evolution is a general concept that expresses how entities at different levels of abstraction evolve or change asynchronously. There are several dimensions in which co-evolution appears. One central dimension in the context of languages and models is known as the Meta Dimension. We have already referred to this dimension as the meta level architecture. In the continuation we will mainly focus on meta model / model co-evolution issues. These issues appear when the conformity relation between a meta model and its models is broken.
PART II
TECHNOLOGY
FOUNDATION
Several technologies are used to realise and verify concepts of this thesis. This chapter describes the most essential technologies and their roles with respect to the proof-of-concept applications. Details regarding usage of the technologies will be provided in the relevant contexts. Note that some references are made to concepts and applications that have not yet been introduced. Naturally, these will be elaborated in later chapters.

### 7.1 Eclipse
Eclipse is a complete workbench and *Integrated Development Environment (IDE)* for software engineering. An important aspect of Eclipse is the inherent plug-in system which can be used to extend default capabilities of the program. In other words, plug-in modules defining tools and other development artefacts can be installed to support new technologies. It is assumed that the reader is familiar with Eclipse or similar IDEs.

Eclipse is used in the design, development and verification of proof-of-concept applications and as a foundation for the LDE Platform by development of custom plug-ins.

### 7.2 UML
The *Unified Modelling Language (UML)* [18] is the most popular general purpose modelling language. It is managed by the Object Management Group. UML’s main purpose is to support software engineers in the many aspects of software design and development. This is achieved by providing different tools and diagrams for capturing and representing information. Some of the most popular notation techniques are class, state chart and use case diagrams. In general, the different diagrams can be categorised according to three groups: structure, behaviour and interaction diagrams. UML supports diverse software development methods.

UML class diagram notations are used to express meta models.

### 7.3 Java
Java 1.6 was chosen as the programming language for realisation of the proof-of-concept applications. There are some reasons to this choice. Firstly, Java is a popular and well-known programming language which features advanced programming concepts like reflection and dynamic proxies. Secondly, the Eclipse Modelling Framework is written in Java. Hence, it becomes easier to investigate issues regarding traditional DSL design with respect to design of composite DSLs, as supported by the proof-of-concept applications.

Java is used to implement the Ecommerce framework, most tools of the LDE Platform, file editors and the Runtime Environment.
7.4 Meta Object Facility

The Meta Object Facility (MOF) [19] is the Object Management Group’s meta modelling architecture. It was originally designed to define UML. MOF defines four meta levels or layers that can be used to formally describe various aspects of software. An important property of the MOF architecture is the classifier relationships between concepts of different levels. Specifically, a concept or element of one level is an instance of a concept on the above level. MOF and meta modelling have been treated in a distinct chapter due to the importance of these concepts. Please refer the chapter Language design.

The concepts and ideas presented by MOF are used to create and reason about meta models.

7.5 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) [17] is a framework for development of Java-based applications and tools. Specifically, EMF supports Model-Driven Engineering. That is, EMF provides code generation facilities that can be used to build the implementation of an application from a structured model. For instance, from an application model expressed using UML. Please refer chapter Traditional software development. Clearly, this increases productivity and standardises applications with respect to architecture. Other features of EMF are model notification, model validation, persistence and a generic reflective API for manipulation of EMF objects. Some of these features are used in a later chapter where we elaborate on the construction of a traditional DSL. EMF defines a meta-meta model known as Ecore. This model is similar to EMOF. It has been used as basis for all languages described in this thesis. It comprises concepts like EClass, EReference, EAttribute, EOperation and more. (Their names suggest their interpretation and use.) We will return to these concepts later. Note that EMF models (Ecore) can be used separately from the EMF framework functionality. In that case, Ecore is observed merely as an independent meta model architecture based on XML Metadata Interchange (XMI).

EMF is used to create a traditional DSL and define language meta models (Ecore).

Note that MOF and EMF are designed to meet different goals. Hence, the design decisions differ. The differences are of less importance with respect to the work presented in this thesis. On that note, it is possible to translate EMF models to MOF and vice versa. This is discussed in [25]. EMF can also read and write serialisations conforming to EMOF. Refer chapter Language design for details on meta modelling.

7.6 Eclipse Tools

Eclipse Modelling Framework Technology (EMFT) is a project that addresses technology for complementation or extension of EMF. Ecore Tools [26] are an EMFT component providing a complete environment for creating, manipulating and maintaining Ecore models. This component can be installed in Eclipse to simplify modelling tasks by providing a graphical editor. Ecore tools support multi-diagrams, refactoring capabilities and more.

Eclipse Tools are used to create and edit meta models conforming to Ecore.
7.7 ATLAS TRANSFORMATION LANGUAGE

Atlas Transformation Language (ATL) [27] is a language designed to define and execute model transformations. It is developed by the Atlas INRIA & LINA Research Group. Specifically, ATL provides declarative and imperative constructs that can be used to specify how a target model is constructed from a set of source models. This is achieved by creating a set of mappings between source model properties and target model properties, organised as rules. Three model handlers compatible with EMF, MDR and UML2 are provided by ATL as default. Consequently, ATL is compatible with models and meta models conforming to the Ecore meta model architecture. ATL’s main purpose is to address issues related to Model-Driven Engineering (MDE).

ATL is used by the ATL Preprocessor (one of the model transformers) of the LDE Platform.

7.8 KOMPOSE

Weaving of meta models is a central point of this thesis. This is achieved using a Model Development Kit (MDK) known as Kompose [11]. It is created by the Triskell team in the IRISA / CSU MDE Research Group. In detail, Kompose is a specialised generic model composition tool based on a generic framework for automatic model composition [11]. There are two major concerns addressed by the generic framework. These are matching and merging of models, as described by a matching operator and merging operator. Kompose supports composition of Ecore models. This has been achieved by creating a custom matching operator and composition strategy. The merging operator has been reused from the generic framework. Kompose is implemented using Kermeta [28] - a language for meta model engineering.

An important aspect of the generic framework is the ability to specify domain-independent composition directives. These can be used to force matches, disallow merges and override default merge rules. There are two main types of directives known as Create and Change (differentiated in Remove, Add, Set). These can be applied pre-merge or post-merge of a model composition process.

Kompose is used to compose the abstract syntaxes of meta models.
PART III
CONTRIBUTION
Chapter 8: Overview of the contribution

This chapter gives an initial overview of the different parts and components of the technical solution:

- The Ecommerce framework
- An atomic Domain-Specific Language
- The Language Development and Execution (LDE) Platform
- A set of composite Domain-Specific Languages

8.1 THE ECOMMERCE FRAMEWORK

A complete framework for creation of web-based ecommerce solutions has been developed. In this case, the framework is used to model web shops. Consequently, a set of services and graphical components related to development of web shop functionality are included as an addition to the native framework. The purpose of making the framework is to illustrate how frameworks can be used to support modelling of domain-specific concerns. In addition, it serves as a natural counterpart to Domain-Specific Languages. Later, we will also use this framework to define language semantics.

8.2 AN ATOMIC DOMAIN-SPECIFIC LANGUAGE

Several languages will be used to illustrate the different aspects of composite languages. We will start off by introducing a basic atomic language using the Eclipse Modelling Framework, and eventually extend this language using the LDE Platform.

Our initial language deals with the problem domain of cars. This domain has been selected due to its obvious and concrete concepts, and their intuitive interrelationships. The language will be used as reference throughout the thesis.
8.3 THE LDE PLATFORM

A complete platform for development of composite DSLs and execution of corresponding models will be introduced: the Language Development and Execution Platform. This platform provides tools, editors and environments for creation of language syntax and semantics, and complies with the concepts and ideas on composite DSL design and development. An overview of the LDE Platform is found in Table 8.1.

<table>
<thead>
<tr>
<th>Name:</th>
<th>Main purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weaver</td>
<td>Used to weave meta models.</td>
</tr>
<tr>
<td>Code Generator</td>
<td>Generates semantics stubs corresponding to abstract syntax.</td>
</tr>
<tr>
<td>Extractor</td>
<td>Extracts information from meta models which is used by the Injector. It may also be used to extract semantics types from the meta model directly, and meta information.</td>
</tr>
<tr>
<td>Injector</td>
<td>Uses information from the Extractor to inject semantics types.</td>
</tr>
<tr>
<td>Model Transformer</td>
<td>Transforms a set of input models to one output model.</td>
</tr>
<tr>
<td>ATL Preprocessor</td>
<td>Generates ATL transformations used for a composite model transformation.</td>
</tr>
<tr>
<td>Model Verifier</td>
<td>Verifies if a model conforms to a given meta model and custom rules.</td>
</tr>
<tr>
<td>Development Environment</td>
<td>A skeleton for development of composite DSLs.</td>
</tr>
<tr>
<td>Runtime Environment</td>
<td>A complete environment for parsing and execution of composite DSL models. Includes a standalone version.</td>
</tr>
<tr>
<td>Language Specification Editor</td>
<td>Used to define Language Specifications.</td>
</tr>
<tr>
<td>Type Specification Editor</td>
<td>Used to view and edit Type Specifications.</td>
</tr>
<tr>
<td>Meta Information Editor</td>
<td>Used to view meta model information.</td>
</tr>
</tbody>
</table>

8.4 A SET OF COMPOSITE DOMAIN-SPECIFIC LANGUAGES

Four composite DSLs will be used to describe the aspects and mechanisms of composite DSLs. Firstly, a composite language will be derived from the atomic Car language, as first built using EMF. Secondly, a Simulator language will illustrate incremental language design utilising the tools of the LDE Platform. Thirdly, a language for modelling of web shops will shed some light on how third-party semantics can be used by composite DSLs. Moreover, this example illustrates how the standalone Runtime Environment can be used. And, finally, we are going to design a General Purpose Language including constructs like class, function and expression. An overview of the main purposes of the different languages is found in Table 8.2.

<table>
<thead>
<tr>
<th>Name:</th>
<th>Main purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car, atomic</td>
<td>Illustrates a traditional approach to DSL design using the Eclipse Modelling Environment.</td>
</tr>
<tr>
<td>Car, composite</td>
<td>Exemplifies how atomic languages can be extended and customised.</td>
</tr>
<tr>
<td>Simulator</td>
<td>Describes a step-by-step process of developing composite languages using the LDE Platform.</td>
</tr>
<tr>
<td>Webshop</td>
<td>Sheds light on development of a more pragmatic and comprehensive language with the use of third-party semantics.</td>
</tr>
<tr>
<td>AGPL</td>
<td>A General Purpose Language used to illustrate interesting points regarding composite languages.</td>
</tr>
</tbody>
</table>
In this chapter we introduce the Ecommerce framework. The first section gives a conceptual overview and introduces the different parts which constitute the framework. Then we will illuminate how the framework is instantiated and extended. There are two purposes of making this framework. Firstly, a thorough understanding in framework design and utilisation is imperative in order to evaluate frameworks with regard to DSLs. Consequently, effort has been concentrated on creating a complete and highly usable framework. This includes focus on flexibility and extendibility. Secondly, the framework will be used to define semantics for the Webshop language (framework instantiation). This requires an in-depth understanding of the relevant framework concepts.

9.1 **AN OVERVIEW OF THE FRAMEWORK**

The Ecommerce framework provides the necessary application logic to build web-based e-commerce solutions. In this thesis, it is used to create web shops. One of the design goals was to make the framework as flexible and open as possible, and thus give the end-user of the framework a high degree of freedom. Another design goal was to create a complete framework to be able to investigate as realistic scenarios as possible. This implies the creation of a complete database abstraction with transaction support, and diverse graphical components.

There are several ways of utilising the framework. The most straightforward and easiest approach is to use the pre-defined services and control logic. However, it is possible to re-implement almost every part of the framework. This makes it possible to adopt the framework to different web shop solutions. In fact, the framework can be used for any kind of application conforming to the request-response design pattern, regardless of problem domain.

Most web shops use a flavour of HTML / CSS for presentation of information and Common Gateway Interface (CGI) variables to perform the interaction with the underlying web shop application logic, following a request-response scheme. We decided to follow this approach with the Ecommerce framework. J2EE Servlet technology was chosen as the interface towards the clients. However, the framework is not limited to this technology and it could easily be integrated with for instance Java Server Pages (JSP).

We start our investigation of the framework by giving an introduction to the main parts and address some of the most important concepts. However, it is outside the scope of this thesis to go into every detail of the implementation. Instead, details will be explained in later sections where appropriate. Interested readers may also consult the Javadoc in the source code for more details on the underlying logic and design decisions.

The framework can be organised into six groups. These are:

- Control logic (CL)
- Domain model (DM)
- Graphical User Interface and presentation (GUI)
- Services / Computation Model (CM)
- Data persistence (DP)
- Utilities (U)
We will now address each of these groups and give a brief explanation of their roles.

9.1.1 CONTROL LOGIC
There are mainly six aspects addressed by the control logic: session handling, event handling, service dispatching, navigation (state handling), persistence and presentation. Most of these aspects are represented by independently working modules. The interaction between the modules is governed by a Controller. We will briefly discuss the main components of the control logic.

Controller
All coordination and orchestration of the framework modules are performed by the Controller. This includes resolving of events, navigation, processing of tasks, and initiation of web page construction. All communication between the framework and the application servlet class is performed via the Controller.

Session handling
The task of the Session Handler is to manage all sessions initiated by the web shop. A new session is created when a client’s Internet browser connects to the web shop. This session is represented by a Session object. All Session objects are stored in the Session Handler and differentiated based on the HTTP session identification returned by the J2EE Servlet API. Session objects represent the users of the web shop.

Event and navigation handling
All interaction between the client’s browser and the framework is done by transferring CGI data. This data is subsequently resolved into states / pages, functions, tasks and services. Specifically, a page has a number of functions. Each of these functions is associated with one or more tasks. A task describes how services of the computation model are combined in order to process a given concern.

Service dispatching
An important design choice of the framework was to base all processing on services. There are two types of services that can be combined. These are business and data services. In addition, there is generic functionality for presentation. These services, together with the domain model, constitute the computation model. Service dispatching is used to ensure that the appropriate services are instantiated for each task. Seen from another perspective, the Service Dispatcher creates an instance of the computation model with the appropriate service classes based on resolved task. The Service Dispatcher can easily be revised to support dynamic class loading using a proxy (an earlier version of the framework used this approach).

Data persistence
Storing and retrieval of problem domain-specific data are performed using data abstraction. This ensures flexibility regarding how and where data is stored. For instance, data can be saved using databases, XML files or other media. The data storage and retrieval routines are the same regardless of the low-level medium used. Clearly, a middle-layer between the generic data access mechanisms and medium is required. We have illustrated storage using a MySQL database. Thus, a complete database abstraction has been developed.
9.1.2 DOMAIN MODEL
A domain model is an essential concept in software engineering. Its purpose is to reflect the entities and structures of the problem domain; making problem domain computations realisable. Typically, a domain model contains the most important concepts found in the problem domain, as identified during the problem domain analysis process. Specifically, the Ecommerce problem domain model comprises 18 classes suitable for reflection of web shop-specific concepts and data. An overview of the domain model can be found in Figure 9.1. Notice that a simplified visualisation is used since class properties are of less importance here. Please refer chapter Traditional software development for more information on problem domain analysis and selection of abstractions.

In short, each user session may result in the creation and placement of an order. This order is related to one customer and comprises information on a set of selected products, payment, delivery and shipping. Other domain model structures can be argued for, including support for creation of several orders simultaneously and more. Note that there are slight differences between the names used in the class diagram and the actual implemented domain model.

An important property of the domain model is generality. Thus, it should provide concepts that support a set of different web shops. These web shops can be differentiated based on variations of the instantiated domain model. Moreover, different web shops can implement subsets of the domain model. For instance, delivery and shipping are two optional components of an order. These can be excluded, implying that orders are sent to the registered address of the customer using a standard shipping method. There are also other properties like currency, language settings, credit options, and the likes relevant for web shops. Nevertheless, the properties reflected by the domain model suffice for our purposes. In other words, the domain model captures enough details to represent different web shops.

9.1.3 GUI AND PRESENTATION
A web page consists of different structures whose task is to align the content. There are three structures available: SPage, SSidebar and SContent. Each structure implements the Structure interface. Components are used together with the structures to construct web pages. A component is a limited graphical block that may contain functionality for
interaction with the web shop. Each component is represented by a class implementing the Component interface. Examples of components are the menu, the shopping cart and diverse forms for filling in domain-specific information like checkout and payment data. Structures and components can be reused for several web pages. A prior version of the framework featured factories. The purpose of these was to reuse structures and components without having to re-instantiate the respective classes. These factories were replaced by associating structures and components directly with pages and functions, respectively.

Structures and components are not an integral part of the framework. There are two reasons for this. Firstly, application design principles suggest separating GUI from the application core. Consequently, graphics and interaction functionality can be created independently of the framework. The only requirements are that the Structure and Component interface methods are implemented properly, and that components relate to the domain model in a sound manner. Secondly, there is often desirable to make custom graphics and interfaces for a specific web shop. This makes it redundant to include structures and components with the framework. However, for the sake of clarity, all classes related to GUI and user interaction are found in the .gui packages. Instead, these could easily be added to a library.

9.1.4 SERVICES AND COMPUTATION MODEL
In this context, the computation model means the classes containing logic for computation of problem domain-related concerns. There are two entities constituting the computation model. These are the services and the domain model. How these entities are structured and organised further define the capabilities of the computation model. Naturally, the services are closely related to the implemented domain model. There are many ways to organise a computation model. An important criterion, in this regard, is how computation is initiated. Clearly, considering the nature of the request-response scheme, a convenient structure is to organise the model in such a way that each request has its corresponding application code. This organisation has been maintained using services. Each service corresponds to one class. In principle, there are three types of services which together constitute a traditional multi-tier architecture. Specifically, the computation model comprises a presentation layer, a business layer and a data layer (the domain model resides on the business layer). This basic structure complies with Separation of Concerns. This concept underlines the importance of preserving focus on one aspect at a time of a given subject. Thus, making it possible to preserve the consistency of the aspect, and comprehend large and complex systems.

Initially, the presentation layer was associated with a set of presentation services whose task was to create web pages containing information to the user of a web shop. In a revised version of the framework, the presentation services were substituted with a generic mechanism for rendering web pages. Consequently, identification of specific presentation services is redundant. Instead, the different functions (menu, product list, shopping cart, and etcetera) available initiate the creation of web pages. This is more logical since there is a strong relation between a function and its visual presentation. As might be expected, the business layer of the computation model is associated with business services. Business services perform the business logic associated with each request initiated by the user of the web shop. Finally, the data layer of the model comprises a set of data services. Data services provide functionality for accessing the underlying data source, for example a database, XML file or other media for storing and
retrieval of data.

All of the aforementioned business and data services are custom-made for a specific purpose, for instance processing and persistence of checkout data or the task of showing the product list. Business services, together with Components, are the only entities that access the domain model. In fact, business services are the only entities which update it. This is important to keep the integrity of the domain model. That is, only a small number of entities change the model’s state.

Each event / request initiated by a user of the web shop is resolved to a specific state / page and function. Moreover, each function can be associated with an arbitrary number of tasks. There are two types of tasks; pre and post tasks. A pre task is processed prior to presentation (visualisation) of a function, while a post task is processed after some kind of action has been triggered (navigation or update). Each task, regardless of type, is associated with one or two services. That is, a task can potentially utilise one business service and one data service. Note that each service can be used for several tasks. Reuse and composition are promoted. The combination of services defines the processing capability associated with a given task.

Each request results in instantiation of the computation model. Clearly, a computation model instance is closely related to a specific processing that needs to be performed. In practice, each request corresponds to appropriate services and related domain model objects. By using a service-oriented, multi-tier architecture we have the benefit of being able to only implement the necessary services and combine them to achieve the desired processing. Furthermore, code is separated in distinct layers, which makes it easier to gain an overview and reason about the implementation.

9.1.5 DATA PERSISTENCE
There are several packages associated with persistence of data. These packages contain classes with functionality for storing and retrieving of problem domain-specific information. This includes storage of the domain model state. All communication with the underlying data source is performed using methods of the DataSource interface. This interface declares six relevant methods, as viewable in Table 9.1.

<table>
<thead>
<tr>
<th>Table 9.1 - The DataSource interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. public interface DataSource extends Transaction</td>
</tr>
<tr>
<td>2. {</td>
</tr>
<tr>
<td>3. public MetaObject addObject( DataObject dataObject );</td>
</tr>
<tr>
<td>4. public MetaObject addObjects( DataSet dataObjects );</td>
</tr>
<tr>
<td>5. public DataObject getObject( DataObject criteria );</td>
</tr>
<tr>
<td>6. public DataSet getObjects( DataObject criteria );</td>
</tr>
<tr>
<td>7. public MetaObject updateObject( DataObject dataObject, DataObject criteria );</td>
</tr>
<tr>
<td>8. public MetaObject updateObjects( DataObject dataObject, DataSet criteria );</td>
</tr>
<tr>
<td>9. }</td>
</tr>
</tbody>
</table>

As can be seen in Table 9.1, different methods for storing and retrieving domain model objects are available. This includes storage and retrieval of one object at the time, or several objects simultaneously. The default data storage for the Ecommerce framework is a MySQL database. Unfortunately, the MySQL database driver does not support storage and retrieval of own-defined objects. Therefore, a MySQL database abstraction has been
made which abstracts away the MySQL-specific procedures and makes it possible to store domain model objects directly in the database. Java reflection is used to access properties of the domain objects. These properties are specified using a set of data source mappings. In simple terms, data source mappings are used to relate properties of domain objects to relational tables in the underlying MySQL database. Consequently, using database abstraction and data source mappings ensure a general algorithm for persistence of any kind of domain objects. The database abstraction supports class hierarchies and inheritance. Classes whose objects should be persisted have to implement the interface `DataObject`.

Other kinds of storage technologies can be used for persistence of domain model objects. Examples of this are Hibernate in combination with MySQL / MsSQL and an object-oriented database, like Caché. The main reason why we implemented our own database abstraction is the desire for heterogeneity. We did not want to introduce dependencies to third-party technologies due to the conceptual nature of the framework. For instance, Hibernate or a similar database abstraction would not, in this context, add significant value to the framework.

9.1.6 UTILITIES
Some common utilities are defined as static methods. These are invoked from different parts of the framework.

9.2 FRAMEWORK INSTANTIATION
This section exemplifies how the framework is instantiated and configured. It is recommended that the reader is familiar with the concepts introduced in the preceding section *An overview of the framework*. Framework instantiation describes the process of instantiating framework concepts and fulfilling the dependencies between these concepts. Instantiation is performed by one or more application classes. Application classes (application model) define an application in terms of framework concepts. In this case, a servlet class will be used to create the web application. Hence, this class contains the necessary framework instantiation code. An overview of the different entities involved in a traditional web application based on the J2EE Servlet technology can be seen in Figure 9.2. Clearly, other technologies can be used to interact with the clients.

![Figure 9.2 Entities involved in a Servlet-based web application](image)

As can be seen in Figure 9.2, the servlet class extends the J2EE class `HttpServlet`. `HttpServlet` contains the basic functionality for performing full-duplex communication with Internet browsers. The servlet class implements two methods that are invoked when a CGI GET or POST request occur. These are `doGet` and `doPost`, respectively. The control flow always starts from these methods and is propagated through the rest of the framework. There are two approaches to utilise the framework depending on the degree of flexibility and customisation needed. We will discuss both approaches.
9.2.1 COMBINING PRE-DEFINED ENTITIES

The most straightforward manner of using the framework is to combine the available pre-defined functions and graphical components. Different combinations of these entities yield different web shops. Specifically, a configuration is used to describe a web shop. It comprises a set of states. Each state has an identification and references a page. Moreover, each page has a set of structures and functions (components). There are several pre-defined functions available: FCheckout, FConfirmOrder, FMenu, FPayment, FProductList and FShoppingCart. Each page can contain an arbitrary number of functions. Furthermore, a function can be used on an arbitrary number of pages. As an example, the FCheckout class is found in Table 9.2.

Table 9.2 - Overview of the FCheckout class

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>public class FCheckout extends Function</td>
</tr>
<tr>
<td>2.</td>
<td>{</td>
</tr>
<tr>
<td>3.</td>
<td>public FCheckout()</td>
</tr>
<tr>
<td>4.</td>
<td>{</td>
</tr>
<tr>
<td>5.</td>
<td>component = new CCheckout();</td>
</tr>
<tr>
<td>6.</td>
<td>}</td>
</tr>
<tr>
<td>7.</td>
<td>public String render( DataSet dataSet, Session session )</td>
</tr>
<tr>
<td>8.</td>
<td>{</td>
</tr>
<tr>
<td>9.</td>
<td>component.clear();</td>
</tr>
<tr>
<td>10.</td>
<td>component.setSession( session );</td>
</tr>
<tr>
<td>11.</td>
<td>if( !initialised )</td>
</tr>
<tr>
<td>12.</td>
<td>{</td>
</tr>
<tr>
<td>13.</td>
<td>for( Navigation n : navigations )</td>
</tr>
<tr>
<td>14.</td>
<td>component.addNavigation( n );</td>
</tr>
<tr>
<td>15.</td>
<td>initialised = true;</td>
</tr>
<tr>
<td>16.</td>
<td>}</td>
</tr>
<tr>
<td>17.</td>
<td>if( dataSet != null )</td>
</tr>
<tr>
<td>18.</td>
<td>component.addDataSet( dataSet );</td>
</tr>
<tr>
<td>19.</td>
<td>return component.getContent();</td>
</tr>
<tr>
<td>20.</td>
<td>}</td>
</tr>
</tbody>
</table>

As can be seen in Table 9.2, a graphical component CCheckout is instantiated at line 5. This component is used to interact with the web shop user, and visualise checkout-related information. The component is rendered by invocation of the method render(…), as defined between lines 7 and 20. In this regard, to render means to populate the graphical component with information. The content of the rendered component is returned from the method at line 19. This content consists purely of Internet browser compatible code, like HTML and references to CSS.
Let us review how the framework is configured. In detail, we will illustrate the configuration of a simple application consisting of one page with checkout functionality.

Table 9.3 - Excerpt from an example framework configuration

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>State checkout = new State();</td>
</tr>
<tr>
<td>2.</td>
<td>checkout.setID( &quot;i003&quot; );</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3.</td>
<td>// CHECKOUT</td>
</tr>
<tr>
<td>4.</td>
<td>Page checkoutPage = new Page();</td>
</tr>
<tr>
<td>5.</td>
<td>checkoutPage.setSidebar( new SSidebar() );</td>
</tr>
<tr>
<td>6.</td>
<td>checkoutPage.setContent( new SContent() );</td>
</tr>
<tr>
<td>7.</td>
<td>Function checkoutFunction = new FCheckout();</td>
</tr>
<tr>
<td>8.</td>
<td>checkoutFunction.setType( Function.CONTENT );</td>
</tr>
<tr>
<td>9.</td>
<td>Task tCheckout = new Task();</td>
</tr>
<tr>
<td>10.</td>
<td>tCheckout.setBSDef( CheckoutBusinessService.class );</td>
</tr>
<tr>
<td>11.</td>
<td>tCheckout.setDSDef( CheckoutDataService.class );</td>
</tr>
<tr>
<td>12.</td>
<td>checkoutFunction.addTask( tCheckout, Function.POST );</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>13.</td>
<td>Navigation n1 = new Navigation();</td>
</tr>
<tr>
<td>14.</td>
<td>n1.setName( &quot;Payment&quot; );</td>
</tr>
<tr>
<td>15.</td>
<td>n1.addSubTask( SubTasks.CHECKOUT );</td>
</tr>
<tr>
<td>16.</td>
<td>n1.setVariable( Navigation.ID );</td>
</tr>
<tr>
<td>17.</td>
<td>n1.setState( payment );</td>
</tr>
<tr>
<td>18.</td>
<td>n1.setVariable( Navigation.ID );</td>
</tr>
<tr>
<td>19.</td>
<td>checkoutFunction.addNavigation( n1 );</td>
</tr>
<tr>
<td>20.</td>
<td>tCheckout.registerNavigation( n1 );</td>
</tr>
<tr>
<td>21.</td>
<td>checkoutPage.addFunction( checkoutFunction );</td>
</tr>
<tr>
<td>22.</td>
<td>checkout.setPage( checkoutPage );</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>23.</td>
<td>Function paymentFunction = new FPayment();</td>
</tr>
</tbody>
</table>

A configuration for a simple web shop application can be seen in Table 9.3. Clearly, this is not a complete web shop, but it illustrates the nature of framework instantiation code. In a later chapter, we will see examples of more complete web shops.

A page intended for checkout functionality is instantiated at line 3. An instance of each available structure is added at lines 4 till 6. The function for checkout functionality is configured between lines 7 and 12. Specifically, the function should appear in the main content of the page, and be associated with a task utilising the services CheckoutBusinessService and CheckoutDataService. This task is of type post. In other words, it should be processed after the user has triggered an event, as defined on the corresponding graphical component - CCcheckout. Typically, the user fills in checkout information that should be stored when navigating to the next page. This storage operation is handled by the tCheckout task defined between lines 9 and 11. As can be seen between lines 13 and 19, navigation functionality is added to the checkout page. This functionality can be used to navigate to a payment page.

Note that the navigation n1 must be registered to the tCheckout task, as seen at line 20. This is a safety requirement. That is, only pre-defined actions (navigation paths) should have the privilege to initiate processing of a task. In other words, if line 20 was omitted, the tCheckout task would not have been executed when navigating to the payment page. Consequently, filled-in checkout data would not have been stored.
A future extension to the framework is support for configuration files. This would make it possible to describe all aspects of the framework configuration in an XML file or similar. We will later see how a DSL can be used to perform framework instantiation. Using a DSL, writing manual instantiation code is not necessary.

9.2.2 REDEFINING FRAMEWORK CONCEPTS
Alternatively, more flexibility can be achieved by redefining and specialising framework concepts. This can be performed by creating completely new functions, components and services or by subtyping the available entities. These entities are known as the framework’s variation points. Variation points, also referred to as hot spots, identify components of a framework that are designed to be replaceable [10]. It is also possible to provide a new domain model or specialise the existing one. In short, the framework is designed to support custom code. A new framework Controller can also be created if needed. However, this requires in-depth insight in order to use the framework correctly. That is, several framework classes have to be instantiated manually. In addition, managing dependencies between the objects of these classes can be challenging.

As an example, important aspects of applications, whose problem domain is within the genre of ecommerce, are security and logging of information. For instance, it can be desirable to log all transactions in order to later be able to trace purchases. Or, it can be required to store session data for acquisition of statistical information on the percentage of people who complete an order. These extensions can be provided by redefining the Controller and other relevant parts of the framework.

Note that alternative domain models, services, functions and components must be provided in order to use the framework for other applications than web shops. These entities are closely related and identify the problem domain of the framework.

9.2.3 CONFIGURATION OF DATA SOURCES
The framework has to be configured for its underlying data source. This is achieved by providing a set of connection details and creation of data source mappings. That is, descriptions on how to connect to the data source and how to store / retrieve problem domain objects in the underlying data structure. For instance, the properties of an object can be mapped to the columns of a relational database table. We will not go into detail on how this is done. However, a small extract of a data source mapping using MySQL is included as a reference. A data source mapping defines a relation between a domain model concept and a storage unit in the underlying data storage.

<table>
<thead>
<tr>
<th>Table 9.4 - Extract of data source mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. DataSourceMapping dataSourceMapping = new DataSourceMapping( &quot;Customer&quot;, DatabaseUtilities.TABLE_CUSTOMER );</td>
</tr>
<tr>
<td>2. dataSourceMapping.addMapping( new Mapping( &quot;customerID&quot;, &quot;id&quot;, Types.INTEGER ) );</td>
</tr>
<tr>
<td>3. dataSourceMapping.addMapping( new Mapping( &quot;firstName&quot;, &quot;firstName&quot;, Types.VARCHAR ) );</td>
</tr>
<tr>
<td>4. dataSourceMapping.addMapping( new Mapping( &quot;lastName&quot;, &quot;lastName&quot;, Types.VARCHAR ) );</td>
</tr>
<tr>
<td>5. dataSourceMapping.addMapping( new Mapping( &quot;phoneNumber&quot;, &quot;phoneNumber&quot;, Types.VARCHAR ) );</td>
</tr>
</tbody>
</table>
As can be seen in Table 9.4, objects of the domain model class Customer is stored to a relational table of a database by mapping its properties to the table’s columns. For instance, the property customerID of a Customer object is stored in a column named id, while the property firstName is stored in a column named firstName.

<table>
<thead>
<tr>
<th>Table 9.5 - Example of storage and retrieval of a domain model object</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Customer c1 = new Customer();</td>
</tr>
<tr>
<td>2. c1.setProperty( FIRST_NAME, &quot;Some Name&quot; );</td>
</tr>
<tr>
<td>// STORAGE</td>
</tr>
<tr>
<td>3. dataSource.startTransaction();</td>
</tr>
<tr>
<td>4. dataSource.addObject( c1 );</td>
</tr>
<tr>
<td>5. dataSource.commit();</td>
</tr>
<tr>
<td>// RETRIEVAL</td>
</tr>
<tr>
<td>6. Customer c2 = dataSource.objects( c1 ).getFirstElement();</td>
</tr>
</tbody>
</table>

Table 9.5 shows an example of how a problem domain object can be stored and retrieved. At line 1 a domain model object of type Customer is constructed. Its firstName property is set at line 2. Lines 3 till 5 start (and commit) a new transaction and stores the object in the underlying data source. Note that these methods are used regardless of the underlying data storage medium. Functionality for querying the data source is available. In this example it is used to search for a certain kind of Customer object, as seen on line 6. Each property of a problem domain object can be used to specify matching requirements for a desirable object(s). In our example, the customer object c1 is used as search criterion. Consequently, all customers with name “Some Name” are returned by the underlying data source.

9.3 THE PURPOSE OF THE FRAMEWORK

As we have seen, framework instantiation is an elaborate process. Extending the framework can also be challenging because internal dependencies must be addressed by the extensions. DSLs are more pragmatic in the way concerns are modelled. Hence, it is not necessary to manually write low-level instantiation code. We will later see how composite DSLs can be refined in a more intuitive manner than what is possible with frameworks, and compare the two approaches. Moreover, we will create a Webshop DSL that is semantically defined using the framework discussed in this chapter.
In this chapter we will illustrate the design process of a Domain-Specific Language using the Eclipse Modelling Framework. The purpose of this chapter is to get familiar with atomic language design and obtain a reference point with respect to composite language design. Obviously, it is not possible to delve into every detail of EMF. Consequently, four aspects of DSL development are covered:

- Creation of a language meta model with associated semantics
- Runtime reading / execution of a model in the language
- Extension of meta model syntax and semantics
- Possibilities and limitations of atomic language design in EMF

10.1 A DSL FOR MODELLING OF CARS

A traditional DSL may also be called an atomic Domain Specific Language. This suggests that all properties of such a language are expressed using one self-contained and atomic definition. This definition contains the language’s meta model and semantics, and eventually other entities like a concrete syntax. A DSL is static by nature. This implies that a DSL’s meta model and corresponding semantics can not be altered dynamically by the software engineers / users of the language. All updates to the language have to be explicitly performed by the language developer.

We will start off our investigation of language design with a simple atomic language for modelling of cars. In practice, such a language could be used to calculate different coefficients and values concerning cars. Or, it could be used in a car manufacturing context. This initial language is simple, but will illustrate how the mechanisms of EMF support design and development of a DSL. Later, we will discuss how this language can be elaborated by applying the concepts and principles of composite language design.

10.2 CREATING THE LANGUAGE META MODEL

A set of concepts related to cars has been identified as the result of problem domain analysis. These are Car, Engine, Chassis, Seat, Window, Wheel and Colour. Naturally, a real car consists of more components than what has been listed here. However, we will keep our Car language simple for the sake of brevity.

EMF provides an Ecore model editor that together with Ecore diagrams provides an efficient way of defining Ecore meta models. Meta models can be built in an incremental fashion by sequentially adding instances conforming to the Ecore meta-meta model classes like EClass, EReference and EAttribute. An Ecore meta model for modelling of cars, using the identified problem domain entities, is shown in Figure 10.1. Each concept has one or more basic attributes.
Figure 10.1 The Car meta model

As can be seen from Figure 10.1, a Car object is composed of objects conforming to the classes Chassis, Wheel, Engine, etcetera. Furthermore, the Chassis and Seat objects refer an arbitrary number of Colour instances.

EMF provides means of generating a model editor, model code and a JUnit TestSuite with associated test skeletons. Generation of these artefacts is initiated from a Generator Model. The plug-in projects of Table 10.1 can be generated using a Generator Model for the Car meta model. Note that in this context, our model is in fact a meta model. Thus, the generated code may be referred to as meta model code. We will use this distinction to avoid confusion.

<table>
<thead>
<tr>
<th>TABLE 10.1 - EMF PROJECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project:</td>
</tr>
<tr>
<td>emf.example.car</td>
</tr>
<tr>
<td>emf.example.car.edit</td>
</tr>
<tr>
<td>emf.example.car.editor</td>
</tr>
<tr>
<td>emf.example.car.tests</td>
</tr>
</tbody>
</table>

10.2.1 THE GENERATED META MODEL CODE

Each concept of the Car meta model has its own corresponding interface and class, as generated by EMF. These are all found in the emf.example.car project. Together these interfaces and classes provide the semantics for the meta model. car contains the interfaces, while car.impl contains the classes. The meta model code includes a couple of factories and a switch. Additions to the semantics may be provided by editing the respective classes and interfaces.

10.2.2 THE CAR META MODEL EDITOR

A Car meta model editor can be generated by EMF. This editor supports creation and validation of Car models. We will use this editor to create a model conforming to the Car meta model. Note that according to EMF, an object of a meta model class referenced with a non-containment reference needs to be instantiated in a separate model and then loaded as a resource in the main model. In this case, two instances of the Colour class have been stored in separate models and subsequently loaded as resources in the main model Car.car. These are named ChassisColour.car and SeatColour.car. Refer Figure 10.2.
Feel free to refer the meta model in Figure 10.1 for details regarding model structure.

10.3 READING A MODEL AT RUNTIME

Until this point we have seen how to define an Ecore meta model and a set of corresponding EMF projects. In addition, we created some models conforming to the meta model. Clearly, it is interesting to execute models and use them in computations. We will illustrate this by reading data from a Car model at runtime. This data can be used in arbitrary processing. A code excerpt, whose purpose is to read the Car.car model and print the number of horse power and torque to the console, is found in Table 10.2.

Table 10.2 - Code for reading from the Car.car model

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>ResourceSet resourceSet = new ResourceSetImpl();</td>
</tr>
<tr>
<td>2.</td>
<td>resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Factory.Registry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());</td>
</tr>
<tr>
<td>3.</td>
<td>resourceSet.getPackageRegistry().put(CarPackage.eNS_URI, CarPackage.eINSTANCE);</td>
</tr>
<tr>
<td>4.</td>
<td>URI uri = URI.createURI(Driver.class.getResource(&quot;Car.car&quot;).toString());</td>
</tr>
<tr>
<td>5.</td>
<td>Resource resource = resourceSet.getResource(uri, true);</td>
</tr>
<tr>
<td>6.</td>
<td>Car car = (Car)EObjectUtil.getObjectByType(resource.getContents(), CarPackage.eINSTANCE, Car.class);</td>
</tr>
<tr>
<td>7.</td>
<td>Engine engine = car.getEngine();</td>
</tr>
<tr>
<td>8.</td>
<td>System.out.println(&quot;Horse powers: &quot; + engine.getHp());</td>
</tr>
<tr>
<td>9.</td>
<td>System.out.println(&quot;Torque: &quot; + engine.getTorque());</td>
</tr>
</tbody>
</table>

Lines 1 till 7 create an EMF ResourceSet that is used by line 8 till 10 to acquire the model pointed to by the URI. Lines 11 till 13 acquire Java objects reflecting the model’s Car and Engine objects. Specifically, the Java objects are instantiated with data read from the Car.car model. Lines 14 and 15 print the results to the console.
10.4 EXTENDING THE META MODEL

It is fairly simple to extend or change parts of an Ecore meta model. An example of such extension is shown in Figure 10.3. Wheel is extended to represent a composite object composed of a Rim and a Tyre.

![Figure 10.3 Extension to the Car meta model](image)

A new Generator Model must be created in order for these changes to be incorporated into the generated code. All relevant code can then be rebuilt using this new model. Furthermore, some manual deleting or editing of generated files may be necessary depending on the type of update performed on the meta model. For instance, removing or renaming concepts in the Ecore meta model leaves previously generated files for old concepts.

Two new interfaces and two new classes are generated in our example. These are: Rim, Tyre, RimImpl and TyreImpl, respectively. Model values for Rims and Tyres can now be accessed in the application using the code of Table 10.3. In this case, we access the rim and tyre of the first wheel.

<table>
<thead>
<tr>
<th>Table 10.3 - Accessing the new model elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. EList&lt;Wheel&gt; wheels = car.getWheels();</td>
</tr>
<tr>
<td>2. Rim rim = wheels.get(0).getRim();</td>
</tr>
<tr>
<td>3. Tyre tyre = wheels.get(0).getTyre();</td>
</tr>
</tbody>
</table>

10.5 CHANGING THE SEMANTICS

Changes to the semantics can be performed by altering or adding code in the generated files. For instance, let us consider the generated code for the Engine concept. According to the Ecore meta model, this concept has two attributes: hp and torque. No explicit operations are declared. Consequently, the only access points to this concept’s semantics are setter and getter methods for the attributes, some helper methods and an overridden toString() method. No additional semantics than value storage is associated with this concept.

It may, however, be necessary to include some kind of computation in a concept’s construct’s semantics. In this example, it could be interesting to see the fictive ratio of torque per horse power (measured at 5000 rpm). An algorithm calculating this could easily be incorporated into our generated code by adding a method named getRatio(). Both the interface and implemented class of Engine need to be updated to accommodate this new semantics addition. The necessary updates are shown in Table 10.4 and Table 10.5.
Table 10.4 - Semantics addition to Engine

1. `public double getRatio();`

Table 10.5 - Semantics addition to EngineImpl

1. `public double getRatio();
2. {
3.     return torque/hp;
4. }

Invoking the `getRatio()` method is easily done from our driver application by using the Engine object. Refer Table 10.6.

Table 10.6 - Invoking the getRatio() method

1. `System.out.println("Ratio: " + engine.getRatio());`

It is also possible to add an operation directly to the language’s meta model. This operation is reflected in the generated meta model code as a skeleton method.

10.6 POSSIBILITIES AND LIMITATIONS USING EMF

EMF supports language development in a reasonable manner. A meta model and the corresponding generated code can be extended and revised easily according to new requirements. However, all updates and changes have to be explicitly performed in the language’s meta model and semantics. Incorporating additional meta models or semantics into the language is not possible. That is, EMF does not include tools or functionality for supporting composite language design.

A great feature is the non-destructive code generation. Basically, this means that manually added code is left untouched during regeneration of the meta model code. To achieve this, manual and generated code are differentiated using an annotation named `@generated`.

As shown, an interesting feature of EMF is support for generation of a unique model editor. This editor provides a concrete syntax that can be used to create models conforming to a specific language. Naturally, only classes defined in the given language’s meta model can be instantiated in the editor. Furthermore, all models created are stored using a proprietary file extension. This makes it easy to identify models. Conversely, the general .xmi file extension does not tell anything about what kind of meta model a model conforms to.
11 The concept of composite DSLs

A traditional DSL comprises language constructs that can be used to express concerns in its problem domain. Each of these language constructs has a pre-defined purpose and role with regard to the concepts of the domain. Refinement of a DSL may be difficult to achieve due to this rigid mapping between constructs and domain concepts. This thesis suggests a new approach in custom language design: to use aspect-oriented weaving to support extension and customisation of DSLs. This kind of refinement is achieved by incorporating one or more aspect languages into a primary language. Consequently, a DSL can be elaborated to meet evolving requirements.

In this chapter we will discuss this new approach and illustrate how the different mechanisms of aspect-oriented language weaving may prove to be very useful when modelling domain-specific concerns. An in-depth discussion will be presented in a later chapter.

11.1 MOTIVATION

A traditional DSL consists of first-class meta classes whose instances can be combined to form models in the language. These classes / constructs have to be general enough to reflect the important properties of the problem domain, since it is not possible to further refine the constructs for use in special-case scenarios. In principle, the only way to provide means of customisation is to include additional constructs. These constructs have to be pre-defined by the creators of the language. Hence, a DSL proves to be very static and difficult to customise for software engineers, if even possible. This is in contrast to development using frameworks where a software engineer is able to extend and specialise already defined classes to provide the necessary extensions or customisations. Clearly, this is a significant advantage using frameworks compared to DSLs. Please refer the Frameworks versus DSLs chapter for a thorough discussion of these issues.

Consequently, if a DSL does not meet certain (critical) requirements it simply can not be used. Traditionally, there are three solutions to this problem:

- A new version of the DSL may meet the requirements
- There might be an alternative DSL that can be used instead
- The task at hand can potentially be solved using a framework

All of the above solutions are questionable. We will discuss each briefly.

A new version of the DSL may meet the requirements

Languages evolve with time. This results in new language versions that address shortcomings of the previous versions, including lack of functionality and instability. An undesirable effect of language evolution is co-evolution phenomena. In short, this can make existing models and programs of a language unusable because they are no longer compatible with a new language version. In the case of our example, this can potentially imply that earlier models created in the DSL must be discarded or ported to the new language version (if no model transformations exist). It is also quite possible that the needed features never will be implemented at all due to their inappropriate nature or difficulties in merging these features with existing language features. Moreover, software engineers become dependent on new language versions in order to solve their modelling or programming task. Obviously, this dependency is not desirable. Thus, in practice it is not possible to wait for a new language version in order to get a task done.
There might be an alternative DSL that can be used instead
Finding another appropriate DSL can be difficult. It can also be quite time-consuming to learn the language’s limitations which can induce the same problem all over again; the alternative DSL is not compatible with requirements. Moreover, porting of previous modelling / programming work has to be done as well.

The task at hand can potentially be solved using a framework
Using a framework instead is perhaps the best solution to the problem. This is because a framework can be customised to reflect a problem domain more accurately. There are also many frameworks for different problem domains publicly available. However, time and effort have been invested using the initial DSL. Changing development approach implies redoing what has already been done since previous models have to be ported to framework concepts. Most frameworks are written in GPLs. Thus, the advantages of using a DSL are lost.

As can be seen, evolving requirements and changing problem domains can result in problematic issues for software engineers. This includes duplication of work and difficulties in selecting the appropriate technology. Hence, highly dynamic problem domains are likely to induce more of the scenarios discussed here. From the perspective of Language Driven Development, software engineers should be able to create languages as part of the software development process. This suggests the use of a highly usable language development platform.

Naturally, the question arises; is it possible to extend the notion of DSL to provide mechanisms for extensions and customisations? As we will see in the remaining of this thesis, this is possible.

11.2 Composite Domain-Specific Languages
Traditional DSLs are, as mentioned, highly static entities. Consequently, refinement of language properties is virtually impossible. What we need is a DSL that dynamically can be revised according to new requirements. As might be expected, this requires dynamic meta models and semantics. We stress that this must not be confused with statically and dynamically typed / scoped languages.

There are mainly two reasons why a DSL fails to meet evolving requirements and changing problem domains. Firstly, the DSL may not include the necessary constructs in order to model a certain kind of concern. Secondly, the semantics of the DSL is not accurate enough, or lacks support for certain special-case scenarios. Both these issues can be addressed by a composite Domain-Specific Language. A composite Domain-Specific Language can be understood as a set of DSLs that work together as a coherent whole. That is, a set of DSLs has been composed in order to create a new language. As a consequence, a composite DSL addresses several aspects of a composite problem domain. Each aspect is explicitly represented by a self-contained and unique language known as an aspect language. Specifically, aspect languages are used to refine or elaborate a primary language. Traditionally, in aspect orientation an aspect language is used to describe aspects. With regard to composite DSLs, an aspect language contains the actual concepts for modelling of an aspect. Hence, an aspect language is explicitly designed to solve or address a certain kind of concern, and can be used on its own.
Initially, a composite DSL starts out as a traditional DSL. That is, it comprises a set of first-class constructs that are described using a meta model and corresponding semantics. These constructs reflect concepts of a problem domain at the desired abstraction level which concerns are most likely to be expressed by software engineers. In principle, a composite DSL works and acts as a traditional DSL. However, the differences between DSLs and composite DSLs are apparent when focus is put on how language refinement can be performed.

A traditional DSL is based on a static meta model. Changes to this meta model has to be performed by the DSL’s developer. A composite DSL, on the other hand, is based on a dynamic evolving meta model and corresponding semantics that can be changed ad-hoc to meet new requirements. This is possible using language composition, also known as aspect-oriented language weaving. Specifically, one or more aspect languages are incorporated into a composite DSL in order to address certain domain aspects. These aspect languages become integral parts of the composite DSL.

The terms primary language and aspect language are relative and used merely to identify the roles of languages during a refinement process. That is, the language being refined and the language(s) used to perform the refinement, respectively. Clearly, a primary language can have the role as an aspect language in another refinement process, and vice versa. In the context of this thesis, a DSL is either atomic or composite. An atomic DSL has not yet been refined using additional languages. Conversely, a composite DSL is an atomic language that has been refined using weaving. The terms DSL and composite DSL are used interchangeably regardless if a language is atomic or composite. This is because an atomic language is a special-case composite DSL. Moreover, composite and atomic languages can be combined arbitrary. For instance, three composite languages and one atomic language can be used to refine constructs of a fourth composite language. We will use the term traditional DSL in order to refer a proprietary DSL.

It is important to be aware that composite DSL is a technology-independent concept. It can be realised by implementing a foundation for language development and model execution, known as a Language Development and Execution Platform. An LDE Platform provides environments and tools for developing composite DSLs and executing corresponding models. Any kind of OS platform and host programming language can be used to realise such a platform. This includes languages like Assembly, C++, Java, C# and more. A fully-working proof-of-concept LDE Platform has been developed in Java. This platform is used as a reference throughout this thesis. We will elaborate on this platform in a later chapter. Composite DSLs are first and foremost considered to support executable models / programs.

A composite Domain Specific Language is a language based on a dynamic meta model that supports ad-hoc extensions and refinements. This is achieved using an incremental approach denoted aspect-oriented weaving. Specifically, one or more constructs of a primary language is refined using additional aspect languages. Consequently, a composite DSL can be observed as a set of DSL languages that together provide the semantic foundation for the expression of concerns in a target problem domain. A composite DSL may be extended and customised repeatedly to reflect its problem domain more precisely.
11.3 COMPONENTS OF A COMPOSITE DSL

A composite DSL has two main components: a meta model and semantics module.

11.3.1 LANGUAGE META MODEL

Every composite DSL is formally described using a meta model. Meta models provide accurate and convenient ways of describing properties of languages. Specifically, meta models can be used to describe a set of language constructs and their interrelationships. This is known as the abstract syntax of a language. Contrary to traditional languages, a composite DSL meta model is a dynamic entity that evolves more rapidly with time. Specifically, new language constructs and structures can be added or altered as it seems appropriate.

There are no restrictions to what kind of meta model architecture to use when designing composite DSL meta models. Any meta-meta model that provides the essential class and relations concepts can be used. Today, the most common meta model architectures are MOF and Ecore. Both architectures have a similar conceptual basis and provide the necessary meta model concepts. The LDE Platform of this thesis is compatible with Ecore meta models.

An important property of a composite DSL’s meta model is its formal and computable format. This is required in order to achieve dynamic capabilities. That is, several components of the LDE Platform use the meta models of composite DSLs extensively in order to solve their tasks. These components are described in the chapter The LDE Platform.

11.3.2 SEMANTICS MODULE

A composite DSL has a formal semantics which describes the meaning and intended interpretation of every construct of the language. This semantics is implemented using a set of host programming language constructs, such as classes and interfaces, or other types of compilation units available. The implemented and computable semantics is referred to as a language’s semantics module. As noted, the LDE Platform is realised using Java. A semantics module compatible with this platform consists of three compilation unit types. These are classes, interfaces and enums. Recall that in EMF, a language’s implemented semantics is known as the meta model code. Please refer the Car example of chapter Designing a traditional DSL.

A meta model comprises a set of meta-meta model class instances, referred to as language constructs. Each of these constructs is mapped to one interface and one class, constituting a semantics pair. There is a one-to-one correspondence between a construct and a semantics pair. Thus, a meta model consisting of three constructs yields a semantics counterpart totalling three interfaces and three classes. These entities / units are denoted as essential entities since they are directly mapped to constructs found in the meta model. In addition, there might be intuitive or necessary to include auxiliary interfaces and classes in a language’s semantics. These entities are known as additional entities.

Note that enums found in the meta model are not referred to as language constructs. The reason for this is the lack of support for enums in the Ecore meta model architecture, as has been used in our LDE Platform implementation. For instance, it is not possible to associate an enum with other properties than enum literals and annotations. However,
enums can be used in a meta model to provide alternatives, constants and the like.

11.4 ASPECT-ORIENTED LANGUAGE WEAVING

Traditionally, aspect-orientation is used in models on the M1 level of the MOF meta model architecture for defining and encapsulating aspects in distinct model fragments. These aspects can later be woven together with a primary model, yet still ensuring that concerns are treated separately. Here we will use aspect-orientation on meta models of the M2 level. Consequently, aspect-orientation is used as an underlying mechanism of selective language composition.

Aspect-oriented (language) weaving is a method of altering a language by refining (or redefining) one or more of the language’s constructs. This method provides means for surgically extending or customising a language. In its simplest form, aspect-oriented weaving describes how to weave a primary language together with an aspect language. The purpose of this weaving is to refine the properties of one or more constructs in the primary language. Depending on the aspect language used, this includes adding new constructs and structure with corresponding semantics. As we will see, it is also possible to restrict the weaving process to only update a construct’s semantics. In a more general form, aspect-oriented weaving can be used successively to refine several constructs, utilising several aspect languages, in one operation. Note that aspect languages always have to relate to the constructs of the primary language. Thus, the primary language constrains what types of extensions and refinements that are possible. Refer Figure 11.1.

Incorporating an aspect language into a primary language results in weaving of the two languages’ meta models. Semantics is not woven together. Instead, it is linked together using explicit interfaces. Please refer Figure 11.2 and Figure 11.3.

Figure 11.1 Aspect-oriented weaving of languages A and B

Figure 11.2 Weaving of meta models and integration of semantics
11.5 COMPOSITE DOMAINS

Each DSL has a unique problem domain in which it is expressive. This problem domain is reflected at a specific abstraction level. In fact, during a refinement process, it is likely that an aspect language reflects concepts on a lower abstraction level than the primary language. As might be expected, weaving of languages yields a composite DSL whose supported problem domain is composite as well. For comparison, let us consider how a GPL supports different problem domains. Problem domain concepts are reflected in a GPL program/model by combining the available general language constructs. In principle, this means that concepts from numerous problem domains can be mixed together (even though this is not a wise approach). Clearly, GPLs support dynamic problem domains. Composite DSLs, on the other hand, support partially dynamic problem domains. Specifically, every refinement process is defined in terms of one primary language and an arbitrary number of aspect languages. As noted, each aspect language has to relate to one or more constructs of the primary language. Introducing arbitrary concepts of arbitrary problem domains will not make sense since these concepts cannot be used by the primary language. The primary language constrains what kind of refinements that are successful. Consequently, a composite DSL does only support partially dynamic problem domains. Or seen from another perspective, the purpose of an aspect language should be clear from the context of the primary language. It is important to be aware of this since, by definition, a DSL should only support a specific problem domain.

An interesting observation regarding composite language design is that generality increases when the abstraction level decreases. This may at first seem a bit counterintuitive, but it makes sense when considering the problem domain concepts that are reflected in the aspect languages. For the most part, an aspect language is incorporated into a primary language with the purpose of adding more detail and expressiveness to the primary language. Thus, the aspect language contains the necessary building blocks to model a concern of the primary language more closely. To encapsulate such detail, a lower abstraction level has to be chosen by the developer of the aspect language. In fact, decreasing the abstraction level will increase the likelihood of more general concepts. As might be expected, these general concepts can be common building blocks for other higher abstraction level concerns. Clearly, a concept’s building blocks are more general than the concept itself. Thus, the problem domains of aspect languages are moving further away from the initial problem domain of the composite DSLs as more detail is required. This results in a more general composite DSL that can be used to model a problem domain more accurately. However, the overall type of concerns that can be expressed by the composite DSL is the same. This latter observation must, however, not be confused
with how generality and overview traditionally are achieved by increasing the abstraction level.

11.6 **REFINEMENT OF A COMPOSITE DSL**

There are primarily two reasons to refine or elaborate a composite DSL. Firstly, the DSL may not contain the necessary constructs in order to model a certain concern. Secondly, the semantics of one or more constructs in the language may not be appropriate for a given task. Naturally, these limitations have to be considered in the light of a set of requirements. We will elaborate on how these limitations can be addressed.

11.6.1 **REFINEMENT OF ABSTRACT SYNTAX**

There are several issues that require refinement of a DSL’s abstract syntax. An overview of the most common scenarios is found in Table 11.1.

**TABLE 11.1 - DSL DOES NOT MEET REQUIREMENTS - ABSTRACT SYNTAX**

<table>
<thead>
<tr>
<th>Issue:</th>
<th>Refinement:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>The DSL does not contain representation of a concept found in the problem domain</strong></td>
<td>Extension of the DSL using an aspect language containing one or more constructs reflecting the desired concept.</td>
</tr>
<tr>
<td><strong>A construct of the DSL does not reflect a concept of the problem domain in an appropriate manner</strong></td>
<td>Substitution of the construct using an aspect language that reflects the concept more closely. The aspect language can consist of one or more constructs.</td>
</tr>
<tr>
<td><strong>A group of constructs of the DSL provides a faulty representation of a complex concept of the problem domain (cross-cutting concern)</strong></td>
<td>Substitution of the existing constructs with a new set of constructs that represent the complex concept accurately. Several aspect languages may be used to achieve a new definition.</td>
</tr>
</tbody>
</table>

**The DSL does not contain representation of a concept found in the problem domain**

A concept of the problem domain may be a standalone concept or have relations to other concepts.

**Standalone concept**

A standalone concept can be added to the DSL using an aspect language that contains a construct for this actual concept. In practice, this scenario implies that an arbitrary aspect language can be woven together with the primary language. This is because there are no relations to the standalone construct being added to the primary language. Clearly, this gives a high degree of flexibility since models conforming to the DSL can include instances of auxiliary language concepts that do not comply with the problem domain of the DSL. However, there are not many practical situations where such functionality is required. In addition, a custom built semantics is necessary in order to access the objects of the auxiliary concepts. This is because, by default, the arbitrary aspect language is not compatible with the primary language. The LDE Platform supports incorporating arbitrary languages to provide a set of auxiliary concepts. At present, there is not decided if this feature will have a significant value and, thus, should be available. Nevertheless, this option can easily be turned off by demanding that each aspect language used in a refinement process explicitly elaborates at least one construct of the existing DSL. In the following, we will treat each aspect language as a refinement of existing constructs of the DSL.

**Concept with relations to other concepts**

There may be scenarios where an interwoven concept of the problem domain has not been reflected in the DSL. This concept can be incorporated into the DSL. Clearly, it is
necessary to update each of the surrounding constructs in the DSL that is going to relate to the new interwoven construct. Thus, potentially several constructs have to be refined simultaneously. An alternative approach is to restrict the refinement process to the most common construct. This construct represents a common root for all the constructs that needs to be associated with the interwoven concept. This common construct can be refined using an aspect language. The purpose of this aspect language is to add the new interwoven construct to the DSL and simultaneously provide new definitions for the surrounding constructs. In practice, incorporating an interwoven concept is a rare scenario. This is because the interwoven concept should have been identified during the problem domain analysis phase of the language development process. On that note, a problem domain may potentially evolve in a manner that requires incorporation of an interwoven or tangled concept. We have not experienced such a scenario so far.

A construct of the DSL does not reflect a concept of the problem domain in an appropriate manner

For the most part, updates to a composite DSL are performed by refining existing constructs of the language. These constructs are either not accurate enough or do not provide the expressiveness necessary to model their reflected concepts in sufficient detail. Refinement of this kind is straightforward and easy to perform, regardless of the complexity of the construct being refined or how many relations it has to other constructs. If a construct is not accurate enough, it can be refined using an aspect language. As a consequence, new constructs and structures are added to the language.

A group of constructs of the DSL provides a faulty representation of a complex concept of the problem domain

In practice, the only reason why this scenario could happen is because the problem domain has changed. This is a consequence of software evolution, as elaborated in the chapter Software evolution. Replacing a set of constructs can be achieved using one or more aspect languages. These aspect languages contain updated versions of the constructs which represent the complex concept / cross-cutting concern in a more appropriate manner.

Scenarios and approaches discussed in this section can be generalised to an arbitrary number of constructs.

11.6.2 REFINEMENT OF SEMANTICS

There are essentially four reasons / scenarios why the semantics of constructs should be updated. These are listed in Table 11.2.

<table>
<thead>
<tr>
<th>Issue:</th>
<th>Refinement:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semantics is inefficient or inaccurate</strong></td>
<td>Refactoring of the current construct’s semantics.</td>
</tr>
<tr>
<td><strong>Semantics is not up-to-date</strong></td>
<td>Refactoring of the current construct’s semantics and eventually the semantics of constructs referencing this current construct.</td>
</tr>
<tr>
<td><strong>There is not a one-to-one correspondence between the abstract syntax and semantics for a construct</strong></td>
<td>Introduction of API methods that correspond to the properties of the meta model.</td>
</tr>
<tr>
<td><strong>The semantics of a construct does not provide the necessary API towards other constructs of the DSL</strong></td>
<td>Updating the construct’s semantics with the necessary API methods.</td>
</tr>
</tbody>
</table>
**Semantics is inefficient or inaccurate**
Refactoring semantics to become more efficient or accurate is a straightforward process. In most cases, it is only necessary to refine the semantics of the current construct. This may comprise updating algorithms and general restructuring of code. Refinement of a construct’s semantics is achieved using an aspect language in a similar manner as described for syntactical refinements.

**Semantics is not up-to-date**
This scenario is similar to the above issue. However, it may be necessary to refine the semantics of other constructs as well to be able to accommodate the new semantics of the given construct. Specifically, if the updated semantics for a given construct provides new API methods, like setter and getter methods, other constructs may need to be refined in order to use these methods.

**There are not a one-to-one correspondence between the abstract syntax and semantics for a construct**
This issue typically happens if the abstract syntax of the composite DSL is updated, while these updates are not reflected in the semantics. That is, properties found in the abstract syntax of the meta model are not properly backed up by the semantics. Usually, this issue is solved by introducing diverse API methods that correspond to attributes, references and operations declared in the abstract syntax.

**The semantics of a construct does not provide the necessary API**
Most constructs have semantics which invokes API methods found in other constructs’ semantics. Every construct must provide the appropriate API methods to avoid inconsistencies. Using the Code Generator will always ensure that the abstract syntax and semantics correspond (are synced) and inconsistencies are avoided. Nevertheless, manual refactoring is supported by the LDE Platform. Naturally, attention must be paid by language developers to verify that inconsistencies are not introduced. Fortunately, an inconsistency can easily be resolved by adding the missing API methods.

As identified above, there are several reasons to refine a DSL. Refining a language can be performed in two ways: manually or by incorporating one or more aspect languages. In the context of composite DSLs we will refer to refinement as the process of weaving a primary language with aspect languages. In a figurative sense, an aspect language is plugged into the primary language in order to refine the primary language’s abstract syntax and / or semantics. Specifically, the mechanisms supporting this kind of refinement are constructed in a manner that makes it easy for software engineers to customise their languages when needed.

### 11.7 Refining a Construct

**Construct refinement** is the process of increasing the expressiveness or accuracy of a construct. Clearly, this is the simplest refinement scenario. Each construct of a DSL may be refined using an aspect language. One aspect language can also refine several constructs simultaneously. Essentially, there are three types of construct refinement. These are referred to as *extension, redefinition* and *semantics customisation.*

Construct refinement is performed using aspect-oriented weaving. Specifically, a construct of a primary language is woven together with an equally named construct of an
aspect language, known as *correlative constructs*. Correlative constructs are the set of matching pairs in a refinement process. That is:

\[ C = \{<p,a> \mid p \in P \text{ and } a \in A \text{ and } R(a,p)\}\]

- \(C\) - the set of correlative constructs
- \(P\) - the constructs of the primary language
- \(A\) - the constructs of the aspect language
- \(R(x,y)\) - \(x\) refines \(y\)

In essence, there are three potential outcomes of a construct refinement process:

- The syntax of the construct is updated (including addition of new constructs and relations)
- The semantics of the construct is refined
- Both syntax and semantics are refined

In most cases, the semantics of a construct has to be updated to accommodate syntactic changes.

### 11.7.1 EXTENSION

Extending a construct means to weave this construct with a set of additional constructs. This increases the expressiveness of the DSL. Specifically, a concept / concern previously reflected by one construct is now reflected by more constructs. Thus, more domain-specific details can be captured in the DSL model. A valid extension implies that the aspect language used to refine a construct contains at least two constructs; a construct with the same name (unification criterion) as the construct being refined and an additional construct that constitutes the actual extension. In practice, an extension can have as many new constructs as necessary. That is, an aspect language used in an extension process always has \(c = 1 + n, n > 0\) constructs, where \(n\) refers the number of new additional constructs. Extension is the most common type of composite DSL refinement.

### 11.7.2 REDEFINITION

Redefinition is the process of substituting a construct of a primary language with a new construct as defined in an aspect language. This process can be generalised to include redefinition of several constructs simultaneously using either one or more aspect languages. All constructs of an aspect language are related to the construct or constructs that are to be redefined, either directly or indirectly. Note that the total number of constructs found in the DSL meta model may be decremented as a consequence of redefinition. Specifically, there may be redefinition scenarios where fewer constructs suffice to represent a concept or concern. Notice that redefinition is not supported by the current LDE Platform version.

### 11.7.3 SEMANTICS CUSTOMISATION

Semantics customisation is a special-case refinement scenario that makes it possible to revise the semantics of a construct within a DSL. This is achieved using an aspect language consisting of merely one trivial construct. That is, a meta class which does not declare any attributes, references or operations. In practice, the semantics of the primary language’s construct is updated by weaving it together with the trivial construct of the aspect language. As a result, the updated semantics is used in preference to the old semantics. Semantics customisation works regardless of how interweaved and tangled a construct is within the DSL. This allows to pin-point semantics updates of the DSL.
Semantics customisation can be used for simultaneous refinement of several constructs’ semantics. This is achieved using an aspect language comprising more trivial constructs.

11.8 First Example of a Composite DSL

We will exemplify the creation of a composite DSL by elaborating our traditional Car DSL. This was described in chapter Designing a traditional DSL. Specifically, we will see how to refine the Engine construct of the Car language using an aspect language named Engine. The purpose of this refinement is to support modelling of more detailed car engines. This initial description of composite DSL construction is very brief in order to grasp the big picture. It also gives a practical overview of a composite DSL before the LDE Platform is introduced. In a later chapter we will go into more detailed DSL design, and discuss how to structure DSL languages in an aspect-oriented manner.

The previously introduced Car DSL comprises seven meta classes whose instances can be combined to model cars. One of the most comprehensive component in a car is its engine. As can be seen in Figure 10.1, the Engine construct of the Car language can reflect two engine properties, namely horse power and torque. In many situations these may not suffice to model a given engine. Thus, the language has a limitation to what kind of engine that can be modelled.

Traditionally, a user of the Car language encountering such a language limitation has three choices. She / he can either send a request to the language developer to incorporate a new set of language features, find another language that supports more details of the problem domain, or use a framework instead. As discussed, none of these alternatives are optimal. On the contrary, the mechanisms of composite language design let us provide the necessary extension in order to capture more engine-specific details. This is achieved by incorporating a language named Engine into the Car language. That is, Engine is an aspect language used to refine the primary language titled Car. Let us see how this is achieved.

As an initial step, we have ported the traditional Car DSL, as originally defined in EMF, to be compatible with the LDE Platform. This includes a slight modification of the abstract syntax and a restructuring of the semantics. An overview of the new meta model is shown in Figure 11.4. Notice that the only difference from the previous Car version is the introduction of a non-changeable semantics attribute for each meta class. This attribute has a default value which is used to inform the Runtime Environment where to find the construct’s corresponding semantics.
Each construct is mapped to a semantics pair. Semantics pairs constitute the implemented language semantics. Using the Code Generator of the LDE Platform on the meta model (.ecore file) generates all the necessary semantics pairs. These are shown in Figure 11.5.

As can be seen in Figure 11.5, the language semantics comprises three packages. Each interface defines the API of one class. For instance, the IEngine interface declares all methods of the Engine class that should be available to the Runtime Environment and other semantics classes. IEngine and Engine constitute one of the seven semantics pairs of Car. Enums are also generated based on the meta model. In this example though, no enums are used as part of the meta model, thus the enums package is empty. The generated semantics correspond to the semantics for our previous traditional DSL. In other words, semantics is generated for reading data from a Car model. A model conforming to Car v1.0 can be created in a similar fashion as for the traditional DSL. Clearly, at this point our composite DSL has the exact same expressiveness as the previous Car DSL.

The next step is to define the new Engine language. This language will eventually be used as an aspect language in a refinement process of Car v1.0. As for our Car language, problem domain analysis and creation of a meta model initiate the design process.
As can be seen in Figure 11.6, the Engine meta model consists of nine language constructs. These support modelling of engines with a given type, an intake, a set of cylinders and related pistons, and an optional compressor. Obviously, each construct could have featured more properties than what has been identified.

Let us return to the previous scenario where a user of the traditional Car DSL encountered a limitation in the language when modelling the car’s engine. This time, the user is working with a composite DSL. On the contrary to a traditional DSL, a new language can now be incorporated into the composite DSL. This is achievable by copying the Engine language into the Language Repository and using the Weaver of the LDE Platform. This results in a composite language meta model that comprises both the constructs of the Car language and the Engine language. The meta model for the refined Car language can be viewed in Figure 11.7.
An interesting tool of the LDE Platform is the Model Transformer. This can be used to transform a given set of input models to one output model that conforms to a specified meta model. In our example, this means that models created using Car v1.0 do not need to be discarded. In fact, the user could model her/his engine and weave this model together with a Car v1.0 model. More on the Model Transformer and the situations where such transformations are possible is found in the sub section The LDE Platform Tool Suite Model Transformer of chapter The LDE Platform. Note that the semantics for the updated composite DSL is distributed among two previous semantics modules. That is, the interfaces, classes and enums for Car and Engine are still distinct units. More on how the semantics integration is performed can be found in the section Runtime Environment of chapter The LDE Platform.

This first approach towards a composite DSL has been treated in a very basic manner. Later we will see how everything works in detail by creating a composite DSL in several steps using more aspect languages. Firstly, we will elaborate on the LDE Platform and reveal its secrets.
12 The LDE Platform

In this chapter we will explore the LDE Platform. Each component of the platform, including tools, environments and editors, will be covered carefully. However, it is out of the scope of this thesis to go into every detail of the low level code structure and algorithms. Thus, conceptual overviews of components are kept very brief and only the major concepts are included. The interested reader may find all the details by consulting the available source code.

At this point, each component of the platform is described from a technical perspective. In a later chapter, we will return to applications of these components and illustrate how they are used to realise the important aspects of composite language design.

12.1 OVERVIEW

An LDE Platform is a collection of tools and environments that support construction of composite DSLs and execution of corresponding models. This is a technology-independent concept. The accompanying proof-of-concept implementation of the LDE Platform is realised in Java.

There are four components constituting the LDE Platform. These are a collection of tools, known as the LDE Platform Tool Suite - for development and verification of composite DSLs, a Runtime Environment, a Development Environment and a set of File Editors which provide natural environments for creating and editing LDE Platform-specific files. This includes syntax highlighting. In addition, the platform also consists of an XML Parser and an XML Composer which are used by several of the platform components.

12.2 THE LDE PLATFORM TOOL SUITE

Seven tools constitute the LDE Platform Tool Suite. Each tool is specifically designed to support a given task and provides one or more operation modes. Please refer Table 12.1.

<table>
<thead>
<tr>
<th>Name:</th>
<th>Type:</th>
<th>Main purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weaver</td>
<td>Plug-in</td>
<td>Used to weave meta models.</td>
</tr>
<tr>
<td>Code Generator</td>
<td>Plug-in</td>
<td>Generates semantics modules (stubs).</td>
</tr>
<tr>
<td>Extractor</td>
<td>Plug-in</td>
<td>Extracts information from meta models which is used by the Injector. It may also be used to extract types from the meta model directly, and meta information.</td>
</tr>
<tr>
<td>Injector</td>
<td>Plug-in</td>
<td>Uses information from the Extractor to inject semantics types.</td>
</tr>
<tr>
<td>Model Transformer</td>
<td>Plug-in</td>
<td>Transforms a set of input models to one output model.</td>
</tr>
<tr>
<td>ATL Preprocessor</td>
<td>Application</td>
<td>Generates ATL transformations used for a composite model transformation.</td>
</tr>
<tr>
<td>Model Verifier</td>
<td>Plug-in</td>
<td>Verifies if a model conforms to a given meta model and custom rules.</td>
</tr>
</tbody>
</table>

Please note that the acronym MLDSL is used in the proof-of-concept applications. This acronym is an abbreviation for Multi-Layer Domain-Specific Language. This term refers to a composite DSL whose building blocks are again other DSLs. To avoid confusion, this term is not used extensively in the thesis text. However, it appears in screenshots and conceptual diagrams. All source code for the applications is available on the accompanying CD. Please refer Appendix A for details on the contents.
12.2.1 WEAVER

An important operation in construction of composite DSLs is weaving of meta models. Meta model weaving is a complex operation. Specifically, there may exist numerous interrelationships between meta model concepts which must not be compromised in the weaving process. The weaver is implemented as an Eclipse plug-in. A conceptual overview of the Weaver is found in Figure 12.1. Only the most central concepts are included in the figure.

![Figure 12.1 Overview of the Weaver](image)

Brief explanation of operation and figure:

LS mode: The specified Language Specification is parsed by the LS Parser. As a result, the extracted information is processed by the CM Builder which creates one or more composition models. These are used by the Composer to compose the abstract syntaxes of the specified languages. The resulting meta model is written to the file system. Note that all languages specified in the Language Specification have to be installed in the Language Repository (more on this later) prior to weaving.

CM mode: A set of composition models is provided directly to the Composer. That is, no parsing of a Language Specification is performed. The remaining processing is equal as for the LS mode.

The dashed arrows indicate instantiation and control, while the solid line arrows illustrate data transfer. Weaver Meta Info and CM Meta Info are used to describe the weaver operation and composition models. <mm elements> represents concepts used to store an internal computable representation of a meta model.

Two operation modes are supported by the Weaver, known as LS and CM. LS is an abbreviation for Language Specification. CM refers to Composition Models. We will review both modes.

**Weaving initiated from a Language Specification - LS**

Meta model weaving initiated from a Language Specification is by far the most common scenario. A Language Specification is a description of the languages (both atomic and other composite languages) which constitute a composite DSL, structured in an XML compatible file. In other words, this is a formal description of the composite DSL. It can also be viewed as the description of the refinement process necessary in order to create the language. Language Specifications have a file extension of type .mldsl. An example of such file is shown in Table 12.2.
The file content of Table 12.2 specifies a composite DSL named Simulator consisting of four atomic languages. The Language Specification has a multi-node tree structure. This can be observed between the lines 10 till 15. As can be seen on line 10, the first-order (denoting weaving sequence) language of the composite DSL is named Simulator. This language is also known as the primary language of the refinement process. At lines 11 and 12, two constructs of the first-order language, Analyser and Visualiser, are refined using the second-order languages Analyser and Visualiser, respectively. One (second-order) construct of the Visualiser language, named Presentation, is refined using the third-order language Presentation, as shown on line 13. Analyser, Visualiser and Presentation are all aspect languages. Notice that the attribute defines is used to specify that the semantics of the identified construct should be defined by the language. In most cases, this is necessary in order to accommodate the new (or changed) abstract syntax introduced by an aspect language. For instance, according to line 13 the Presentation language should define the semantics of the Presentation construct in the Visualiser language. As can be seen, abstract syntax and semantics are refined in one operation. We will later see how the semantics of several constructs can be defined simultaneously by one aspect language.

The weaving process is initiated from a Language Specification. Refer Figure 12.2.
This results in the Language Specification being parsed and represented in a computable format which is used by the Composition Model Builder (CM Builder). This unit creates a set of composition models based on the meta models that are going to be weaved together. Each meta model weaving (of two meta models) is represented by one composition model. Thus, the number of composition models built to create a given composite DSL is always one less than the total number of languages constituting the composite DSL. All composition models are stored in the file system using the File Writer. Furthermore, the composition models are then used by the Composer which utilises Kompose to execute the compositions. The composition models conform to the Kompose meta model for composition models. An overview of a composition model is found in Table 12.3.

Table 12.3 - An example composition model - CM002-Visualiser-Presentation.kompose

1. <?xml version="1.0" encoding="ASCII"?>
2. <kompose:Composer xmlns:xmi=http://www.omg.org/XMI
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:kompose=http://www.kermeta.org/kompose
   primaryModelURI="platform:/resource/no.uio.ifi.hennb.mlsl.environment.re/mlsl/metamodel/MLDSL.ecore"
   aspectModelURI="platform:/resource/no.uio.ifi.hennb.mlsl.environment.re/lanugages/language/presentation/metamodel/Presentation.ecore"
   composedModelURI="platform:/resource/no.uio.ifi.hennb.mlsl.environment.re/mlsl/metamodel/MLDSL.ecore"
   metamodellName="http://www.eclipse.org/emf/2002/Ecore"/>
3. <predirectivesAM xsi:type="kompose:Set" propertyName="name">
4.  <target xsi:type="kompose:NameRef" qname="presentation"/>
5.  <value xsi:type="kompose:StringLiteral" value="simulator"/>
6. </predirectivesAM>
7. <predirectivesPM xsi:type="kompose:Remove" propertyName="eStructuralFeatures">
8.  <target xsi:type="kompose:NameRef" qname="simulator::Presentation"/>
9.  <value xsi:type="kompose:NameRef" qname="simulator::Presentation::semantics"/>
10. </predirectivesPM>
11. </kompose:Composer>

After all compositions have been processed, the composite meta model is stored in the file system. This meta model conforms to the Ecore meta-meta model.

Weaving initiated from a set of Composition Models - CM

As noted, all composition models are stored in the file system. This is a requirement imposed by Kompose, as it is not compatible with handling a data stream containing the composition models. This, however, allows initiating of weaving based directly on the composition models, and supports partial weaving of composite DSLs. It is possible to utilise a subset of these composition models in order to create different versions of the composite DSL. For instance, the weaving process of our example Simulator DSL utilises three composition models. Using only two of these models creates a partial DSL meta model. Partial weaving may be used as a debugging feature in order to create a meta model in an incremental fashion.

The mechanisms of meta model weaving

As mentioned, the Weaver is based on Kompose. Kompose is a Model Development Kit that supports composition of two meta models, referred to as the primary model and the aspect model. As a result, a composed model is created. A meta model composition is
described using a composition model. Composition models consist of two parts. These are specification of composition properties like namespaces and model URIs, and specification of directives. Please refer Table 12.4.

Kompose works by merging instances of classes found in the Ecore meta-meta model. For instance, EClass instances in the aspect model are merged with EClass instances in the primary model. These instances are unified based on the instances’ names. This is also the criterion for unification of other meta-meta model instances like attributes (EAttribute) and references (EReference). Furthermore, the composition of meta models are governed by an optional set of directives. Each directive translates to a rule that is executed on its target. There are two kinds of composition directives, referred to as pre-merge directive and post-merge directive. Several directives may be used of each type.

**TABLE 12.4 - KOMPOSE COMPOSITION MODEL DIRECTIVES**

<table>
<thead>
<tr>
<th>Type:</th>
<th>Target:</th>
<th>In the context of composite DSLs used to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-merge directive</td>
<td><strong>Primary model</strong></td>
<td>Remove conflicting semantics attribute from the construct that is defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove conflicting namespace attribute from the construct that is defined*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove properties to support redefinition*</td>
</tr>
<tr>
<td></td>
<td><strong>Aspect model</strong></td>
<td>Rename package to be compatible with the primary model’s package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename concepts for adaption to the primary model*</td>
</tr>
<tr>
<td>Post-merge directive</td>
<td><strong>Composed model</strong></td>
<td>(Can be used to remove certain properties)</td>
</tr>
</tbody>
</table>

* Notice that these applications are not used in the current version of the LDE Platform. They address issues necessary in order to support namespaces, redefinition and concept renaming. Please refer the following paragraphs Namespaces and Concept renaming.

Let us consider a basic composition of two simple meta models merely consisting of three classes each. This example may be generalised to more comprehensive meta models. No directives are used at this point. As can be seen in Figure 12.3, we have a primary model consisting of three concepts, X and Y and Z. These concepts are represented using EClass instances. Two of the classes have one or more attributes. In addition, X has a containment reference to Y. Thus, an instance of X is a composite object. Note that Z is included as a standalone class merely to illustrate the mechanisms of Kompose’s composition strategy.

![Figure 12.3 Primary model](image)

The aspect model consists of three classes where two of these have equivalent names of classes found in the primary model, X and Y respectively. These classes also have one attribute. The third class is named W. Refer Figure 12.4.

![Figure 12.4 Aspect model](image)
As can be seen in Figure 12.5, the resulting composed model consists of four classes, where \( X \) and \( Y \) is the sum of weaving the respective equally named classes of the primary and aspect models together. All attributes are preserved within their belonging class, and the relation between \( X \) and \( Y \) is kept. \( Z \) and \( W \) are merely copied to the composed model.

To make things a little bit more interesting, some conflicting properties have been added to the example meta models. As can be seen in Figure 12.6, the primary model consists of three concepts, where \( X \) and \( Y \) have two and one attribute respectively. There are also two relations between these two concepts.

There are two conflicting properties in the aspect model with regard to the primary model, as seen in Figure 12.7. These are the attribute \( x1 \) and the relation \( x \). In the primary model, \( x1 \) of \( X \) has the type \( EInt \). In the aspect model, \( x1 \) has the type \( EChar \). Furthermore, the relation denoted by \( x \) is a non-containment reference in the primary model, while it is a containment reference in the aspect model. In addition, the multiplicities specified for this relation differ.

According to the composition strategy of Kompose, a conflicting attribute is resolved by leaving the attribute without type in the composed model. A conflicting relation is resolved by picking the relation type and multiplicity from the primary model in preference to the aspect model. This is depicted in Figure 12.8.
Attributes found in a class of the primary model are added together with the attributes of an equivalently named class of the aspect model. In the context of composite DSLs, the primary model represents the primary language’s meta model, while the aspect model represents the meta model of an aspect language used to refine the primary language. In most cases, it is desirable to preserve the attributes of a primary model class. Naturally, this is because other constructs in the composite DSL may depend on values from these attributes. There may, however, be scenarios where it is desirable to completely remove all existing attributes from a class in order to provide a complete new definition. This can be solved by executing a set of pre-merge directives on the primary model.

Some focus will be put on specification of a construct’s semantics. As noted, each construct of a language is associated with semantics as defined in the language’s semantics module. A meta class is related to its semantics using an EString attribute named semantics. This attribute has a default value determined by the language developer. It conforms to the following format:

```
language.lname1.classes.cname[:vp]
language.lname2.classes.cname[:vp]
...  
language.lnameN.classes.cname[:vp]
```

- **language** specifies that the construct’s semantics exists within the Language Repository
- **lnameX** refers a language found in the Language Repository with the given name
- **classes** points to the classes in the semantics module of the language identified
- **cname** specifies the actual class that contains the implemented semantics for the construct
- **:vp** provides means of specifying a variation point

Consequently, language semantics for a given construct can easily be changed by editing the semantics attribute’s value. The current version of the LDE Platform supports specification of one semantics class for each construct. Future versions may be able to utilise several semantics classes for one single construct. That is, a construct’s semantics can be distributed among a set of semantics classes. This includes descriptions of framework variation points (if a framework is used to define the language’s semantics).

Recall that refinement of a construct implies a new meta model segment (aspect language) being incorporated into the composite meta model. Specifically, this new segment contains a construct with name equal to the construct being refined. These identical named concepts are woven together as illustrated. However, as both constructs provide a semantics attribute, a conflict occurs. Clearly, the refined version of the construct has an updated semantics that should be used. As a consequence, the semantics attribute of the construct being refined is removed prior to the meta model composition. It is replaced by the semantics attribute from the aspect language’s construct. More on semantics mapping is found in the section *Runtime Environment*.

The Weaver uses Kompose repeatedly to build a composite meta model in an incremental fashion. Kompose supports composition of two meta models, giving one resulting meta model. The Weaver generalises this composition process to support an arbitrary number of meta models. To achieve this, the composition models used to weave a composite DSL are linked together in a chain. By this technique, a composed model from one Kompose composition is used as primary model for the next composition, and so on. This is illustrated in *Figure 12.9*. 
There may be situations where a given aspect language provides a definition for several constructs in a composite DSL. Specifically, this may happen if the aspect addressed by the aspect language cannot be identified and represented using merely one root construct. That is, it may be necessary to refine several constructs of the primary language simultaneously using correlative constructs of the aspect language. This is automatically performed using the Weaver. However, each construct of the primary language whose semantics should be refined must explicitly be identified in the Language Specification. Consider the following examples.

Table 12.5 - Excerpt of Language Specification for refinement of Engine

```
1. <xml version="1.0" encoding="UTF-8"?>
2. <specification>
3.  ...
4.  <language name="Engine">
5.     <language name="Motor"/>
6.  </language>
7. </specification>
```

As can be seen in Table 12.5, Motor is an aspect language intended for refinement of the primary language Engine. Both languages comprise the constructs Cylinder and Compressor. The above Language Specification specifies that all correlative constructs of the two languages should be woven together (not explicitly stated, all constructs with matching names are woven together). However, the semantics as defined in the primary language Engine is still used after the refinement process. The following Language Specification can be used in order to provide new semantics for the refined constructs.

Table 12.6 - Excerpt of Language Specification for refinement of Engine, including semantics

```
1. <xml version="1.0" encoding="UTF-8"?>
2. <specification>
3.  ...
4.  <language name="Engine">
5.     <language name="Motor" defines="Cylinder"/>
6.     <language name="Motor" defines="Compressor"/>
7.  </language>
8. </specification>
```

As can be seen in Table 12.6, lines 4 and 5 ensure that the semantics defined in the aspect language is used for the constructs Cylinder and Compressor.
Some future extensions to the Weaver, *namespaces* and *concept renaming*, have been identified. These features have not been evaluated as critical in order to illustrate the properties of composite DSLs and have, due to time constraints, not been implemented.

**Namespaces**

Different languages can have meta model concepts with equal names. According to the Kompose merging algorithm, names are the primary unification criterion. This unification criterion is used when refining a construct of a composite DSL. Intentionally, the aspect language used to refine a construct of a primary language contains an *EClass* instance (construct) with the same name as the *EClass* instance to be refined. However, if several classes of the aspect model unintentionally share names with classes in the primary model, these classes will also be merged creating a corrupt meta model. This can be solved by prefixing each concept of a language with a given set of characters. However, in the long run, this is not a practical approach. A more efficient approach is the use of namespaces.

A namespace is a mechanism used to differentiate and encapsulate entities. In the context of composite DSLs it is desirable to provide unique namespaces for groups of constructs, corresponding to each language. These namespaces should be preserved during the meta model weaving process. The Ecore meta-meta model specifies one concept that can be used to specify packages / namespaces, namely the *EPackage* concept. However, the underlying weaving technologies used require all constructs of both the primary language and aspect languages to reside within an equally named package. This implies that packages of aspect languages have to be renamed prior to the weaving process (handled automatically by the Weaver). Consequently, there is no flexible mechanism available for definition of construct namespaces. Instead, namespaces are specified using a meta model attribute titled *namespace*. This is a non-changeable attribute of type *EString* that has a default value literal. This value can be an arbitrary set of characters. However, it is suggested to follow traditional namespace conventions. Naturally, giving all classes of a language the same value for the *namespace* attribute implies a common namespace for the classes. (Note that in this case a namespace works on the construct level.)

Namespaces can be realised by the use of composition model pre-merge directives whose task is to rename conflicting concepts of the aspect model based on the value of the *namespace* attribute. These directives can be generated automatically by the Composition Model Builder. Unique concept names can also be achieved by adding some kind of hash string to each concept’s existing name. However, this decreases a language’s readability. Another approach is to selectively choose what kind of meta model concepts to merge. This is currently being worked on by the Kompose crew.

**Concept renaming**

In our Car example, the *Engine* construct was refined using an aspect language named *Engine*. This aspect language comprised a new definition of the *Engine* construct together with a set of new related constructs. Weaving of the Car and *Engine* languages’ meta models worked fine because the root concept of the Engine language was named *Engine*. In other words, there was a name correspondence between a concept of the primary language and a concept of the aspect language. However, the Engine language could easily have had a root construct named *Motor*, instead of *Engine*. Consequently, no reasonable relation could possibly be created automatically between any concept in the primary language and concepts of the Engine language, according to the Kompose main
merging criterion. This kind of issue is solved by renaming the root concept of the aspect language prior to the weaving process using a pre-merge directive. To achieve this, the root concept of an aspect language could be specified in the Language Specification, as shown in Table 12.7. This information could then be used by the Composition Model Builder to create the necessary pre-merge directive.

<table>
<thead>
<tr>
<th>Table 12.7 - Excerpt of a revised Language Specification for Car - root attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2. &lt;specification&gt;</td>
</tr>
<tr>
<td>3.  &lt;language name=&quot;Car&quot;&gt;</td>
</tr>
<tr>
<td>4.   &lt;language name=&quot;Engine&quot; root=&quot;Motor&quot; defines=&quot;Engine&quot;/&gt;</td>
</tr>
<tr>
<td>5.  &lt;/language&gt;</td>
</tr>
<tr>
<td>6. &lt;/specification&gt;</td>
</tr>
</tbody>
</table>

In Table 12.7, Engine is refined using a construct named Motor of an aspect language Engine. Notice that the root attribute of the <language> tag refers to the root concept of the aspect language Engine, while defines indicates that the semantics of the Engine construct of Car should be defined by the Engine language. Another approach could be to define explicit relations between constructs of a primary language and aspect language. An example of this is shown in Table 12.8.

<table>
<thead>
<tr>
<th>Table 12.8 - Excerpt of a revised Language Specification for Engine - relation set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2. &lt;specification&gt;</td>
</tr>
<tr>
<td>3.  &lt;language name=&quot;Engine&quot;&gt;</td>
</tr>
<tr>
<td>4.   &lt;language name=&quot;Motor&quot;&gt;</td>
</tr>
<tr>
<td>5.     &lt;relation source=&quot;Cylinder&quot; target=&quot;Cylinder&quot;/&gt;</td>
</tr>
<tr>
<td>6.     &lt;relation source=&quot;Turbo&quot; target=&quot;Compressor&quot;/&gt;</td>
</tr>
<tr>
<td>7.  &lt;/language&gt;</td>
</tr>
<tr>
<td>8. &lt;/specification&gt;</td>
</tr>
</tbody>
</table>

Engine contains the two constructs Cylinder and Compressor. Here, Motor is used to refine these constructs. The correlative constructs in Motor are named Cylinder and Turbo, respectively. This is specified at lines 5 and 6. Thus, based on this information the Composition Model Builder can encode renaming of the aspect language concepts in the composition model using pre-merge directives. Note that a manual renaming of concepts is used in this thesis.

Notice that the LDE Platform is not dependent on Kompose in order to work. In fact, there is a loose coupling between the LDE Platform and abstract syntax weaving as currently performed by Kompose. This implies that any model weaving tool can be used. The only requirement is that the resulting composite meta model conforms to the Ecore meta-meta model.

12.2.2 CODE GENERATOR

A language consists of the language meta model and semantics module. Moreover, each meta model construct of an language is mapped to a semantics pair comprising an interface and class. It is possible to write the semantics module by hand. However, it soon becomes a tedious task, especially if the meta model is large. Therefore, a Code Generator
has been designed which generates the semantics module based on the meta model definitions. The Code Generator is implemented as an Eclipse plug-in. A conceptual overview of the Code Generator is found in Figure 12.10.

![Figure 12.10 Overview of the Code Generator](image)

**Brief explanation of operation and figure:**

The specified meta model is parsed by MM Parser which creates a computable version of the meta model. This is sent to the Stubs Generator, and a set of Java files is created. These are written to the file system using the File Writer. `<mm elements>` represents concepts used to store an internal computable representation of the meta model. `<code elements>` represents concepts used to build the internal representations of the Java classes and interfaces.

Code generation is the process of generating a semantics module comprising interfaces, classes and enums from a language's meta model. These entities are written to the file system in their respective packages. An overview of the written files for the Car v1.0 language is found in Figure 11.4. Excerpts of `ICar` and `Car` are shown in Table 12.9 and Table 12.10. All generated files have a similar structure.

**Table 12.9 - Excerpt of ICar**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>public interface ICar extends IExecutable</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>public void setEngine( IEngine engine );</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>public IEngine getEngine();</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>public void setWheel( IWheel wheel );</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>public Iterable&lt;IWheel&gt; getWheels();</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>...</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

**Table 12.10 - Excerpt of Car**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>public class Car implements ICar</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>private IEngine engine;</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>private Vector&lt;IWheel&gt; wheels;</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>...</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>this.wheels.add( wheel );</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>}</code></td>
</tr>
<tr>
<td>9.</td>
<td><code>public Iterable&lt;IWheel&gt; getWheels()</code></td>
</tr>
<tr>
<td>10.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>11.</td>
<td><code>return wheels;</code></td>
</tr>
<tr>
<td>12.</td>
<td><code>}</code></td>
</tr>
<tr>
<td>13.</td>
<td><code>...</code></td>
</tr>
<tr>
<td>14.</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
The Code Generator is compatible with all Ecore meta-meta model concepts including inheritance, operations and enums. It is also partially compliant with annotations. Notice that primitive data types are not used for parameters and return types in the API methods of interfaces and classes, as found in a semantics module. Instead, the corresponding wrapper classes are used. The reason for this is how the reflective engine of the Runtime Environment utilises proxies to dynamically allocate types for parameters and return values. Primitive types can be used for local variables and instance variables. More on this is found in the section Runtime Environment.

12.2.3 EXTRACTOR
An important aspect of composite DSL design is how languages may be developed by different people on different locations. As before, this is achieved using aspect-oriented weaving of two languages based on a set of correlative constructs and corresponding API methods. However, in order to incorporate an aspect language into a primary language, all existing semantics relations of refined constructs must be preserved by the new “overridden” semantics.

Recall that the first version of the composite Car language had an Engine construct with two engine-specific attributes: hp and torque. Refer Figure 11.4. Consequently, the semantics interface and class for this Engine construct comprise two setter and two getter methods (for the attributes), together with additional methods and code. An aspect language used to refine the Engine construct must incorporate these API methods as part of its semantics in order to be compatible with the primary language. Naturally, this is because other constructs of the primary language may invoke these methods to acquire the horse power and torque for the engine. In short, this is achieved by exporting the essential semantics of the composite DSL which is then used by the aspect language developer in a selective manner. The developer only incorporates the necessary API methods in order to ensure compatibility with the primary language. A thorough description and illustration of these issues are found in a later chapter. For now, it is sufficient to know that API methods must be exported in order to create compatible languages. This is where the Extractor comes at play. It is used to extract the Type Specification of a given language, composite or atomic.

![Figure 12.11 Overview of the Extractor](image)

**Figure 12.11 Overview of the Extractor**

Brief explanation of operation and figure:

MM Parser builds a computable, internal meta model representation. This is used by Type Extractor to extract the relevant information. The output is either a:

- Type Specification
- Set of Java interfaces that correspond to the types of the meta model
- Meta information for the specified meta model

Target files are written to the file system using the XML Composer, or File Writer directly. This depends
on the file type. `<mm elements>` represents concepts used to store an internal computable representation of the meta model. `<code elements>` represents concepts used to build the internal representations of the Java interfaces.

**Type extraction**

*Type extraction* is one of the Extractor modes. It shares some properties with the Code Generator. Using the Extractor in this mode creates the interfaces of a given language’s semantics module (based on the meta model). One or more of these interfaces can then be incorporated by an aspect language developer. This includes using Java subtyping or copying the interface methods by hand. Note that the extracted interfaces are identical with the default interfaces of the language’s semantics module (Java-specific).

**Type Specification extraction**

Composite DSL design is a technology-independent concept. Consequently, an alternative Extractor mode exists. This mode is known as *Type Specification extraction*. It results in the creation of a `.types` file which specifies a language’s interfaces in a generic way. Thus, language interfaces can be distributed regardless of host programming language used to realise the LDE Platform. An extract of the Car language’s *Type Specification* is found in Table 12.1.

<table>
<thead>
<tr>
<th>Table 12.11 - Extract of a Language Type Specification - car_1.0.types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</code></td>
</tr>
<tr>
<td>2. <code>&lt;specification&gt;</code></td>
</tr>
<tr>
<td>3. <code>&lt;meta&gt;</code></td>
</tr>
<tr>
<td>4. <code>&lt;name&gt;</code>car<code>&lt;/name&gt;</code></td>
</tr>
<tr>
<td>5. <code>&lt;/meta&gt;</code></td>
</tr>
<tr>
<td>6. <code>&lt;type name=&quot;IEngine&quot;&gt;</code></td>
</tr>
<tr>
<td>7. <code>&lt;method name=&quot;setHp&quot; type=&quot;void&quot;&gt;</code></td>
</tr>
<tr>
<td>8. <code>&lt;param name=&quot;hp&quot; type=&quot;Double&quot;/&gt;</code></td>
</tr>
<tr>
<td>9. <code>&lt;/method&gt;</code></td>
</tr>
<tr>
<td>10. <code>&lt;method name=&quot;getHp&quot; type=&quot;Double&quot;/&gt;</code></td>
</tr>
<tr>
<td>11. <code>&lt;method name=&quot;setTorque&quot; type=&quot;void&quot;&gt;</code></td>
</tr>
<tr>
<td>12. <code>&lt;param name=&quot;torque&quot; type=&quot;Double&quot;/&gt;</code></td>
</tr>
<tr>
<td>13. <code>&lt;/method&gt;</code></td>
</tr>
<tr>
<td>14. <code>&lt;method name=&quot;getTorque&quot; type=&quot;Double&quot;/&gt;</code></td>
</tr>
<tr>
<td>15. <code>&lt;/type&gt;</code></td>
</tr>
<tr>
<td>16. <code>&lt;/specification&gt;</code></td>
</tr>
</tbody>
</table>

Each of the language construct’s semantics interface is specified as a *type* with a set of methods. Moreover, each method has a *type* and an optional set of *params*. A *type* always refers to another *type* within the Type Specification, or one of the LDE Platform standard types. At present, these standard types have the same naming as the Java wrapper classes for primitive types. Notice the API methods for the `IEngine` type.

**Meta information extraction**

In addition to the previous modes, the Extractor can be used to extract meta information about a meta model. An excerpt of meta information for the Car language’s meta model is found in Table 12.12. This meta information can be used to easily acquire a complete overview of the meta model. All information regarding the meta model is shown simultaneously in a clear manner. Thus, it is not necessary to access a property page for each element of the meta model as would be the case using the Ecore Model Editor or Ecore Diagram.
### Table 12.12 - Meta Information - car_1.0.minfo

1. Ecore meta model
3. Number of classes: 7
4. [ Class name: 'Car'
5. [ Structural Feature name: 'engine'
6. [ Type: reference
7. Data type: Engine
8. Default value: null
9. Changeable: true
10. Lower bound: 1
11. Upper bound: 1
12. Containment: true
13. ] ...
14. ] ...
15. ] ...

Be aware that additionally added interfaces to a semantics module will not be recognised by the Extractor. This is because the Extractor uses information exclusively from the language’s meta model.

#### 12.2.4 INJECTOR

As noted, the Extractor can be used to produce a Type Specification written to the file system as a .types file. This specification describes the semantics interfaces of a language’s constructs. As might be expected, this specification can be read by the Injector resulting in regeneration of the interfaces. The Injector of the LDE Platform creates Java interfaces. These interfaces are equal to the interfaces of the source language from which the specification was extracted. In a sense, they are injected into a new language.

![Figure 12.12 Overview of the Injector](image)

**Figure 12.12 Overview of the Injector**

Brief explanation of operation and figure:

Input to the Injector is a Type Specification. This is parsed by the TS Parser. The extracted information is used to create internal representations of Java interfaces. Finally, the interfaces are written to the file system using Type Injector and File Writer. `<code elements>` represents concepts used to build the internal representations of the Java interfaces.

Notice that a language developer only needs to incorporate a subset of the interfaces / semantics types described in a Type Specification. Moreover, the file content of a specification can be manually edited to only include a subset of the types. A target package for the injected interfaces can be specified in the .types file, as illustrated in Table 12.13. Thus, the interfaces can easily be added to a new language being created.
Table 12.13 - Language Type Specification - meta information

1. <specification>
2. <meta>
3. <name>car</name>
4. <target>
5. <package>no.uio.hennb.mldsl.car.sl.window</package>
6. </target>
7. </meta>
8. </specification>

12.2.5 MODEL TRANSFORMER

There may be situations where it is desirable to transform a set of input models to one output model conforming to a given meta model. This can be achieved using the Model Transformer.

Figure 12.13 Overview of the Model Transformer

Brief explanation of operation and figure:

The input models are parsed by the TM Parser, while the output meta model (target meta model) is parsed by MM Parser. Information from both parsers is provided to the MT Engine which creates an output model conforming to the output meta model. Subsequently, the resulting output model is written to the file system using File Writer. MM Map and Element Factory are used during the transformation process to represent the meta model and model elements. <mm elements> represents concepts used to store an internal computable representation of the meta model.

As can be seen in Figure 12.13, the Model Transformer takes an arbitrary number of .xmi files and one .ecore file as input. The meta model is parsed and represented in a computable format. Then, all models are analysed and a Virtual Model is created. This entity represents the output model before the model elements are woven together. Thus, the next step is to weave / merge the correlative model elements together. This is performed in the Model Transformer Engine (MT Engine). All merged elements are then stored in the Element Factory. This factory works as a repository containing all possible elements of the output model. An important property of the Model Transformer is its meta model-driven nature. That is, the output model is created by traversing the meta model and populating the output model with (merged) instances from the input models conforming to this meta model. In other words, model instances are mapped to their corresponding meta model classes. This resulting structure is referred to as the Meta Model Map (MM Map) and closely resembles the output model. The Meta Model Map is used when writing the output model to the file system. Note that only entities of the Element Factory conforming to the meta model will be used. Clearly, a non-related input...
model can be specified which results in elements that do not conform to the meta model. These elements will not be included in the output model. The non-related model is simply ignored. An example will illustrate how the Model Transformer works. Three meta models and three models are identified in Figures 12.14-19. Each model is instantiated from its respective meta model. The Reflective Ecore Model Editor has been used to visualise the models in order to gain a more compact view. Each model is populated with test data.

![Figure 12.14 Meta model A, also known as DSL v1.0](image1)

![Figure 12.15 Model a conforming to meta model A](image2)

<table>
<thead>
<tr>
<th>Meta class name:</th>
<th>Attribute name:</th>
<th>Attribute value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>y</td>
<td>1.5</td>
</tr>
<tr>
<td>A</td>
<td>z</td>
<td>text from A</td>
</tr>
<tr>
<td>B</td>
<td>y</td>
<td>2.6</td>
</tr>
<tr>
<td>B</td>
<td>z</td>
<td>text from B</td>
</tr>
<tr>
<td>C</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>A2</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>A3</td>
<td>z</td>
<td>text from A3 #1</td>
</tr>
<tr>
<td>A3</td>
<td>z</td>
<td>text from A3 #2</td>
</tr>
<tr>
<td>A3</td>
<td>z</td>
<td>text from A3 #3</td>
</tr>
</tbody>
</table>
In this example, $A$ is the meta model of a primary language. We will denote it DSL v1.0. Furthermore, $B$ and $C$ are meta models of two aspect languages used to refine two concepts of the $A$ meta model. For the sake of clarity, these concepts are named $B$ and $C$. Using the aspect languages to refine the DSL gives a new meta model, as found in Figure 12.16: Meta model $B$.

![Figure 12.16 Meta model $B$](image1)

Table 12.15 - Attribute values for model $b$

<table>
<thead>
<tr>
<th>Meta class name</th>
<th>Attribute name</th>
<th>Attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$y$</td>
<td>3.7</td>
</tr>
<tr>
<td>$B_1$</td>
<td>$z$</td>
<td>text from $B_1$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$x$</td>
<td>1.4</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$z$</td>
<td>text from $B_3$</td>
</tr>
</tbody>
</table>

![Figure 12.17 Model $b$ conforming to meta model $B$](image2)

![Figure 12.18 Meta model $C$](image3)

Table 12.16 - Attribute values for model $c$

<table>
<thead>
<tr>
<th>Meta class name</th>
<th>Attribute name</th>
<th>Attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>$y$</td>
<td>4.8</td>
</tr>
<tr>
<td>$C_1$</td>
<td>$x$</td>
<td>4</td>
</tr>
</tbody>
</table>

![Figure 12.19 Model $c$ conforming to meta model $C$](image4)

In this example, $A$ is the meta model of a primary language. We will denote it DSL v1.0. Furthermore, $B$ and $C$ are meta models of two aspect languages used to refine two concepts of the $A$ meta model. For the sake of clarity, these concepts are named $B$ and $C$. Using the aspect languages to refine the DSL gives a new meta model, as found in Figure 12.16: Meta model $B$. 

![Figure 12.16 Meta model $B$](image5)
12.20. It will be denoted DSL v2.0.

Clearly, this new meta model implies a new language version. As might be expected, models made using an earlier version of the language will not be compatible with this new meta model. Such backwards compatibility is possible to preserve. However, it requires a special effort put into the language design. This is not always feasible or desirable. There are, however, situations where it is possible to utilise an existing set of models and transform these to conform to the new meta model. Let us see what the Model Transformer brings to the table.

In our example, there are models conforming to each of the meta models presented. From another point of view, there exist a model conforming to the first version of the DSL and two models that express the new concerns as modelled in the additional languages. Weaving these models together would give a new model that reflects all details as expressed by the three existing models. Refer Figure 12.21. The described transformation process is also referred to as recycling of models, or Model Recycling.

Note that the target meta model and models do not need to reside within the same file system folder when using the Model Transformer.

The resulting output model from the Model Transformer is written to the file system. In this example, it is titled a_transformed.xmi. An overview of this model and its attribute values can be seen in Figure 12.22 and Table 12.17.
Recall that the default meta model weaving process (using the Weaver) always elaborates on existing classes of a meta model. Attributes are never removed from classes, since other classes may indirectly depend on these attributes. Clearly, this non-destructive weaving requirement must be fulfilled by the Model Transformer as well. The result of this requirement / property can be observed by inspecting the attribute values of C in the a_transformed model. These are found in Table 12.17. The values for these attributes are acquired from the models a and c. Refer Table 12.14 and Table 12.16.

Another important property of the Model Transformer is correct handling of attribute value conflicts. In the example, such a conflict occurs for the attribute y of the concept B. As can be seen from the preceding figures and tables, the model a and b have different values for this attribute. As might be expected, the value of y is selected from the b model. This is a logical choice, since b represents an updated version of the B concept / concern.

An overview of the model transformation process can be seen in Figure 12.23. As mentioned, the Model Transformer takes an arbitrary set of input models conforming to a set of input meta models, and transforms these to an output model conforming to an
output meta model. All meta models conform to a common meta-meta model, which in the case of the implemented LDE Platform is Ecore.

The model transformation process can be described by substitution and weaving. As can be seen in Table 12.14, the model \( a \) consists of a set of objects. Specifically, it contains one \( B \) and one \( C \) object. Clearly, the models \( b \) and \( c \) also contain \( B \) and \( C \) objects, respectively. Refer Table 12.15 and Table 12.16. A mental conception of the transformation process is to substitute these \( B \) and \( C \) objects of \( a \) with more rich objects from \( b \) and \( c \). However, since the \( B \) and \( C \) concepts of the \( A \) meta model may contain essential properties not reflected by the corresponding concepts in the \( B \) and \( C \) meta models, it is necessary to weave these objects together. Thus, a new object of a given type is created by copying attributes from the corresponding source objects (equally named). This is illustrated in Figure 12.24. Attribute value conflicts are treated as described above. Note that there is not possible to encounter attribute conflicts. That is, a naming conflict where two or more attributes with the same name have different types. This is because the model transformation process is driven from the output meta model. Only input model objects that conform to the output meta model are included in the output model. Naturally, the output meta model must be a valid model where potential attribute conflicts have been resolved in the weaving process. More on attribute conflicts are found in the sub section Weaver.

It is important to be aware that objects conforming to equally named classes of different meta models are unique. These objects have in theory nothing in common and can not be unified based on their formal definition (as found in their respective meta models). They might, however, have properties that can be combined to create a new kind of object as described in a new meta model. On the other hand, it can be easier to grasp the ideas of the model transformation process by considering the equally named objects as instances of
the same meta class. This is not accurate, but will always make sense since the main unification criterion in the meta model weaving process is qualification of names.

Obviously, there are situations where it is not possible to transform a set of input models to a sound and valid output model. Meta models can be altered in ways that compromise the conformity relation with its models. With regard to the transformation process this might induce inconsistencies making it impossible to transform the existing available models to a model conforming to the output meta model. There is, however, a common set of scenarios where the Model Transformer will create a sound output model, as identified by the following general rule:

The Model Transformer will always derive a valid output model from a set of input models if a non-destructive meta model weaving has been used to create the composite output meta model to which the output model conforms.

In other words, all concepts and their interrelationships of the input meta models have to be present in the output meta model and not altered in any way during the weaving process. This requirement relates closely to how new languages are used to extend a DSL. That is, to create a more rich language that reflects additional problem domain properties.

We have earlier mentioned how the Model Transformer could be used in the Car - Engine example. The purpose of the Model Transformer in that example was to be able to recycle a model of an earlier version of the Car language. This prior model could then be woven together with an Engine model to create a model conforming to the new version of Car. Naturally, the Model Transformer is a tool that meets the challenges of software evolution. In a later chapter we will introduce the concept of language views. In short, this concept deals with what perspective to use when creating models in a language. The Weaver and Model Transformer support this concept by providing means of weaving meta model segments and merging models. As can be seen, the Model Transformer has two main purposes: model recycling and selective modelling. More on these subjects are found in the chapter *Software evolution and views*.

The current version of the Model Transformer is at the proof-of-concept stage. Specifically, only containment references are supported. Furthermore, the merging strategy does not behave correctly in the specific situation where an object of type \(x\) is composed of several objects of type \(y\), that is again composed of other objects. A future version of the Model Transformer may address this issue. It could also be beneficial to provide some kind of batch specification of models. Thus, it would be possible to transform an arbitrary number of models in one simple user operation.

Another approach of transforming models is to use a language especially designed for this purpose. In this case, ATL has been used. This is described in the next sub section.

### 12.2.6 ATL PREPROCESSOR

There exist languages specifically designed to support model transformations. Naturally, it is interesting to utilise such a language for an alternative approach of composite DSL model transformations.

ATL is a comprehensive language for describing transformations between a set of input models and a set of output models. Transformations are described in an *ATL Module*. This
entity is used by the *ATL Engine* to create the target models. The *ATL Preprocessor* is a component of the LDE Platform that can be used to create ATL Modules.

An overview of the ATL Preprocessor is found in Figure 12.25.

![Overview of the ATL Preprocessor](image)

Figure 12.25 Overview of the ATL Preprocessor

Brief explanation of operation and figure:

Input and output meta models are parsed by MM Parser. The extracted information is used by the ATL Preprocessor to build an ATL Module. This is written to the file system using File Writer. `<mm elements>` represents concepts used to store internal computable representations of the meta models.

All meta models of a given transformation must be specified in an ATL Module. This implies the meta models of input models and meta models of output models. These meta models are used by the ATL Preprocessor to build an ATL Module consisting of *ATL Matched Rules*. A Matched Rule is one of the rule types supported by ATL. It has the format as shown in Table 12.18.

### Table 12.18 - ATL Matched Rule format definition

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>rule</code> rule_name</td>
</tr>
<tr>
<td>2.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>from</code></td>
</tr>
<tr>
<td>4.</td>
<td>in_var1 : in_type1,</td>
</tr>
<tr>
<td>5.</td>
<td>in_var2 : in_type2,</td>
</tr>
<tr>
<td>6.</td>
<td>...</td>
</tr>
<tr>
<td>7.</td>
<td>in_var_n : in_type_n [(condition)]?</td>
</tr>
<tr>
<td>8.</td>
<td><code>[using</code></td>
</tr>
<tr>
<td>9.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>10.</td>
<td>var1 : var_type1 = init_exp1;</td>
</tr>
<tr>
<td>11.</td>
<td>...</td>
</tr>
<tr>
<td>12.</td>
<td>var_n : var_type_n = init_exp_n;</td>
</tr>
<tr>
<td>13.</td>
<td><code>}</code></td>
</tr>
<tr>
<td>14.</td>
<td><code>)</code></td>
</tr>
<tr>
<td>15.</td>
<td><code>to</code></td>
</tr>
<tr>
<td>16.</td>
<td>out_var1 : out_type1,</td>
</tr>
<tr>
<td>17.</td>
<td>(bindings1),</td>
</tr>
<tr>
<td>18.</td>
<td>out_var2 : distinct out_type2 foreach (e in collection)</td>
</tr>
<tr>
<td>19.</td>
<td>(bindings2),</td>
</tr>
<tr>
<td>20.</td>
<td>...</td>
</tr>
<tr>
<td>21.</td>
<td>out_var_n : out_type_n</td>
</tr>
<tr>
<td>22.</td>
<td>(bindings_n)</td>
</tr>
<tr>
<td>23.</td>
<td><code>[do</code></td>
</tr>
<tr>
<td>24.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>25.</td>
<td>statements</td>
</tr>
<tr>
<td>26.</td>
<td><code>}</code></td>
</tr>
<tr>
<td>27.</td>
<td><code>)</code></td>
</tr>
<tr>
<td>28.</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
Note that the brackets enclose optional content, like the do section at the bottom of the definition. As the name suggests, ATL Matched Rules use matching to select the element of a source model that is used to initialise a target model element. Source elements, are specified in the source pattern section of the rule, as identified by the from keyword, lines 3 till 7. All model elements that conform to a specified type will be matched. Moreover, it is possible to select a subset of the matched model elements by providing a boolean condition. Examples of model elements are instances of meta model classes and enums.

Target patterns, as found between lines 15 and 22, are used to create model elements in the target model. The target pattern section is identified by the to keyword. There are two types of target patterns: simple and iterative. A simple target pattern uses a set of bindings to initialise a target model element feature with a value from a source model element feature. A feature in this context is either an attribute or a reference. The simple target pattern type is of most interest with regard to the ATL Preprocessor.

In addition, local variables can be specified in the local variables section identified by the keyword using, as seen between lines 8 and 14. Furthermore, an ATL Module may also include a set of imperative statements that are executed after the target model elements have been generated. This section resides at lines 23 till 27. The imperative block is identified with the keyword do.

We will return to our scenario comprising the meta models A, B, C and the composite ABC meta model. Refer Figures 12.14-22. As in our previous example, we would like to transform the models a, b and c to a model conforming to the ABC meta model. This time though, ATL is being used. In order to use ATL for transformation of the models, an ATL Module must be created. Traditionally, this is done by hand. However, this is a tedious and error-prone task. Instead, the ATL Preprocessor can be used to automatically create the ATL Module based on the meta models of the intended transformation process. Running the ATL Preprocessor with meta models A, B, C and ABC as input results in an ATL Module consisting of eight rules. These rules are stored in a file with the file extension .atl. In this case, this file is named T000.atl. An excerpt of this file is shown in Table 12.19.

<table>
<thead>
<tr>
<th>Table 12.19 - Excerpt of generated ATL Module for meta models A, B, C and ABC - T000.atl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. module</td>
</tr>
<tr>
<td>2. create</td>
</tr>
<tr>
<td>3. rule r3</td>
</tr>
<tr>
<td>4.</td>
</tr>
<tr>
<td>5. from</td>
</tr>
<tr>
<td>6. v0 : a!B, v1 : b!B</td>
</tr>
<tr>
<td>7. to</td>
</tr>
<tr>
<td>8. t0 : abc!B</td>
</tr>
<tr>
<td>9.</td>
</tr>
<tr>
<td>10. z &lt;&lt; v0.z,</td>
</tr>
<tr>
<td>11. c &lt;&lt; v0.c,</td>
</tr>
<tr>
<td>12. a3s &lt;&lt; v0.a3s,</td>
</tr>
<tr>
<td>13. b1 &lt;&lt; v1.b1,</td>
</tr>
<tr>
<td>14. b2 &lt;&lt; v1.b2,</td>
</tr>
<tr>
<td>15. y &lt;&lt; v1.y</td>
</tr>
<tr>
<td>16. )</td>
</tr>
<tr>
<td>17. }</td>
</tr>
<tr>
<td>18. }</td>
</tr>
<tr>
<td>19. ...</td>
</tr>
</tbody>
</table>
Lines 1 and 2 constitute the ATL Module header. This header has to be filled in to match the Eclipse Run Configuration used. This configuration specifies the meta models and their corresponding models. In addition, it can be used to specify model handlers and other transformation parameters. An overview of a completed header can be seen in Table 12.20.

<table>
<thead>
<tr>
<th>Table 12.20 - Example of ATL Matched Rule header declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. module T000;</td>
</tr>
<tr>
<td>2. create OUT_ABC : ABC from IN_A : A, IN_B : B, IN_C : C;</td>
</tr>
</tbody>
</table>

Rule r3 will be used to illustrate the mechanisms of how the output model is created, as can be seen at lines 3 till 17 of Table 12.19. As can be seen at line 6, the matching source elements of this rule are the object conforming to the concept named B from the package a (meta model A) and the object instantiated from the concept B from the package b (meta model B). Recall that these elements represent the objects of the class B in the two meta models A and B, where the latter meta model provides an updated definition of the class. As can be seen at line 8, the target element of the output model is an object of a concept B. Specifically, this concept is the class B of meta model ABC whose model object will consist of the combined properties of the objects a.B and b.B. Notice that the object is created at this line. A set of variables are created for the given rule. These are named v0, v1 and t0 and used to refer to the diverse model elements. Moreover, a set of bindings is found between lines 9 and 16. These bindings are used to initialise the features of the target model element with values from the source model elements. For instance, the class B of ABC has an attribute named z. This attribute is initialised with the value from the attribute z in the class B of meta model A. Furthermore, the reference b1 in the meta model ABC is initialised with the value from reference b1 in the meta model B.

As can be seen from the description of the rule r3, an important task of the ATL Preprocessor is to resolve all cross references between meta models. For example, in r3 source elements from two meta models are used to create one target element. Thus, a search strategy is needed in order to find all properties of the target model as a function of the input models. Implementation wise, this implies that several meta models are addressed by the ATL Preprocessor simultaneously.

As might be expected, execution of the pre-processed ATL Module results in the creation of a model equivalent to the output model from the Model Transformer, as described in the previous example. See Figure 12.22 and Table 12.17 for details.

There is one difference using the Model Transformer and ATL worth mentioning: the strategy used to transform models. As pointed out earlier, the Model Transformer uses the output meta model as a reference in the model weaving/merging process. This is the only meta model required by the Model Transformer to perform a valid transformation. ATL, on the other hand, needs to refer all meta models of the transformation process in order to function. That is, meta models of input models and output models.

A future extension to the ATL Preprocessor is to combine it with the ATL Engine to support direct execution of the ATL Module. The intended application can be built as an Eclipse plug-in, making it easy to specify the related meta models and models of a transformation, as illustrated with the Model Transformer. See Figure 12.21.
12.2.7 MODEL VERIFIER
An important aspect of any model is its correctness with regard to its conforming meta model. Obviously, a model needs to be valid in order to be usable. Since this is such an important requirement, a Model Verifier has been developed. This is presented in Figure 12.26.

![Figure 12.26 Overview of the Model Verifier](image)

**Figure 12.26 Overview of the Model Verifier**

Brief explanation of operation and figure:

The information of the specified model and meta model is acquired using the Model Loader and MM Parser respectively. This information is sent to the MV Engine which analyses whether the model conforms to the meta model. Note that custom rules can be verified as well (more on this later). `<mm elements>` represents concepts used to store an internal computable representation of the meta model.

There exists a built-in model verifier in EMF. This model verifier analyses that a model conforms to its specified meta model. However, it is not possible to add rules and heuristics to this model verifier in order to verify additional properties. This is possible using a custom-made model verifier, as is the case with the Model Verifier. By default, the Model Verifier analyses if a model possesses the properties of Table 12.21 with respect to its meta model. (Verification of string attributes can easily be turned on.)

<table>
<thead>
<tr>
<th>TABLE 12.21 - MODEL VERIFIER ANALYSIS PROPERTIES / DEFAULT RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property:</td>
</tr>
<tr>
<td>General model structure is preserved</td>
</tr>
<tr>
<td>Model contains elements</td>
</tr>
<tr>
<td>Root element conforms with root concept of meta model</td>
</tr>
<tr>
<td>String attribute has a non-zero length</td>
</tr>
<tr>
<td>Multiplicity is exact</td>
</tr>
<tr>
<td>Attribute is legal</td>
</tr>
<tr>
<td>Attribute has legal value</td>
</tr>
<tr>
<td>Attribute has value, if mandatory</td>
</tr>
<tr>
<td>Reference is legal</td>
</tr>
<tr>
<td>Reference has legal value</td>
</tr>
<tr>
<td>Reference has value, if mandatory</td>
</tr>
</tbody>
</table>

The result of the analysis determines if a model conforms to its meta model. Notice that the properties listed are differentiated by type / group.
A model can be verified against any given meta model. In essence, there are two outcomes of this action. The model is either a valid instance of the selected meta model, or it is not. In the last case, a dialogue box will give a specific message of what caused the verification process to fail. After resolving this issue, the Model Verifier can be run again. New dialogue boxes will appear until all inconsistencies in the model have been resolved.

An example of a verification error dialogue box is found in Figure 12.27. It contains an indication of the error type and a description of what caused the error to occur. The error types correspond to the default rules of the Model Verifier.

![MLDSL Model Verifier](Figure 12.27 Dialogue box indicating model inconsistency)

An interesting feature of the Model Verifier is the ability to easily verify a model against an arbitrary meta model. In some cases this is quite useful. For instance, a model can in principle conform to several meta models. Or more specifically, to subsets of meta models. (Recall that reuse of languages is supported.) In EMF a model is always checked towards the model’s specified meta model. Thus, the inherent meta model path of a model must be manually edited in order to verify the model against another meta model.

It can be interesting to analyse if additional properties, not governed by a meta model, are fulfilled. Let us return to our previous Car DSL v1.0. Please refer Figure 11.4 for details. A Car model captures a set of properties for a given car. It might be interesting to verify if this car possesses a certain kind of properties. For instance, we might want to know if the car has a horse power of more than 120, four seats and a blue chassis colour. These properties can be verified according to a new set of non-default rules that can be used by the Model Verifier. An overview of these rules is found in Table 12.22.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horsepower more than 120</td>
<td>Custom/Concept</td>
<td>No</td>
</tr>
<tr>
<td>Has four seats</td>
<td>Custom/Relation</td>
<td>No</td>
</tr>
<tr>
<td>Blue chassis colour</td>
<td>Custom/Relation</td>
<td>No</td>
</tr>
</tbody>
</table>

By this technique, it is possible to provide means of querying. In general, such technology can be used to search a large number of models for a specific kind of car. Alternatively, each model can be inspected manually to investigate if the given properties are fulfilled. Clearly, this is a cumbersome method. Notice that the custom rules expressed above do not refer to properties described by a meta model. Consequently, custom rules may be used to verify any kinds of model properties.

In fact, the Model Verifier can have several Verifiers. At present, two such verifiers are implemented. These are ModelVerifier and CustomVerifier. ModelVerifier contains implementations for the default rules listed in Table 12.21, while
CustomVerifier can be used to execute custom rule sets. New rules can be added by subtyping the abstract class Rule and override the verify method, as shown in Table 12.23.

Table 12.23 - The verify(...) method of Rule

```java
public abstract Vector<VerificationError> verify( Element element );
```

Hence, rule code can be expressed in simple terms in the defined verify(...) method of a subclass. Note that the element passed to the method is the model object that the rule targets. Rules are identified by their type and name of the rule’s target entity. For instance, a rule verifying if a car has at least a horse power of 120 is shown in Table 12.24. Naturally, this rule targets the hp attribute of the Engine class. A rule can be used to verify one or more properties of a model object.

Table 12.24 - Custom rule for verification of car horse power

```java
public class CarHp extends Rule {

    public CarHp() {
        super( Rule.CONCEPT, "Engine", "hp" );
    }

    public Vector<VerificationError> verify( Element element ) {
        Attribute attribute = element.getAttribute( "hp" );

        if ( attribute == null ) {
            errors.add( new CustomError( "Engine does not have a value for hp" ) );
            return errors;
        }

        double hp = Double.parseDouble( attribute.getValue() );
        if ( hp < 120 ) {
            errors.add( new CustomError( "Horse power is not above 120" ) );
            return errors;
        }

        return null;
    }
}
```

Lines 9 till 21 comprise the actual rule code. A rule for verification of car seats count can be seen in Table 12.25. seats is a reference of the Car class.
In fact, it is possible to write complex rules that verify a model segment or complete model. This is achieved by passing the root model element to the rule. It is also possible to run several verifiers on a model in one operation. In our example, it could be beneficial to initially run the default rules to assure that a given model conforms to its meta model. Then, custom rules could be executed to verify that the model is of a certain “subtype”.

A rule set can be used to identify a given model. Consequently, it is possible to differentiate between an arbitrary number of models conforming to the same meta model. Another usage is to create rules that verify certain model aspects. These aspects may be addressed by models conforming to different meta models. For instance, it can be critical that all model features of some models are initialised. This can be verified with a set of custom generic rules. Another application is to verify if models address cross-cutting concerns like security, reliability, and more.

Notice that the Model Verifier plug-in is configured to run the default rules as found in ModelVerifier. However, evaluation of custom rules can easily be accommodated. Also, only containment references are supported by the current version of the Model Verifier. From its inception, the Model Verifier was designed to be able to verify if a model conforms to its meta model. Model verification using custom rules is not directly associated with the concept of composite DSLs. However, we decided to include a brief overview of such verification due to its interesting applications regarding language-based software engineering (Language Driven Development).

### 12.3 Runtime environment

Until this point, we have seen how to define and develop languages that are highly dynamic. As might be expected, execution of these languages is not a straightforward process. There are some special concerns that need to be taken care of. Some of these are listed in Table 12.26.
Dynamic meta model
A traditional language has a static meta model. This means there is no way for the software engineer / language user to change the syntactic or semantic properties of the language. As a consequence, it is possible to utilise a static parsing and compilation; a model always has to conform to the current supported meta model of the language. A composite DSL, on the other hand, has a dynamic meta model. In fact, there are no rules to how the meta model of a composite DSL is modified or extended as long as the meta model conforms to its meta-meta model. Specifically, a composite DSL meta model can evolve according to the concept of refinement, or in an arbitrary manner by altering the required properties.

Dynamic semantics
Refining constructs of a composite DSL implies that new semantics is incorporated into the language. This semantics is unique. Furthermore, it is mapped to the abstract syntax of the language. Hence, semantics changes according to the DSL’s meta model. In addition, it is possible to provide an arbitrary semantics for one or more constructs, independently of the abstract syntax. Notice that, in this context we treat semantics as a separate entity even though it may be treated as an integral part of the meta model.

Concepts with several semantics definitions
As noted previously, extension and customisation of a composite DSL implies that one or more constructs of the language are refined using one or more aspect languages. Specifically, there is one or more constructs of every aspect language that are unified with correlative constructs of the primary language. This yields an updated semantics of the DSL. However, there are now two previous semantics modules that include semantics for the unified constructs. Further refinements introduce even more semantics modules with overlapping semantics for unified concepts.

Semantics data types integration issues
We have earlier discussed the components of a semantics module. These are interfaces, classes and enums. Languages can be developed by numerous parties. However, such development occurs on different platforms with unique class paths. Naturally, data types used in other languages’ semantics are not available on these platforms and vice versa. And, according to the Java type system, it is not possible to refer these types remotely.

Distributed semantics
Semantics for a language can be distributed among several semantics sources. Until this point we have only considered composite DSLs with one semantics source. However, it is quite possible that a language’s semantics is segmented in multiple sources. For instance,
a third-party framework can be used as an addition to a language’s semantics module. We will later see how this can be performed.

Clearly, there has to be some kind of dynamic runtime environment to accommodate the dynamic nature of the languages. An illustration of the Runtime Environment can be seen in Figure 12.28.

![Figure 12.28 Overview of the Runtime Environment](image)

**Figure 12.28 Overview of the Runtime Environment**

Brief explanation of operation and figure:

A specified model and its meta model are parsed by one or more instances of the M Parser, and MM Parser respectively. Consequently, a Java object model is created which is executed by the Java JVM. The program control is initiated in the Interpreter where the `process()` method of the root model element is invoked. `<mm elements>` represents concepts used to store an internal computable representation of the meta model.

The most essential component of the Runtime Environment is the **RE Engine / Compiler** and its two sub components: the **MM Parser** and **M Parser**. We will shed some light on these entities.

**Meta Model Parser (MM Parser)**

As the name suggests, the purpose of the MM Parser is to parse the composite DSL meta model and represent it in an internal computable format. A meta model comprises instances of a set of meta-meta concepts (Ecore meta concepts). All these concepts have their corresponding representation in the MM Parser. These are listed in Table 12.27.
INTERNAL REPRESENTATIONS OF META MODEL CONCEPTS

<table>
<thead>
<tr>
<th>Concept:</th>
<th>Ecore meta-meta concept and instances:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaModel</td>
<td>Represents a computable format for an Ecore meta model</td>
</tr>
<tr>
<td>MMAnnotation</td>
<td>eAnnotations</td>
</tr>
<tr>
<td>MMClass</td>
<td>ecore:EClass (eClassifiers)</td>
</tr>
<tr>
<td>MMEnum</td>
<td>ecore:EEnum (eClassifiers)</td>
</tr>
<tr>
<td>MMLiteral</td>
<td>eLiterals</td>
</tr>
<tr>
<td>MMOperation</td>
<td>eOperations</td>
</tr>
<tr>
<td>MMPackage</td>
<td>ecore:EPackage</td>
</tr>
<tr>
<td>MMPParameter</td>
<td>eParameters</td>
</tr>
<tr>
<td>MMStructuralFeature</td>
<td>ecore:EAttribute and ecore:EReference (eStructuralFeatures)</td>
</tr>
<tr>
<td>MMSemantics</td>
<td>LDE Platform semantics specification (semantics module)</td>
</tr>
<tr>
<td>MMTypes</td>
<td>LDE Platform compatible data types</td>
</tr>
</tbody>
</table>

Notice that inheritance and supertypes are covered by the **MMClass** concept. An object of type **MetaModel** is returned by the MM Parser. This object is used by the M Parser. MM Parser is also used by tools of the LDE Platform.

**Model Parser (M Parser)**

M Parser is the core component of the RE Engine. It features comprehensive recursive and reflective algorithms that support the highly dynamic nature of composite DSL models. In fact, two levels of recursion are needed. Firstly, recursion is used within the M Parser algorithms. Secondly, one or more additional instances of M Parser can be created from within the algorithms of the first M Parser in order to process model segments. Subsequently, new M Parser instances can be created within the additional M Parser instances, and so on. This includes a separate **Proxy Repository** for each M Parser instance.

**Proxies and Proxy Repository**

We will address one runtime concern in greater detail, identified above as *Semantics data types integration issues*. One of the key mechanisms that make it possible to support languages created by different developers on different platforms is **dynamic proxies**. The purpose of a proxy is to create a bridge between two semantics types using method forwarding / dispatching. In essence, a proxy instance is a substitute for another target object. This is achieved by dispatching invocations on the proxy instance to the target object using an invocation handler.

A **dynamic proxy** is an instance of a dynamically created (anonymous) class that implements one or more interfaces at runtime. Thus, this class represents a dynamically generated proxy definition. Each interface of a semantics module (except the root class’ interface) is represented by a dynamic proxy definition. Note that we will use the term **proxy** as the instance of a given **proxy definition** (dynamically defined class). We will illustrate this mechanism using a simple example.

Consider the two languages L1 and L2 of Figure 12.29.
Using L2 as an aspect language to refine the C construct of L1 gives the language of Figure 12.30. We will refer to it as L3. Notice the semantics interfaces and classes for C, as both languages define semantics for this construct. Stars are used to differentiate between the two versions. That is, C* is an alias for language.l1.classes.C, and C** points to language.l2.classes.C. We will use these distinctions in the discussion.

As a result of language weaving, semantics for the C construct is now defined by the class C**. Specifically, the value of the semantics attribute of C points to C**. Refer Figure 12.30. As might be expected, IA, IB, A and B of the semantics module are not compatible with IC** and C**. Specifically, A references IC* and C*.
Figure 12.30 Composite language L3 consisting of L1 and L2

Table 12.28 - Overview of IA and A

1. package language.l1.interfaces;
2. import no.uio.ifi.hennb.mldsl.specifications.IExecutable;
3. import language.l1.interfaces.IC;
4. import language.l1.interfaces.IB;
5. public interface IA extends IExecutable{
6. public void setC( IC c );
7. public IC getC();
8. public void setB( IB b );
9. public IB getB();
10. }
11. }
12. package language.l1.classes;
13. import language.l1.interfaces.IA;
14. import language.l1.interfaces.IC;
15. import language.l1.interfaces.IB;
16. public class A implements IA{
17. {
18. private IC c;
19. private IB b;
20. public void setC( IC c ) { this.c = c; }
21. public IC getC() { return c; }
22. public void setB( IB b ) { this.b = b; }
23. public IB getB() { return b; }
24. public void process() { }
25. }
Let us review IA and A. As can be seen on lines 7 and 8 in Table 12.28, IA declares two methods that refer IC*. These methods are implemented by A. Refer lines 20 and 21. IC* declares the API methods of C*. Recall that C* defines the semantics for the construct C in the language L1. However, after the refinement process C* does no longer define the semantics of the C construct in language L3. Specifically, the semantics is now defined by C**. Obviously, the types IC* and IC** is not compatible, neither the types C* and C**. In order to make a bridge between the types, a proxy is defined for IC*. Its purpose is to dispatch all method invocations to an instance of the class C**, instead of C* (C* is bypassed). Consequently, the setC(...) method, as implemented at line 20, is always invoked with a proxy that represents IC*.

Recall that C** must implement all methods from IC* in order for the language L2 to be compatible with L1 (C is the only refined construct). In this example, IC** extends the IC* interface which has been renamed to ICP1. Refer Figure 12.30. Specifically, C** has to implement setC(...) and getB(). See table Table 12.29. These methods reflect the relation between B and C in L1. As a result, C** defines all methods which are potentially dispatched by the proxy for IC*.

A conceptual overview of the proxy for IC* is shown in Figure 12.31. The behaviour of the proxy is implemented by a generic invocation handler. This behaviour is denoted as Dispatching in the figure.

An important property of using dynamic proxies is transparency. Hence, the semantics code of A will function properly regardless of the argument passed in invocation of setC(...). For instance, the code in Table 12.30, will behave correctly regardless if c is a proxy representing C*, C** or any other class that defines the methods setC(...) and getB().

Table 12.29 - Overview of IC*

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>package language.proxy1.interfaces;</td>
</tr>
<tr>
<td>2.</td>
<td>import language.proxy1.interfaces.IB;</td>
</tr>
<tr>
<td>3.</td>
<td>public interface IC</td>
</tr>
<tr>
<td>4.</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>public void setC(IB b);</td>
</tr>
<tr>
<td>6.</td>
<td>public IB getB();</td>
</tr>
<tr>
<td>7.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12.31 Conceptual overview of proxy for IC*
Table 12.30 - Example of semantics code for A

1. public void process()  
2. {  
3.   c.setB( new B() );  
4.   c.getB();  
5. }

Figure 12.32 Example model conforming to L3

A model conforming to L3 is depicted in Figure 12.32. Executing this model initiates construction of five dynamic proxy definitions representing IB, IC*, IC**, ID and IE. A printout dump of these proxy definitions (class names) can be seen in Figure 12.33. Notice that a proxy is not created for the interface of the root class, IA. Moreover, only the necessary proxies for executing a given model are created.

'Proxy0' represents: 'language.proxy1.interfaces.IC'
'Proxy1' represents: 'language.proxy1.interfaces.ID'
'Proxy2' represents: 'language.proxy2.interfaces.ID'
'Proxy3' represents: 'language.proxy2.interfaces.IE'
'Proxy4' represents: 'language.proxy2.interfaces.IC'

Figure 12.33 Overview of dynamic proxy definitions

In total, six proxy instances are used to execute the model. Specifically, each proxy definition is instantiated one time, except for $Proxy1 that is instantiated twice. This is because two instances of the meta model class B are used in the model. Refer Figure 12.32. Also notice that two proxies ($Proxy0 and $Proxy4) must be used to establish communication between C** and the semantics classes A and E. $Proxy0 represents the proxy definition for IC* as elaborated above. Or in other words, it represents the mapping between IC* and C**.

Proxy instances are stored in the Proxy Repository. This is used in the processing of associations. Briefly speaking, the runtime proxy object pointed to by an association may not have been created when the association is encountered in parsing of the model. Thus, associations have to be resolved in the final stage of the model processing when all proxies are available. For instance, a FunctionCall in some kind of language can not associate a Function before the actual function (instance) is created.

Other interesting scenarios appear when several constructs are refined simultaneously, and when proxies are used as argument and return values for other proxies (proxy nesting). As mentioned, a given execution scenario may require instantiation of more M Parsers. Consequently, this will create a Proxy Repository for each M Parser. It is out of
the scope of this thesis to go into further details regarding execution of models. The reader is invited to consult the source code of the M Parser and the generic invocation handler.

Execution of a model is performed in two steps known as the **object model generation phase** and **interpretation phase**. During the object model generation phase a Java object model is constructed to reflect the language model being parsed. This includes generation of proxy definitions and initialisation of semantics classes with attribute values and proxy instances. The object model is then interpreted during the interpretation phase. Specifically, this phase processes the language semantics. Note that the semantics class corresponding to the root concept of the meta model has to implement the interface `IExecutable`. This interface declares a method `process()` which is used by the Runtime Environment to pass program control to the object model.

**Language Repository**

All languages used by the Weaver and Runtime Environment have to be installed in the Language Repository. The Weaver uses the Language Repository to find meta model segments, while the Runtime Environment searches the repository at runtime for appropriate language semantics. An installed language consists of a meta model, the semantics module and optional template / example models. We will later introduce the notion of a remote Language Repository.

There also exists a standalone version of the Runtime Environment that works independently of Eclipse. Consequently, a composite DSL can be used in any JVM compatible environment. For instance, it is possible to use composite DSLs together with web applications running on application and web servers. Furthermore, using the standalone Runtime Environment allows the creation of self-contained bundles consisting of composite languages, frameworks and other third-party software. These bundles constitute a logical whole and can be deployed as one product. The standalone Runtime Environment is later used to create a web application.

Note that there are certain scenarios that are not supported by the current version of the Runtime Environment. These are scheduled to be resolved by an updated version.

### 12.4 Development Environment

At present, the Development Environment consists of some tools and a proposed folder structure. This folder structure is represented using a Java project. Tools directly supporting development of composite DSLs are the Weaver, Code Generator, Extractor and Injector. In addition, several small example languages are included as reference. Further additions to the Development Environment could include Language Development Kits with language templates and meta model fragments. It may also be beneficial to define a new type of Eclipse project, perspectives and wizards. We will show an example of this later.

### 12.5 File Editors

There are three specific file types used in the context of composite DSL development. These have extensions `.mldsl`, `.types` and `.minfo`. Each type has its own Eclipse editor which provides custom syntax highlighting, as illustrated in Table 12.31, Table 12.32 and Table 12.33 respectively. A file can be opened in its corresponding editor by
right-clicking on the file and choosing the editor from the Open With sub menu. Please refer the DTDs and XML Schemas for Language Specifications (.mlds/) and Type Specifications (.types) on the accompanying CD.

Further additions to the editors may include features like file verification (towards DTD), content suggestions and ‘code’ completion. Future work can also include definition of GUI-based editors supporting drag-and-drop functionality. This way, no manual creation and editing of files are needed.

Table 12.31 - Excerpt of an .mlds file viewed in the Language Specification Editor

```xml
1. <language name="Car">
2.   <language name="Engine" defines="Engine"/>
3. </language>
```

Table 12.32 - Excerpt of a .types file viewed in the Type Specification Editor

```xml
1. <type name="ICar">
2.   <method name="setName" type="void">
3.     <param name="name" type="String"/>
4.   </method>
5. </type>
```

Table 12.33 - Excerpt of a .minfo file viewed in the Meta Information Editor

```xml
1. Class name: 'Car'
2. [
3.   Structural Feature name: 'engine'
4.   [
5.     Type: reference
6.     Data type: Engine
7.     Default value: null
8.   ]
9. ]
```

...
13 Designing a composite DSL

We have earlier illustrated briefly the design process of a composite DSL, as exemplified with a language for modelling of cars. In this chapter, focus is put on composite language design and development. Specifically, we will see how a language can be customised to meet new requirements. Tools from the LDE Platform Tool Suite will be used throughout the chapter. Please refer The LDE Platform chapter for details on these tools. Different scenarios will be described to give a contextual background to why a language customisation is performed and how the composite language mechanisms prove their worth.

13.1 A SIMULATOR LANGUAGE

Simulation is an important concept of many scientific disciplines. Well-known usage areas are risk analysis, performance optimisation and meteorology. In order to perform a simulation of a system it must be possible to capture and represent key characteristics and behaviours. Naturally, simulation and modelling are closely related. In fact, the outcome of a simulation is in many cases directly related to the quality of the model used as basis. With this in mind, we will consider an example language for calculation of a weather prediction prognosis. This language will be customised in several steps.

Note that the languages in this chapter are kept as simple as possible. The purpose is to illustrate the mechanisms of composite language design, not the art of weather simulation. Clearly, meteorology is a complex affair based on mathematical analysis in several variables. In our example, such mathematics is replaced with trivial first-order calculations. Important aspects of any simulation are the approximations and assumptions used, and the validity of the outcome. This is of lesser importance here.

The initial version of the Simulator meta model is illustrated in Figure 13.1. Notice that a compact representation of the diagram is used. Attributes are of minor interest at this point.

![Figure 13.1 Meta model for Simulator v1.0](image)

Simulator is a language comprising five constructs. Simulator is the root class of the meta model. Three concerns are captured by the meta model (and associated semantics). These are computation of a statistic variable, capturing of source data and visualisation of the computed result. Specifically, the semantics of the Analyser concept contains an algorithm that computes a decimal value indicating the probability of nice weather. This algorithm uses input data from two sensors, here known with simplified names as Temperature and Pressure. Both sensors provide a decimal value indicating whether their reflected weather property resides within a pre-determined range. An assumption is that the temperature should be within the range of 20 to 30 degrees Celsius and the pressure above 1030 mBar. Consequently, a measured value residing within the desired
range should yield a high returned value from the respective sensor. Thus, in this case, 30 degrees Celsius would be the optimal temperature yielding a value of 1.00. The expression calculating the probability of satisfactory weather based on the sensors inputs is found in Table 13.1.

<table>
<thead>
<tr>
<th>Table 13.1 - Excerpt from the Analyser semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. value = temperature.getValue() * pressure.getValue();</td>
</tr>
</tbody>
</table>

This is obviously a very basic calculation, but suffices to illustrate the important points. Notice that the two sensors referred can be used in a real-time context by measuring their current weather property. Alternatively, the sensors can be programmed with arbitrary values making it possible to simulate different scenarios.

A Simulator, as defined in Figure 13.1, can have one or more Visualisers whose purpose is to present the computed result on the screen. A textual presentation is used in this example, as shown in the console output excerpt in Figure 13.2. The console output is the result of executing a model conforming to the Simulator meta model. It was populated with some test input. Notice that the Simulator semantics is created in a similar fashion as for the Car language using the Code Generator. This semantics can be found in the available source code.

...  
[RUNTIME] Method origin: 'interface no.ui.ofi...  
Probability: 37.5 %

Figure 13.2 Executing an example model of Simulator v1.0

13.2 SCENARIO 1 – LOCAL EXTENSION OF LANGUAGE

Languages evolve with time. With respect to the Simulator language it might be necessary to incorporate a new sensor type into the language to increase the fidelity of the calculated probability. There are two ways to perform such an extension. Clearly, the existing language meta model can be extended directly with new concepts. It is also possible to add these concepts using composite language refinement. That is, to define a separate aspect language comprising the new concepts. This can then be woven together with the existing Simulator language. The latter is an advantageous approach since different aspects are expressed in distinct languages. Furthermore, this supports modelling according to selected language views and model recycling (more on this later). Also, the work load of creating a new separate aspect language is nearly the same as updating the existing language directly. Hence, we will use aspect-oriented weaving to refine Simulator. The meta model of an aspect language providing an additional type of sensor is found in Figure 13.3. This language is simply named Analyser.
Note that the concept of Simulator being refined is Analyser. A new sensor named Humidity can be incorporated into Simulator using the meta model of Figure 13.3 and associated semantics. Using the Weaver gives the meta model of Figure 13.4.

As can be seen in Figure 13.4, the new concept has been woven together with the existing concepts of the Simulator meta model. As might be expected, the semantics of the Analyser concept has to be updated to accommodate the new kind of sensor. An updated version of the algorithm calculating the weather prediction probability is found in Table 13.2. As in the first version of Simulator, this algorithm translates to one statement. Notice that the Runtime Environment deals with mapping of syntax and semantics and ensures that the correct semantics is always used for a given concept. In our example this means that the new semantics for Analyser that supports the Humidity sensor is used. (We assume that Simulator and Analyser are compatible.)

In practice, updating a composite DSL as shown above is a simple and straightforward process. Specifically, if the aspect language is available as a finished compatible unit no in-depth knowledge is needed to merge the two languages. In this case, there are merely two tasks that need to be performed; installation of the language and weaving.

In this section we illustrated how a language can be updated by defining an aspect language and weaving this with a primary language. Moreover, this approach allows using the Model Transformer to recycle existing models. In the scenario title the word locally was used. In this context, locally refers to the process of updating a language in-house. However, as expertise is spread around the world this is not always the case. The next scenario will illustrate how to use third-party languages.
13.3 SCENARIO 2 – USING A THIRD-PARTY LANGUAGE

This scenario illustrates how a third-party language can be used to refine a certain aspect of our Simulator language. As an example, we will see how to provide new means of displaying the computed probability on the screen using a language titled Visualiser. We assume that this language has been developed on request. That is, the fictive developers of Visualiser have received the Type Specification, as described in The LDE Platform Tool Suite Extractor of chapter The LDE Platform, and created an aspect language that is compatible (can be unified) with the Visualiser concept of the Simulator v1.1 language. Notice, that the names of the new language and the Visualiser concept of the Simulator are exact the same. There is no specific reason to this other than to illustrate the aspect language’s purpose and problem domain. Naturally, the Visualiser language can be used to model visualisers. An overview of this language’s meta model is depicted in Figure 13.5.

![Figure 13.5 Meta model of Visualiser](image)

As can be seen from the meta model in Figure 13.5, Visualiser is a class whose instances are a composite object. The task of the Presentation and Style concepts are to provide functionality for presentation and styling of the output result. Visualiser is the concept of our existing Simulator meta model that is targeted by the Visualiser language. Installation of the Visualiser language and subsequent meta model weaving result in a composite DSL with the meta model in Figure 13.6.

![Figure 13.6 Meta model for Simulator v1.2](image)

Since the Visualiser language was developed by a third-party on request it can be used directly to update the Simulator language. Or seen from another perspective, the Simulator language’s visualisation aspect is updated merely by copying in Visualiser in the Language Repository, creating a Language Specification (.mldsl file) and weaving the
primary and aspect languages together. All aspects of runtime execution are taken care of by the generic Runtime Environment.

Existing models of Simulator v1.1 can be used together with a model of Visualiser to create updated models conforming to Simulator v1.2 (model recycling). We will illustrate this by example. A model conforming to version 1.1 of Simulator is depicted in Figure 13.7. Notice that two instances of Visualiser are used in the model.

A model created in the Visualiser language is depicted in Figure 13.8.

Using the Model Transformer results in creation of a new model conforming to the Simulator v1.2 meta model. This is shown in Figure 13.9.

Clearly, both objects of the class Presentation are equal as they are copied from the Visualiser model. This is also true for the Style objects. However, this is not the case for the Visualiser objects. As can be seen in Figure 13.7, the model contains two objects of the Visualiser class. These objects are not equal. Thus, they are treated as unique entities by the Model Transformer. More specifically, each Visualiser object of the Simulator model is merged separately with the Visualiser object of the Visualiser
model giving two new unique Visualiser objects. These new objects are stored in the resulting output model. Refer the indexes of the objects in the figures. An overview of the transformation process can be seen in Figure 13.10.

![Diagram](image)

**Figure 13.10 Overview of the transformation process of ma and mb**

As can be seen in the Figure 13.10, a model mc is created based on the two input models ma and mb. mc conforms to the latest version of the Simulator language. Sv1.1 and Sv1.2 refers to the 1.1 and 1.2 versions of the Simulator meta model, respectively. Vv1.0 refers to the meta model of the Visualiser language. Notice that Sv1.1 and Vv1.0 are not needed by the Model Transformer to produce the output model.

Let us execute the new model - mc. Specifically, we would like to have different styles on the output from the two visualisers, as supported by the new incorporated Style concept. An EInt attribute named type of Style can be used to achieve this. Refer Figure 13.5. Giving this attribute different values yields different styles. The available styles are shown in Table 13.3.

<table>
<thead>
<tr>
<th>Attribute value</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Stars</td>
</tr>
<tr>
<td>1</td>
<td>Hyphens</td>
</tr>
</tbody>
</table>

A runtime screen shot of executing the output model of the Simulator v1.2 language is found in Figure 13.11. Each visualiser presents the result successively with different output styles. Notice that the calculated result is different from the earlier output. This is because an additional sensor for humidity is used, thus the result is adjusted.

```
... [RUNTIME] Method invoked: 'language.visualiser...
[RUNTIME] Method invoked: 'language.visualiser...
**************************
Result: 0.16075
**************************
...
[RUNTIME] Method invoked: 'language.visualiser...
[RUNTIME] Method invoked: 'language.visualiser...
-------------
Result: 0.16075
-------------
```

**Figure 13.11 Execution of the output model**
13.4 SCENARIO 3 – DEFINING A THIRD-PARTY LANGUAGE

We have just seen how to use a pre-fabricated compatible aspect language. In this section we will shed some light on how to build such a language. To keep it simple, the requirements specification for our new language describes an alternative way of representing the calculated probability. Specifically, there should be possible to show the probability as a fraction. Let us assume that this language is developed in another department with expertise knowledge of number formatting.

In order to refine the number formatting aspect of Simulator, we must address the concept that is to be defined in a new language. According to the current Simulator meta model, as found in Figure 13.6, we have two choices. We can refine the Visualiser concept or the Presentation concept. Both choices will work fine. However, since the requirements specification describes an alternative presentation of the probability, it is more intuitive to refine the Presentation concept. Consequently, a meta model for a new language titled Presentation is created.

![Figure 13.12 Meta model for Presentation](image)

This language has two concepts: Presentation and Formatter. The latter concept is optional and can be used to format the printed probability. If no Formatter object is used in the model, the default presentation of Visualiser should be used, as shown in Figure 13.11. Formatter has one domain-specific attribute known as type whose value determines the desired formatting. An overview of the available formatting options is found in Table 13.4.

<table>
<thead>
<tr>
<th>Attribute value</th>
<th>Style:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A decimal number</td>
</tr>
<tr>
<td>1</td>
<td>A percentage</td>
</tr>
<tr>
<td>2</td>
<td>A fraction</td>
</tr>
</tbody>
</table>

The semantics module of the language is created using the Code Generator. Clearly, additional code has to be written to meet the specified requirements. Specifically, the semantics must support the three formatting alternatives as specified in Table 13.4. The type of formatting used should be decided by the value of the Formatter.type attribute.
Furthermore, the semantics of Presentation must be integrated with the semantics of Simulator. Recall that the Presentation concept of Simulator is related to the Visualiser concept and the Style concept. These relations must be preserved. Please refer to Figure 13.6. In order to preserve these relations in the semantics, a description of API methods must be provided. Such a description, referred to as the Type Specification, can be created using the Extractor on the existing Simulator meta model. Refer to The LDE Platform Tool Suite Extractor of chapter The LDE Platform for details on the Extractor. The generated Type Specification contains API descriptions for all the relations found in the meta model. Naturally, only the relations between the Presentation, Visualiser and Style concepts are of interest. An excerpt of the altered specification can be seen in Table 13.5.

<table>
<thead>
<tr>
<th>Table 13.5 - Subset of the Type Specification for Simulator v1.2 - simulator_1.2.types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2. &lt;specification&gt;</td>
</tr>
<tr>
<td>3. &lt;meta&gt;</td>
</tr>
<tr>
<td>4. &lt;name&gt;simulator&lt;/name&gt;</td>
</tr>
<tr>
<td>5. &lt;target&gt;</td>
</tr>
<tr>
<td>6. &lt;package&gt;language.presentation.interfaces_sim&lt;/package&gt;</td>
</tr>
<tr>
<td>7. &lt;/target&gt;</td>
</tr>
<tr>
<td>8. &lt;/meta&gt;</td>
</tr>
<tr>
<td>9. &lt;type name=&quot;IStyle&quot;&gt;</td>
</tr>
<tr>
<td>10. &lt;method name=&quot;setType&quot; type=&quot;void&quot;&gt;</td>
</tr>
<tr>
<td>11. &lt;param name=&quot;type&quot; type=&quot;Integer&quot;/&gt;</td>
</tr>
<tr>
<td>12. &lt;/method&gt;</td>
</tr>
<tr>
<td>13. &lt;method name=&quot;getType&quot; type=&quot;Integer&quot;/&gt;</td>
</tr>
<tr>
<td>14. &lt;method name=&quot;applyStyle&quot; type=&quot;String&quot;&gt;</td>
</tr>
<tr>
<td>15. &lt;param name=&quot;data&quot; type=&quot;String&quot;/&gt;</td>
</tr>
<tr>
<td>16. &lt;/method&gt;</td>
</tr>
<tr>
<td>17. &lt;/type&gt;</td>
</tr>
<tr>
<td>18. &lt;type name=&quot;IPresentation&quot;&gt;</td>
</tr>
<tr>
<td>19. &lt;method name=&quot;setFormatter&quot; type=&quot;void&quot;&gt;</td>
</tr>
<tr>
<td>20. &lt;param name=&quot;formatter&quot; type=&quot;IFormatter&quot;/&gt;</td>
</tr>
<tr>
<td>21. &lt;/method&gt;</td>
</tr>
<tr>
<td>22. &lt;method name=&quot;getFormatter&quot; type=&quot;IFormatter&quot;/&gt;</td>
</tr>
<tr>
<td>23. &lt;method name=&quot;process&quot; type=&quot;String&quot;/&gt;</td>
</tr>
<tr>
<td>24. &lt;/method&gt;</td>
</tr>
<tr>
<td>25. &lt;/type&gt;</td>
</tr>
</tbody>
</table>

This specification can be copied into the root folder of the Presentation language’s semantics module and injected using the Injector. Notice that the lines between 5 and 7 are added subsequent of the extraction. These lines specify the target package of the injected types: language.presentation.interfaces_sim.

A principle is to declare all operations of a construct explicitly in the meta model (abstract syntax). There are two reasons for this. Firstly, this ensures that semantics stubs are automatically generated for these operations using the Code Generator. Secondly, the operations are exported using the Extractor. Nevertheless, methods can also be added directly to the semantics interfaces. This is the case with the method process() of interface IPresentation. Consequently, it has to be added manually to the Type Specification, as can be seen at line 23.

Three interfaces have been created in a package named interfaces_sim using the Injector. These interfaces are exact copies of interfaces from the semantics module of the Simulator language, except from the package declarations which differ. By observing the Simulator v1.2 meta model it is clear that the interfaces_sim.IPresentation and
.interfaces_sim.IStyle interfaces need to be incorporated into the Presentation language. The reason for this is the existing relations between Visualiser - Presentation and Presentation - Style.

An object of the Visualiser class is composed of a Presentation object. Naturally, the relation between these concepts has to be maintained in the semantics since a Visualiser object may invoke any of the public methods available on the Presentation object. These methods are described in .interfaces_sim.IPresentation as extracted from the Simulator v1.2 meta model. Consequently, the Presentation class of the Presentation language must implement all these methods. This can be performed in two ways, either by copying the interface content into .interfaces.IPresentation (of the Presentation language) or by extending the .interfaces_sim.IPresentation interface. Please refer .interfaces_sim.IPresentation as found in Table 13.6.

Recall that all constructs of the meta model has a corresponding semantics pair comprising an interface and a class. All public methods of the class are described in its interface. This interface is always used as type in variable and parameter declarations. In other words, an object of an essential semantics class is always referenced by a variable of a supertype (its interface). Conversely, an additional semantics class can be referenced either by a supertype variable or directly using a variable of the class type.

### Table 13.6 - The .interfaces_sim.IPresentation interface

1. package language.presentation.interfaces_sim;
2. public interface IPresentationSim
3. {
4.   public void setStyle( IStyle style );
5.   public IStyle getStyle();
6.   public void process( String data );
7. }

As can be seen in the Simulator v1.2 meta model, a Presentation object is composed of a Style object. To achieve this semantics wise the Presentation object needs to have a reference to the Style object. Consequently, in order to preserve this relation in the new Presentation language, the IStyle interface must be available to the Presentation class. This can be achieved by copying the interfaces_sim.IStyle interface into the .interfaces package. Notice that it is up to the language developer to use the features provided by a Style object. This may be achieved by invoking one of the available API methods of IStyle as can be seen in Table 13.7. In a later chapter we will discuss how semantics can be inherited.

### Table 13.7 - The .interfaces_sim.IStyle interface

1. package language.presentation.interfaces_sim;
2. public interface IStyle
3. {
4.   public void setType( Integer type );
5.   public Integer getType();
6.   public String applyStyle( String data );
7. }
It is not necessary to implement IVisualiser or make this type available to the semantics of the Presentation language since objects of this type is not referenced by the Presentation concept (or other concepts of the Presentation language). On the contrary, a bi-directional reference between Visualiser and Presentation would require incorporation of IVisualiser into the Presentation language as well. Notice that the interfaces of the Simulator language could have been injected directly into the .interfaces package. This, however, would overwrite the original IPresentation interface as previously generated by the Code Generator.

The next step is to update the language semantics to meet the specified requirements. Firstly, the Presentation class has to be completed. Notice that .interfaces_sim.IPresentation has been renamed to IPresentationSim and copied into the .interfaces package.

<table>
<thead>
<tr>
<th>Table 13.8 - Overview of the IPresentation interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. package language.presentation.interfaces;</td>
</tr>
<tr>
<td>2. public interface IPresentation extends IPresentationSim</td>
</tr>
<tr>
<td>3. {</td>
</tr>
<tr>
<td>4. public void setName( IFormatter formatter );</td>
</tr>
<tr>
<td>5. public IFormatter getName();</td>
</tr>
<tr>
<td>6. }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 13.9 - The Presentation class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. package language.presentation.classes;</td>
</tr>
<tr>
<td>2. import language.presentation.interfaces.IPresentation;</td>
</tr>
<tr>
<td>3. import language.presentation.interfaces.IFormatter;</td>
</tr>
<tr>
<td>4. import language.presentation.interfaces.IStyle;</td>
</tr>
<tr>
<td>5. public class Presentation implements IPresentation</td>
</tr>
<tr>
<td>6. {</td>
</tr>
<tr>
<td>7. private IStyle style;</td>
</tr>
<tr>
<td>8. private IFormatter formatter;</td>
</tr>
<tr>
<td>9. public void setName( IStyle style ) { this.style = style; }</td>
</tr>
<tr>
<td>10. public IStyle getName() { return style; }</td>
</tr>
<tr>
<td>11. public void setName( IFormatter formatter )</td>
</tr>
<tr>
<td>12. public IFormatter getName() { return formatter; }</td>
</tr>
<tr>
<td>13. {</td>
</tr>
<tr>
<td>14. if( formatter != null )</td>
</tr>
<tr>
<td>15. data = formatter.format( data );</td>
</tr>
<tr>
<td>16. return style.applyStyle( data );</td>
</tr>
<tr>
<td>17. }</td>
</tr>
<tr>
<td>18. }</td>
</tr>
<tr>
<td>19. }</td>
</tr>
</tbody>
</table>

IPresentation and Presentation are found in Table 13.8 and Table 13.9. Notice that IPresentation extends the IPresentationSim interface. Presentation contains implementations of all the methods specified by IPresentation (11 and 12) and IPresentationSim (9, 10 and 13 till 18). Lines 15 till 17 describe the custom semantics for the class. It invokes the method format(...) on the Formatter object if such object exists. In practice this depends on the model that is being executed. Recall that Formatter is an optional construct according to the Presentation meta model. A style is
applied to the formatted data using the `applyStyle(...)` method on the `Style` object, in a similar fashion as is done by the `Presentation` semantics of Simulator. Be aware that `setStyle(...)` at line 9 and `setFormatter(...)` at line 11 are invoked by the Runtime Environment during the object model generation phase with the appropriate proxies.

Table 13.10 - Overview of the `IFormatter` interface

```java
1. package language.presentation.interfaces;
2. public interface IFormatter {
3.  
4.  public static final int DECIMAL = 0;
5.  public static final int PERCENT = 1;
6.  public static final int FRACTION = 2;
7.  public void setType( Integer type );
8.  public Integer getType();
9.  public String format( String data );
10. }
```

Table 13.11 - Excerpt of the `Formatter` class

```java
1. package language.presentation.classes;
2. import language.presentation.interfaces.IFormatter;
3. 
4. public class Formatter implements IFormatter {
5.  
6.  private int type;
7.  
8.  public void setType( Integer type ) { this.type = type; }
9.  public Integer getType() { return type; }
10. 
11.  public String format( String data ) {
12.     double value = Double.parseDouble( data.substring( data.indexOf( " " ) + 1 ) );
13.     if( type == IFormatter.DECIMAL )
14.         return "Result as decimal: " + data;
15.     else if( type == IFormatter.PERCENT )
16.         return "Result as percent: " + value * 100 + " %";
17.     else if( type == IFormatter.FRACTION )
18.         return "Result as fraction: " + decimalToFraction( value );
19.     }
```

`IFormatter` and `Formatter` are found in Table 13.10 and Table 13.11. `format(...)` is invoked at runtime (interpretation phase) from the `Presentation` object. This method formats the probability data according to the value of the `type` attribute. This can be seen between lines 8 and 18. The `type` attribute is initialised with the value from the language model by the Runtime Environment during the object model generation phase. This is achieved by invocation of `setType(...)` at line 6. The formatted probability is returned to the `Presentation` object, and subsequently, printed to screen by the `Visualiser` object.

As might be expected, the Presentation language can now be used to refine the presentation aspect of Simulator. The new meta model after weaving can be viewed in Figure 13.13.
Creating and executing a model in the new 1.3 version of Simulator gives the example console output of Figure 13.14. As earlier, the model uses two instances of **Visualiser**.

```plaintext
... [RUNTIME] Method invoked: 'language.visualiser...
[RUNTIME] Method invoked: 'language.presentati...
******************************
Result as fraction: 16875/100000
******************************
...
[RUNTIME] Method invoked: 'language.visualiser...
[RUNTIME] Method invoked: 'language.presentati...
------------------------
Result as percent: 16.875 %
------------------------
```

Figure 13.14 Executing an example model of Simulator v1.3

A new atomic language named Presentation was created in this scenario. Alternatively, other pre-factored languages could have been used to define this language. For instance, the **Formatter** concept of the Presentation language could have been refined using a specific aspect language for number formatting. Moreover, one or more concepts of such number formatting language could already have been defined using other languages, and so on. In practice, it is no difference using an atomic language or a composite language when extending / customising a language. They can be used interchangeably.

We will use the 1.3 version of Simulator to exemplify the execution sequence of the semantics classes, known as the interpretation phase. Refer the section *Runtime Environment* of chapter *The LDE Platform*. In other words, how the program control is propagated through the runtime Java object model. The object model constructed from the program used to produce the console output of Figure 13.14 is shown in Figure 13.15. All method calls / messages are synchronous.
As noted, the generated object model is evaluated invoking the method `process()`, here implemented by the semantics class `Simulator`. Each method call is prefixed with a number indicating the call’s position in the overall invocation sequence. The generated proxy definitions are also listed. It is the language developers’ responsibility to design efficient semantics that propagates the program control in a sound manner. Propagation of control can be analysed based on a log written to the console by the Runtime Environment. The complete runtime log for the preceding example is found on the accompanying CD. The purpose of an analysis is to validate the logic of the semantics. In other words, it is possible to gain an overview of the behaviour of the composite DSL.

### 13.5 Scenario 4 – Updating a Concept’s Semantics

We have seen how to update the syntax and semantics of a composite DSL to meet new requirements using an aspect language. In some cases it might be sufficient to merely update the semantics of one or more constructs. In other words, the existing DSL syntax suffices to model the concerns of the problem domain. With regard to the Simulator example, updating the algorithm calculating the final probability does not require introduction of new language constructs. Clearly, a revised `Analyser` semantics would suffice. The same approach for refinement of a composite DSL, as described in the previous scenarios, can be used to update the semantics. This implies that an appropriate aspect language is incorporated into the Simulator language. However, this new language only contains one concept. This concept has the same name as the construct to be revised (otherwise renaming is used). It can be argued that a language consisting of merely one concept is not a true language. Nevertheless, we will use the same notions to avoid
introducing new terms.

**Figure 13.16 Meta model of AnalyserRevised**

A new language titled *AnalyserRevised* will be used to refine the semantics of the Analyser concept. Its meta model is shown in Figure 13.16. Analyser has one attribute which specifies the semantics class of the concept. Recall that this attribute is named semantics. As can be seen in Figure 13.13, Analyser is related to the concepts Simulator, Temperature, Pressure and Humidity. These relations have been preserved by injecting a set of interfaces into the AnalyserRevised language’s semantics module. The updated Analyser algorithm is found in Table 13.12.

![Analyze](image)

<table>
<thead>
<tr>
<th>Table 13.12 - Updated algorithm of Analyser</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. value = temperature.getValue() * pressure.getValue() *</td>
</tr>
<tr>
<td>2. Math.pow( humidity.getValue(), 0.35 );</td>
</tr>
</tbody>
</table>

Notice that the only difference from the prior algorithm is scaling of the humidity value, as seen on line 2. Weaving the AnalyserRevised language with the existing composite DSL gives a new meta model which closely resembles the meta model of Simulator v1.3. There is one difference: the value of Analyser’s semantics attribute now points to the new semantics. That is, a mapping between the Analyser syntax and the refined Analyser semantics has been created.

An excerpt of the Language Specification for the final 1.4 version of Simulator is shown in Table 13.13. This file describes the different languages that constitute the composite DSL. Or, from another perspective; the aspects of Simulator that have been refined. This file can be used to create the 1.4 version of Simulator in one user operation.

<table>
<thead>
<tr>
<th>Table 13.13 - Excerpt of the Language Specification for Simulator 1.4 - Simulator_v1.4.mldsl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;language name=&quot;Simulator&quot;&gt;</td>
</tr>
<tr>
<td>2. &lt;language name=&quot;Analyzer&quot; defines=&quot;Analyzer&quot;/&gt;</td>
</tr>
<tr>
<td>3. &lt;language name=&quot;Visualiser&quot; defines=&quot;Visualiser&quot;&gt;</td>
</tr>
<tr>
<td>4. &lt;language name=&quot;Presentation&quot; defines=&quot;Presentation&quot;/&gt;</td>
</tr>
<tr>
<td>5. &lt;/language&gt;</td>
</tr>
<tr>
<td>6. &lt;language name=&quot;AnalyzerRevised&quot; defines=&quot;Analyzer&quot;/&gt;</td>
</tr>
<tr>
<td>7. &lt;/language&gt;</td>
</tr>
</tbody>
</table>

...
PART IV
DISCUSSION
14 Aspects of composite DSLs

This chapter presents discussions on important topics regarding composite DSLs. Some topics are treated more comprehensively due to their importance.

The main structure of this chapter is as follows:

- Core aspects
- Comprehensive scenarios
- Separation of concerns in composite DSLs
- Using third-party semantics
- Defining a General Purpose Language
- Language design methodologies

Objective: Extension and customisation of languages
Contribution: Designing a composite DSL, The Ecommerce framework (Weaver, Extractor, Injector, standalone Runtime Environment)

14.1 CORE ASPECTS

In this thesis we have suggested an approach for design and development of composite DSLs. The ideas and concepts have been verified using a set of proof-of-concept applications. Here we will shed some light on various aspects of composite DSLs that have not been addressed explicitly in the other parts of this thesis. This includes specialisation and reuse of semantics, the stakeholders involved in language development and applications of composite DSLs.

14.1.1 SPECIALISATION OF SEMANTICS USING SUBTYPING

Subtyping can be used to specialise the semantics of a language. This is achieved in the traditional sense by extending a semantics class. We will exemplify this using the Car language.

Table 14.1 - Specialisation of Car

```java
1. public class CarS1 extends Car
2. {
3.   public void process()
4.   {
5.     ...
6.     super.process();
7.   }
8. }
```

Table 14.2 - Alternative specialisation of Car

```java
1. public class CarS2 extends Car
2. {
3.   public void setEngine( IEngine engine )
4.   {
5.     ...
6.     this.engine = engine;
7.   }
8. }
```
Two specialisations of the previous introduced Car semantics class are shown in Table 14.1 and Table 14.2. These specialisations are created using subtyping. Moreover, the methods process() and setEngine(...) are redefined using overriding. The semantics attribute of the meta model class Car has to point to the appropriate specialisation in order for the new semantics definition to be used by the Runtime Environment. As can be seen, subtyping is an ideal mechanism for specialising a language’s semantics.

14.1.2 REUSE AND INTEGRATION OF SEMANTICS

Refining a language implies that new semantics has to be provided for one or more of the language’s constructs. In many cases, it is desirable to build on the existing semantics of the primary language. This is perfectly possible by subtyping the appropriate classes. Unfortunately, this introduces new dependencies between the primary and aspect languages’ semantics modules. Furthermore, it requires the superclasses to be available for the developer of the aspect language. As might be expected, this makes the aspect language less flexible since it depends on the semantics of the primary language. In principle, an aspect language should be usable for refinement of several primary languages, thus, subtyping classes of the primary language is not a proper solution. Recall that interfaces are the single point of unification between languages semantics wise.

There are mainly two types of refinement scenarios:

- An aspect language is created on-demand based on specified interfaces
- A generic aspect language is incorporated into the primary language

An aspect language is created on-demand based on specified interfaces

An aspect language created on-demand can be fully integrated with the target primary language “out of the box”. This is because all necessary interfaces are implemented by the aspect language. A future version of the Extractor may export critical semantics code as well, making it easy to reuse semantics for outsourced language development.

A generic aspect language is incorporated into the primary language

The semantics of a generic aspect language, on the other hand, may need to be slightly specialised to be compatible with a primary language. Such an aspect language may for instance be downloaded from a community web site. Recall that semantics modules of languages are not woven together, but kept apart. As described, refining one or more constructs of a language implies using new semantics for these constructs as defined by the aspect language. This new semantics does not necessarily take care of existing interdependencies between the constructs being refined and other constructs of the primary language. Moreover, properties of the existing semantics are not preserved by the new semantics. These issues can be addressed by updating the semantics of the acquired aspect language using subtyping. An example will illustrate this.
Two languages are identified in Figure 14.1. Language A comprises two constructs A and B, while language B only has one construct B*. As can be seen, B and B* has a set of operations. The language B is used to refine the B construct of language A. Weaving these constructs together gives the resulting construct B**. This construct declares all operations of both B and B* (syntactic wise).

Naturally, the semantics class for B* is used as semantics class for the resulting construct B**. The operations op1() and op2(...) are not implemented in the semantics class for the B* construct. Refer Table 1.4.3 and Table 1.4.4. Thus, these have to be added manually by the primary language developer using subtyping of B*. Refer Table 1.4.5. Be aware that this subtyping updates the inherent semantics module of the B language and not the A language. As a consequence, there is not introduced a dependency between the semantics modules of the two languages.

Table 1.4.3 - Semantics class for B

| 1. package language.A.classes; |
| 2. public class B |
| 3. { |
| 4. public Integer op1() {} |
| 5. public void op2(String s) {} |
| 6. } |

Table 1.4.4 - Semantics class for B*

| 1. package language.B.classes; |
| 2. public class B |
| 3. { |
| 4. public Double op3(Byte b) {} |
| 5. public Integer op4(Character c) {} |
| 6. public void op5() {} |
| 7. } |
Finally, the `semantics` attribute of `B**` has to refer the new semantics class, `BSub`. That is, its value should be updated to `language.B.classes.BSub`.

Future work may include investigation of semantics weaving. This can possibly be addressed by generating a subclass where the missing attributes and methods are injected into the class. This was shown manually in the above example.

### 14.1.3 SEMANTIC COHERENCE

Semantic coherence is a term used to represent the state of achieving semantically rich associations between different artefacts in a software development process [29]. This is not a trivial issue since artefacts are defined using various formalisms and technologies. One approach in order to link artefacts is to create a common meta model that describes the explicit link between the artefacts. However, in many situations this is not practical. A proposed approach in [29] is to create *asset descriptions* whose purpose is to specify an explicit meaning to an association between two or more artefacts. Examples of artefacts are textual requirements specifications and design documents, visualisations and diagrams, and implemented code. In the context of composite DSLs, we will also treat each language as an artefact. The reason for this is that, according to dynamic modelling (as supported by refinements), a DSL is a significant asset used in a language development process, where the goal is to create a composite language.

In general, linking two arbitrary languages (as defined using different formalisms) can potentially introduce unpredictable results like side effects and other anomalies. Furthermore, it can be difficult to formally deduce how the combined semantics will operate. Clearly, an important aspect of language composition is semantic coherence. To achieve full semantic coherence languages are woven together. This is possible since each language conforms to a common formalism, namely the Ecore meta-meta model. This formalism represents a semantically rich link between primary and aspect languages’ abstract syntax. Moreover, the interaction of semantics, as defined by the languages’ interfaces, assures that all communication between primary and aspect languages’ constructs is formal and explicit. This ensures a semantically rich interoperation between the languages. As a consequence, it is easy to deduce what parts of a language to revise or elaborate as requirements are changed or updated. Also, it is possible to reason about and analyse the composite language with a higher degree of accuracy due to traceability in the language development process.

An interesting aspect with regard to language design / analysis is to try and use first-order logic to investigate if the process of combining a set of sound atomic languages preserves truth. That is, if the new composite language is sound as well. In this respect, a *sound* language implies that it can only be used to create logical and reasonable models. This can easily be assessed for one atomic language. However, the meta model weaving and
interaction of semantics may potentially destroy the integrity of the atomic languages by introducing logical anomalies. Such anomalies can eventually be prevented by structuring languages differently. Due to time constraints, investigation of truth-preserving language weaving has not been prioritised. Nonetheless, no violation of language integrity has been observed so far.

14.1.4 INTEGRATION WITH OTHER TECHNOLOGIES
An important aspect of software development is integration of artefacts. Here we will see how languages of the LDE Platform can be integrated with relevant technologies.

Integration with EMF
A composite DSL meta model is defined in terms of Ecore concepts. Consequently, this meta model can be used directly in EMF without modifications. This is achieved by creating a Generator Model from the meta model using an appropriate model importer of EMF. As we have seen, meta models defined in EMF can be used as basis for composite DSLs as well. This, however, requires all meta classes to declare a semantics attribute. A future tool could add such attributes automatically.

Conversion between meta model architectures
There is technology available for converting Ecore meta models to MOF and vice versa. Thus, the meta model of a composite language can serve as starting point for another language or system based on a different meta model architecture.

Compatibility with third-party software
Integration with third-party software can easily be performed since the semantics module of a language is defined using general purpose host language concepts. This includes technologies like Hibernate, JSecurity, JavaFX, JBoss and AspectJ. Note that the Runtime Environment can be integrated directly with such technologies as well.

Standalone environment ensures independence
The standalone version of the Runtime Environment ensures that composite DSLs can be used in other contexts than Eclipse. Specifically, the standalone Runtime Environment is available as a library which makes integration with other applications easy.

Generic decentralised environment
Semantics types can easily be exported to a generic format using the Extractor. Subsequently, the types can be injected into a composite DSL defined using another host language. This requires a compatible Injector written in the alternative host language.

Available tools
Tools compatible with Java, UML, ATL, Kompose and Ecore can be used in design and development processes. For instance, Ecore Tools were used during the course of this thesis.

14.1.5 APPLICATIONS OF COMPOSITE DSLs
There are many applications of composite DSLs. In theory, composite DSLs can be substituted for frameworks and traditional DSLs. Obviously, this depends on the required degree of integration with tools and other technologies, and the type of platform the composite DSL complies with (Java, C++, assembly, etcetera). In this thesis, the LDE Platform is implemented using Java. Thus, composite DSLs in this context would
integrate well with Java-based technologies.

Composite DSLs are especially suited for situations where the problem domain and requirements specification are highly dynamic as refinement and extension are first-class operations of composite DSL development. Moreover, composite DSLs provide mechanisms that support construction of arbitrary languages and language versions. Specifically, software engineers can combine the aspect languages that will address their concerns.

14.1.6 STAKEHOLDERS
For the most part, there are three types of stakeholders related to design, implementation and use of a composite DSL, as inspired by [30]. These are:

- Problem domain analysts and language developers / software engineers
- Model and program developers that use the composite DSL to express / model concerns
- Domain experts and customers / business analysts

An interesting aspect of composite DSL development is that a language can be created as a common effort. That is, different parts or segments of a language can be designed and implemented separately and then be combined in the final stages of development. This promotes a close cooperation between the different stakeholders. A consequence is easier tracing between artefacts of the development process. An example of this is tracing of requirements from specification to implementation. This is also referred to as Participatory Design. It may also be argued that quality assurance routines can be performed more easily since the language development process is more open. Also refer the sub section Semantic coherence.

14.1.7 SEVERAL ROOT CONCEPTS
For the most part, meta models have one single root concept. With respect to the Runtime Environment, program control is always initiated from the semantics class of this root concept by invocation of the method process(). This method is described in the IExecutable interface. It is possible to have several root concepts in a meta model. However, the practical applications of this are few. As a consequence, the Runtime Environment does not implement a mechanism for selecting the target executable construct, even though it is possible to declare several constructs as executable. A future version of the Runtime Environment may support selection of several executable constructs by providing runtime configurations. This includes support for sequential and concurrent (multi-threaded) runtime scenarios. As a result, the semantics of a language can be analysed more efficiently. Moreover, error detection and debugging can be simplified because a specific part of the semantics can be run separately (partial model execution).

14.1.8 CONCRETE SYNTAX
There has not been much focus on concrete syntax. The default concrete syntax provided by EMF has been utilised. Obviously, a concrete syntax can be developed and included as part of a language’s installation package. This includes both textual and graphical syntaxes. In the context of the Eclipse workbench this can be achieved by creating a custom language editor. Later we will see an example of a textual concrete syntax for a small General Purpose Language.
14.2 COMPREHENSIVE SCENARIOS

To further illustrate how aspects of DSLs are refined we will include a couple of more comprehensive examples.

14.2.1 TANGLED LANGUAGE CONSTRUCTS

Clearly, language constructs can be more tangled than what have been seen in the preceding Simulator example. Consider the following meta models.

\[X\] is a tangled concept of language \[A\]. See Figure 14.2. There are five relations between this concept and the other concepts of the language. In fact, \[X\] is related to all concepts of the language. \(B\) is a language for modelling of \(X\) concerns. It comprises three constructs, including a root construct conveniently named \(X\). Refer Figure 14.3. It is compatible with the \(A\) language. Clearly, \(B\) can be used to refine the \(X\) construct of \(A\). Let us anticipate events and investigate the resulting meta model after weaving the two meta models together.
As can be seen in Figure 14.4, there are now eight relations between \( x \) and the other concepts of the AB meta model. All these relations have to be supported by the language semantics. Note that the \( x \) syntax has been updated with \( x\text{opB}(\ldots) \). Corresponding semantics for this operation is defined in the B language. As stated, B is a language compatible with A. Or in more precise terms, B can be used to refine the \( x \) construct of the A language. To achieve this, B must incorporate all API methods of the A language’s \( x \) construct. For the sake of clarity, no attributes are specified in the language classes except the semantics attribute. This attribute is non-changeable and has a default value, thus no API methods are generated by the Code Generator for this property. Its value is only of interest to the Runtime Environment. We have earlier illustrated how to design a third-party aspect language that is compatible with a primary language “out of the box”. This purpose of this example is to describe in more details how to create compatible languages that can be used to refine tangled concepts.

Table 14.6 - Excerpt of the Type Specification for A - A_v1.0.types

Let us return to the meta model for A and B. Using the Extractor on the A meta model and the Code Generator on the B meta model give the initial entities needed to define a
compatible B language. These are the semantics module for B and the Types Specification for A, respectively. Notice that the meta models of both languages specify operations.

The developer of the B language needs to incorporate the essential interfaces from A. These interfaces, as described in the exported Type Specification, can be injected into the B language. An extract from the Type Specification is seen in Table 14.6. It contains the specification for the type IX. All methods of this extracted type have to be implemented by the x construct of B.

### Table 14.7 - Extract of X

```java
... 
1. public class X implements IX {
2. | 
3. | private IA2 a2;
4. | private IA4 a4;
5. | private IB1 b1;
6. | private IB2 b2;
7. // From IX
8. | public void setB1( IB1 b1 ) { this.b1 = b1; }
9. | public IB1 getB1(){ return b1; }
10. | public void setB2( IB2 b2 ) { this.b2 = b2; }
11. | public IB2 getB2() { return b2; }
12. // From IXA
13. | public void setA2( IA2 a2 ) { this.a2 = a2; }
14. | public IA2 getA2() { return a2; }
15. | public void setA4( IA4 a4 ) { this.a4 = a4; }
16. | public IA4 getA4() { return a4; }
17. | public void xop1( Integer par1, Double par2 ) {}
18. | public void xop2() {}
19. | public void xop3() {}
20. }
```

Note that the interface IX injected from A is renamed to IXA. All methods of IX and IXA are implemented by X. Most of these methods are simple set and get methods. The additional methods prefixed with xop can be used to implement the behaviour of X. These methods may invoke methods on objects of the related semantics classes, as accessed using the instance variables a2, a4, b1 and b2. Please refer Table 14.7. As can be seen in this latter example, creating languages that can be used for refinement of tangled constructs can easily be created using the Extractor and Code Generator.

#### 14.2.2 Refining Properties for Multiple Constructs

Another interesting scenario to explore is refinement of properties for multiple constructs. We have already seen how to update the semantics for one construct without introducing new language constructs. This process can be generalised to comprise several constructs. Furthermore, it is also possible to add attributes or operations to the constructs using the same approach. The main point is the possibility to tailor existing language constructs of a composite DSL without introducing new constructs or relations. An example of this is described on the accompanying CD.
### 14.3 Separation of Concerns in Composite DSLs

An advantage with Language Driven Development [1] is the freedom to create custom abstractions. This implies that a first-class construct can be designed to address a certain aspect of the problem domain. In theory, this includes cross-cutting concerns as well. However, in many cases it is not possible to represent a concern using merely one construct. Let us consider a couple of examples.

**Figure 14.5 Refinement of construct B using an aspect language B**

A simple language comprising the four constructs A, B, C and D can be seen in Figure 14.5. The construct B is refined using a language B consisting of the constructs B1 and B2. This is a straightforward refinement scenario. A cross-cutting concern named Aspect E is addressed by the constructs B, C and D. We will assume that this concern is logging of runtime data. Thus, each of the constructs defines operations for storing of status information.

**Figure 14.6 Refinement of aspect E using an aspect language E**

It is possible to use a language for refinement of multiple constructs simultaneously. In Figure 14.6, a language named E comprising the constructs E1, E2, E3 and E4 are used to elaborate the constructs B, C and D. As pointed out, a concern (Aspect E) is reflected by the collective operation of the latter constructs. Hence, they can be updated simultaneously to reflect the cross-cutting concern more precisely. In the case of our example, a possible refinement could be updating routines for logging of information. This example is of course trivial, but illustrates partly how cross-cutting concerns can be dealt with. Obviously, the two refinement processes illustrated in the figures can be executed in any order since the two languages B and E are disjunctive (we assume that the B and E languages provide semantics for the B construct that can be unified).
14.4 USING THIRD-PARTY SEMANTICS

There are many ways to define semantics. Until this point, we have referred to a language’s semantics as the semantics module consisting of Java compilation units like classes, interfaces and enums. However, a language’s semantics can also be defined in terms of other concepts than what is natively available in Java.

There are a large number of frameworks available. These frameworks provide concepts for modelling and computing within a specific domain. Many of these have been developed incrementally over a long period of time and have proved to be efficient and stable. Moreover, they capture expertise knowledge and make this knowledge available in a formalised manner. Creating languages that utilise such frameworks promotes reuse and ensures stability. This section elaborates on how to use a framework to provide the semantics foundation for a composite DSL. We will create a Webshop DSL that uses the Ecommerce framework as described in chapter The Ecommerce framework. First off, let us start by defining a meta model for the Webshop language. This is shown in Figure 14.7.

![Figure 14.7 Meta model for the Webshop language](image)

As seen in Figure 14.7, the meta model provides concepts for modelling of simple web shops. According to the perspective used in the problem domain analysis phase, web shops comprise one or more pages. Each of these pages contains one or more functions. There are six functions defined by the meta model. These are MMenu, MProductList, MShoppingCart, MCheckout, MPayment and MConfirmOrder. Each function can have an arbitrary name. Furthermore, each Webshop model must specify one initial function. This is specified by setting the initState attribute to true. Thus the page at which this function resides is loaded when a customer visits the web shop. The meta model is kept simple to get the broad view. Each construct is prefixed with an “M” to avoid eventual naming conflicts with concepts of the framework.
In previous examples, all language semantics has been provided directly in the essential interfaces and classes of the language’s semantics module. It is also possible to add additional interfaces, classes and enums to the semantics module that are not directly mapped to the abstract syntax. There are no rules to the naming of these additional units, or the packages in which they reside. This makes it possible to add frameworks and tools directly to the semantics module of a language. Alternatively, libraries and frameworks enclosed in .jar files can be instantiated directly from classes or enums of the semantics module. These alternatives are known as *semantics configurations*. An overview of the different semantics configuration options is found in Table 14.8.

<table>
<thead>
<tr>
<th>Option:</th>
<th>Variation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>All semantics is enclosed in essential semantics units</td>
<td></td>
</tr>
<tr>
<td>Semantics is distributed among essential and additional units</td>
<td></td>
</tr>
<tr>
<td>Third-party software is included directly in the semantics module</td>
<td>Using an integration layer</td>
</tr>
<tr>
<td>Third-party software is instantiated from essential units of the semantics module</td>
<td>Using an integration layer</td>
</tr>
<tr>
<td>Semantics is distributed using a combination of the above options</td>
<td>Using an integration layer</td>
</tr>
</tbody>
</table>

In some situations it can be desirable to centralise instantiation and initialising of third-party software. This can be achieved using an *integration layer* that works as a buffer between the semantics module and additional software constituting the language semantics. Moreover, this layer provides mappings between concepts of the composite DSL and concepts of the additional software. This is illustrated in Figure 14.8.

![Figure 14.8 Integration with additional software](image)

In the remainder of this section, we will illustrate how to combine the Webshop DSL with the Ecommerce framework to form a self-contained bundle that can be deployed as a web application. This web application should support modelling of different web shops using the DSL. An integration layer will be used between the semantics module and framework.

We have earlier described the Ecommerce framework and the structure of semantics modules. Hence, focus will be put on how the two entities are linked together. In this example, the integration layer is implemented as a single class named `Integration`. An excerpt of this is found in Table 14.9.
Please recall that the most straightforward manner of using and customising the Ecommerce framework is by defining a configuration. This configuration describes all aspects of a web shop including states, navigations and tasks. Thus, the purpose of the Integration class is to build a framework configuration based on the provided Webshop DSL model. Different Webshop models give different configurations. We will shed light on the most important parts of the code in Table 14.9. Firstly, the standalone Runtime Environment is instantiated at line 3. The MLDSLDriver instance is then used to compile a Java object model based on the Webshop meta model and a Webshop model, as seen at line 4. As can be seen in Figure 14.7, each web shop consists of a number of pages. These are iterated at line 14 and used to build a set of States and Pages which are added to the configuration. Notice that the State and Page concepts are parts of the Ecommerce framework. At line 18, the functions of the pages, as described in the web shop model, are iterated. These functions are described in the configuration using the framework-specific concept Function. Furthermore, each function is associated with pre and post tasks and a set of navigation options. How this is implemented can be seen in the available source code.

In this example, the concepts of the meta model are closely related to concepts of the framework. This is not always the case due to different abstractions used by the language and third-party software. In such situations, the integration layer has to create composite mappings between concepts. Specifically, these mappings will relate one concept of the language to several concepts of the framework or vice versa. It could also be necessary to create mappings directly between distinct concept properties, like attributes and operations. Mappings between language and framework concepts are illustrated in Figure 14.9. Refer Meta modelling → Mappings of chapter Language design.
The last step in the integration process of the framework and DSL is to initialise the framework with the configuration created in the `Integration` class. In this example, the `Integration` class is instantiated from the servlet class of the web application. An extract of the `init()` method of the servlet class is found in Table 14.10.

As seen at line 3, the `Integration` class' constructor takes two parameters: the metamodel and model of the DSL to use. These entities are used to create the framework configuration. This configuration is used by the framework controller to build the desired framework, as seen at line 4. In this example, the Webshop model is selected prior to deployment of the servlet. Another approach would be to create an admin page to support ad-hoc selection of Webshop models.

An overview of the complete software bundle is found in Figure 14.10. It is organised as a Web Archive and packed in a file named `webshop.war`. The complete bundle runs as a J2EE Servlet.
As can be seen in the Figure 14.10, the Webshop language components are found in two folders. The *metamodel* folder contains the Webshop meta model, while the *WEB-INF/classes/language/webshop* folder contains the Webshop semantics module. Furthermore, the Ecommerce framework and integration layer are found under *WEB-INF/classes/no/uio/ifi/hennb*. Notice that the framework classes are added directly to the software bundle in a similar manner as described by the configuration option

*Third-party software is included directly in the semantics module.*

The *lib* folder contains two libraries: one for accessing a MySQL database used by the Ecommerce framework and one containing the standalone Runtime Environment as used by the integration layer. *models* contains three web shop models. *mldsl.properties* is a configuration file that specifies DSL properties. In this case it describes what meta model and model to use. Or in other words, what kind of web shop the web application represents. See Table 14.11. *web.xml* is used to configure the servlet, while *style.css* defines the graphical styles used in the web shop.

<table>
<thead>
<tr>
<th>Table 14.11 - MLDSL properties - mldsl.properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. metaModel=WEB-INF/metamodel/Webshop.ecore</td>
</tr>
<tr>
<td>2. model=WEB-INF/models/webshop.xmi</td>
</tr>
</tbody>
</table>

We will create a couple of different web shop versions to illustrate the finished web application solution. Consider the following model.

![Figure 14.11 A simple Webshop model](image)

**Figure 14.11** shows a Webshop model comprising five objects. It describes a web shop consisting of one page with three functions: a menu, product list and shopping cart. Notice the links between the *MMenu* object and the *MProductList* and *MSHoppingCart* objects. These links represent the navigation patterns of the web shop. Thus, it is possible to navigate to the product list and shopping cart from the menu. In this initial web shop though, all functions are displayed on the same page. This means that the navigation link from *MMenu* to *MSHoppingCart* is practically unnecessary. The link from *MMenu* to *MProductList* is needed to trigger the task that populates the product list with items. Navigation patterns make more sense when the web shop has several pages. A screen shot of the web shop built from the model of Figure 14.11 is shown in Figure 14.12. Obviously, this is not a complete web shop. However, it serves its purpose as an example.
As can be seen in Figure 14.12, the resulting web shop consists of the three functions specified in the model. The product list has been populated with products from the database after clicking Product list in the menu. Each of the products in the product list has been manually added to the shopping cart. In this case, three mother boards. We will elaborate on the first model and update the web shop with an additional page comprising more web shop functions.

As can be seen in Figure 14.13, the new web shop provides an additional page with the new functions MCheckout, MPayment and MConfirmOrder. In addition, all valid navigation patterns have been provided in the model. For instance, it should be possible to navigate from the shopping cart to checkout in the resulting web shop. Hence, the lower part of the model is a directed graph and can virtually be observed as a state diagram.

Clearly, it would be possible to incorporate explicit support for modelling of states in the Webshop meta model. According to the Webshop meta model navigation patterns can be created between all functions. There are, however, navigation routes that are less logical. These can be disallowed by specifying a set of OCL constraints.
Recall that each function of the Ecommerce framework can be associated with a set of pre and post tasks. For instance, the product list has to be populated with products from the database. This is achieved by defining a pre task using two services named `ProductsBusinessService` and `ProductsDataService`. These services have to be instantiated and used in the correct manner. Using the framework directly requires insight in defining tasks and using the appropriate services. However, it is possible to create a standard generic specification for all functions in the framework. Such specification is incorporated in the `Integration` class. Thus, a model-driven definition of web shops can utilise this specification directly making it unnecessary to deal with tasks and services. It is also possible to add additional properties to the Webshop meta model. These properties can describe preferences regarding the underlying processing in the framework. Consequently, all aspects of the desired web shop can be expressed in a model (no framework instantiation code has to be manually created). Hence, web shop developers without software engineering experience only need to deal with domain models (web shop concepts) within their area of expertise. This is a concrete example of DSLs / composite DSLs usability and expressiveness. Please refer `The Ecommerce framework` chapter for details.

We have examined how a third-party framework can be used to express a Webshop language’s semantics. This approach can be generalised to support all kinds of frameworks and libraries. This includes libraries for mathematical representations, like denotational semantics, and logical representations like theorem provers. Thus, the semantics module works as a superstructure that binds the constructs of a language with its underlying semantics foundation. Other types of formal semantics that can be useful are Kripke automation, Petri nets and Process algebra [31]. Also note that the concepts and ideas discussed can be implemented on other platforms than Java.

### 14.5 Defining a General Purpose Language

Until this point we have mostly considered DSLs. There are, however, no limitations to the domain that can be reflected by a composite language. We will illustrate this by creating a small GPL and refine it in two steps. Specifically, the language will define the concepts `package`, `class`, `attribute`, `function` and `expression`. In addition, a type hierarchy and mathematical operator are included. To keep it simple, we will have this interpretation of the concepts:

- A class consists of an arbitrary number of attributes and functions
- An attribute has a type as defined by a type system, and a value
- A function comprises one (zero) or more expressions
- Expressions can either be a value or defined using an operator supporting diverse operations
- Each class has a operation named `main` that starts execution of class code
- Executing a class implies that attribute values are printed to screen and functions invoked
- Invoking a function means that expressions are evaluated and printed to screen

Notice that the `main` operation invokes all functions of a class. There are three purposes with this example:

- Illustrate that domains of general concepts are supported by the LDE Platform
- Encourage iterative and incremental language development
- Show how a type system (hierarchy) can be implemented
Clearly, the LDE Platform is not limited to Domain-Specific Languages. In fact, all types of domains are supported. In this example we will focus on how a language can be built in an iterative and incremental fashion. The advantage of this approach is that language segments / aspects can be tested independently before the composite language is created. This ensures easier development and simplifies the testing process. We will anticipate events and consider the composite language’s meta model. This is found in Figure 14.14.

As can be seen in Figure 14.14, a package is composed of zero or more classes. These classes are composed of attributes and functions. Each attribute has a type as defined in the type hierarchy. A function has zero or more expressions. Notice that the classes GType, GNumber and GExpression are abstract. Each construct has been marked with a colour to indicate the primary / aspect language it belongs to. Each root construct of the respective languages is denoted with the language name: AGPL1, AGPL2 or AGPL3. We will refer to the composite language as AGPL. AGPL1 is a primary language, while AGPL2 and AGPL3 are aspect languages that define functionality for attributes and functions. Consequently, value storage and calculation are two aspects that are addressed by the aspect languages AGPL2 and AGPL3, respectively.

The primary language AGPL1 can be refined in either one or two steps to support class attributes and functions. We have earlier elaborated on how languages are woven and integrated. Here we will underline the advantages of developing languages step-wise. Constructing a language can be complicated. Complexity can be dealt with by dividing the language into segments that can be designed, analysed and optimised iteratively until the requirements are met. The segments can then be combined in an incremental fashion with continuous testing and revision (including execution of quality assurance routines). Additionally, development of one or more of the segments can be outsourced to departments and companies with expertise knowledge.
Many languages, especially General Purpose Languages, have some kind of type hierarchy. This kind of structure is used to organise and arrange entities, and relate them together. In addition, inheritance makes it possible to define one entity on the foundation of others. Consequently, operations can be defined on multiple similar entities using different kind of polymorphic mechanisms. There are many types of hierarchies. The one illustrated in the AGPL meta model is *Single-Type inclusion with strict Partitioning (STP)* [32].

We will conclude this section with an example that illustrates the general-purpose properties of the AGPL language. In short, AGPL is a language for storing of textual and numerical values, and calculating mathematical expressions. We will create a model in the language illustrating both language aspects. There are seven operations supported by the GOperator semantics: *add*, *sub*, *multi*, *div*, *power*, *max* and *min*. These could be represented as subtypes to GOperator. However, in this example we differentiate between the types of operation using the attribute *type* of GOperator. The value of this attribute determines the performed operation. Note that precedence follows natural mathematical laws. We will illustrate an alternative concrete syntax in this example. Every language construct is in bold. Notice that the provided code is an excerpt of a larger program.

```java
GPackage p1 = new GPackage( "package1" );
GClass c1 = new GClass( "class1" );
GAttribute a1 = new GAttribute( "attribute1", 4 );
GAttribute a2 = new GAttribute( "attribute2", 3.4 );
GAttribute a3 = new GAttribute( "attribute3", "Value of attribute 3" );
GFunction f1 = new GFunction( "function1" );
{  
  GExpression e1 = new GValue( 5.6 );
  f1.setExpressions( e1 );
}
GFunction f2 = new GFunction( "function2" );
{  
  GExpression e2 = new GOperator( "sub" );
  {  
    GExpression e3 = new GOperator( "sub" );
    {  
      GExpression e4 = new GValue( 10.0 );
      GExpression e5 = new GValue( 39.0 );
    }
    e3.setOperandA( e4 );
    e3.setOperandB( e5 );
    ...
  }
  e2.setOperandA( e3 );
  ...
}
f2.setExpressions( e2 );
c1.setAttributes( a1 );
...  
c1.setFunctions( f1 );
c1.setFunctions( f2 );
p1.setClasses( c1 );
```

Notes on the concrete syntax:

- An attribute’s type is inferred from the provided attribute value
- The methods used in the concrete syntax correspond to automatic generated setter and getter methods of the semantics module (references are used to generate method names). Thus, these are not explicitly declared in the abstract syntax by default. Please refer the semantics
module of AGP by consulting the source code
- Braces are used to increase readability of the code
- The keyword `new` is used to indicate instantiation of a class

Executing the program gives the printout as can be seen in Figure 14.15.

```
[RUNTIME] Method invoked: 'language.agpl3...
  attribute1:GInteger = 4
[RUNTIME] Method invoked: 'language.agpl3...
  attribute2:Gfloat = 3.4
[RUNTIME] Method invoked: 'language.agpl3...
  attribute3:GText = Value of attribute3
[RUNTIME] Method invoked: 'language.agpl3...
  function1 = 5.6
[RUNTIME] Method invoked: 'language.agpl3...
  function2 = -40.9
[RUNTIME] Method invoked: 'language.agpl3...
  function2 = 6.73842E-5
```

Figure 14.15 Executing example program in AGPL

14.6 LANGUAGE DESIGN METHODOLOGIES

An important manner of dealing with cross-cutting concerns is to design languages according to some kind of strategy or methodology. In the scope of this thesis, we will narrow this down to identification of some principles and guidelines of composite DSL design. The overall design goal is to create sound, logical and flexible atomic DSLs that can easily be used to create composite languages.

Important points in a sound language design are:

- A language should reflect concepts of a cohesive and confined problem domain
- The constructs of a language ought to be on the same abstraction level
- Cross-cutting concerns should be defined in distinct languages, if possible
- An aspect language should support refinement of multiple primary languages

**A language should reflect concepts of a cohesive and confined problem domain**

A DSL is, as the name suggests, domain-specific. In the context of composite DSLs it is especially important to be aware of the problem domain represented by the DSL. Firstly, the concepts reflected by the DSL should be closely related and cohesive. Secondly, there should not be more constructs available in a DSL that is strictly required to model the desired concerns. Additional and optional constructs can easily clutter the language and make its role less clear. As a consequence, this makes a DSL more difficult to refine because more dependencies have to be maintained in order to preserve the language’s integrity. It is desirable to design languages as self-contained units that can easily constitute a combined logical whole.

**The constructs of a language ought to be on the same abstraction level**

Detail is often dependent on the abstraction level chosen. Clearly, a low abstraction level illuminates more details than a higher abstraction level. Mixing constructs reflecting concepts on different abstraction levels makes a language less distinct and less intuitive
with regard to composition. For the most part, this applies to atomic languages since weaving of languages ultimately will result in constructs on several different abstraction levels. Note that modelling according to language views and meta components are means of dealing with multiple abstraction levels.

**Cross-cutting concerns should be defined in distinct languages, if possible**

There are many types of cross-cutting concerns. Some concerns may be concretised and abstracted away in a distinct language or language segment. For instance, logging is a cross-cutting concern that potentially can be represented in a separate language segment.

Figure 14.16 Separation of cross-cutting concern L

Consider the meta model in Figure 14.16. It consists of two parts or meta model segments, where instances of $L_1$, $L_2$ and $L_3$ can be used to model logging functionality. The purpose of such logging functionality is to log status information for the functions $FA$, $FB$ and $FC$. Notice that the two meta model segments are connected merely by the association functions. Clearly, the language segments are very loosely coupled. Semantics wise, the constructs expressing logging functionality can access data of the functions using generic reflection / introspection. The main point of this example is to illustrate how an initial cross-cutting concern can be separated in a distinct segment of a language. This is desirable because it simplifies languages and increases reusability.

**An aspect language should support refinement of multiple primary languages**

One of the key points behind the concepts discussed in this thesis is a high degree of reusability both with respect to syntax and semantics. However, to increase a language’s usability it should be designed wisely. Clearly, proprietary solutions are not desirable. A language ought to be as generic as possible, but still, without compromising the problem-specific nature of the language. This is achievable using a thorough problem domain analysis and by following best-practices and methodologies.

The consequence of not following the principles discussed here can be languages that reflect interwoven problem domains and concepts on several abstraction levels. This is not desirable since the complexity of language refinement is increased. There is, however, no practical limitation to the structure of a language with respect to the LDE Platform. In fact, the only critical requirements are that a language conforms to the Ecore meta-meta model and that its semantics corresponds closely to the first-class entities of the meta model.

More work has to be done to investigate explicit architectures, design patterns and development strategies that will support composite language construction. There is literature available which describes more general language design and implementation patterns that should be applied. A discussion of traditional DSL design methodologies and patterns is found in [23].
Software evolution is a phenomenon that occurs on several levels. This results in incompatibility issues between different components or parts of a system. In the context of topics discussed in this thesis, it is important to consider two types of software evolution: evolution of a language’s meta model and evolution of its models. As noted previously in chapter Software evolution, a general concept for describing asynchronous evolution between two entities is referred to as co-evolution. In the case of DSLs, co-evolution is used to describe the asynchronous evolution between meta models and models.

Objective: Evolving languages
Contribution: The concept of composite DSLs, The LDE Platform, Designing a composite DSL (Model Transformer, ATL Preprocessor)

15.1 RECYCLING OF MODELS

We have earlier seen how refinement of a language results in incompatibility issues because existing models are not compatible with the new language version. Naturally, these issues appear because the conformity relation between the new meta model and the existing models are not fulfilled. The dilemma is clear; a language can be updated to be more expressive and accurate, however, at the same time the existing models will not conform to the language’s new meta model. An immediate solution to this is to keep old language versions to avoid discarding old models. In the long run, this is not a practical way to deal with software evolution since there will eventually be many coexisting language versions. Clearly, this makes it difficult and confusing to relate to the language and its models.

As previously illustrated, a convenient manner of dealing with evolving languages is to recycle models. This is achieved by performing model transformations, as exemplified using the Model Transformer and ATL Preprocessor. Specifically, a model conforming to an old version of a language is merged with one or more models conforming to the aspect languages used to elaborate the language. The result of the model transformations is a new model that conforms to the new language’s meta model. Consequently, existing models of a language can still be used even after a refinement of the language. An example of model recycling is found in the section Scenario 2 - Using a third-party language of chapter Designing a composite DSL.

15.2 MODELLING ACCORDING TO LANGUAGE VIEWS

A composite DSL can be extended repeatedly to capture more details. Naturally, such extensions increase the size of the meta model. This makes it more difficult to get an overview of the language. Furthermore, it can be challenging to know what language constructs to use when modelling a given concern. A suggested approach of dealing with this issue is to create models according to language views, as inspired by [11]. As the name suggests, a language may be observed using several views or perspectives. Each view corresponds to a certain language aspect or segment. Thus, selecting a view makes it possible to reason about a separate aspect or part of a language. This can be done independently of how the aspect or part is related to the remaining language.

Another term used for modelling according to different views is Domain-Specific Multimodelling [33]. A challenge within Domain-Specific Multimodelling is to address
the *Coordination Problem*. In short, this problem states the difficulties in integrating the different views. We will see how the LDE Platform addresses this issue.

The LDE Platform supports modelling according to language views. Each language view corresponds to a language used to build a composite DSL. For instance, recall that the previously introduced Car language was refined using an Engine language. Clearly, the composite Car v2.0 language comprises general Car constructs and Engine constructs. As might be expected, this composite DSL provides the developer with two language views; one view for modelling of the general car and one explicit view for modelling of the car’s engine. Thus, it is possible to model the car separately from its engine and vice versa. There are some important points to this feature, specifically:

- A composite DSL model can be created by several people
- Models corresponding to different views can be validated separately
- Modelling is simplified, thus, language overview is preserved

Each point is discussed below. Note that the entire language can also be viewed according to a *general view*.

**A composite DSL model can be created by several people**
Expertise and knowledge are distributed among people. Modelling according to language views makes it possible for several developers to cooperate in creating a model / program. Clearly, developers who are best suited to model certain aspects can provide their expertise directly by creating an optimal aspect model. This principle applies both to local and decentralised development. Furthermore, each developer only has to relate to language constructs that are directly related to her / his assigned aspect or field of expertise. This is closely related to Aspect-Oriented Modelling and an Information Design methodology known as *Polyscopic Modelling* [34].

**Models corresponding to different views can be validated separately**
An advantage modelling according to different language views is that each model corresponding to a view can be validated separately. Exemplified with the Car v2.0 language, this means that a developer with expertise knowledge on engines can model a given engine and validate the model against the engine-specific part of the language.

**Modelling is simplified, thus, language overview is preserved**
Language views are a mechanism that ensures separation of concerns. Instead of having the complete overview, a software engineer can focus on the part of a language that is within her / his area of expertise. In practice, this implies that a developer creates a model that conforms to one of the aspect languages’ meta model.

The installation package of a language contains the meta models for all languages (primary and aspect languages) constituting the composite meta model. These meta models correspond directly to the native language views available for the composite DSL. That is, developers can create models conforming to each of the aspect meta models. A model conforming to the composite DSL meta model can then be created from these models using model transformations. A simple example will illustrate this concept further.
The meta model of a composite DSL, as constructed using three languages A, B and C, is shown in Figure 15.1. It can be viewed according to several perspectives, or views. In principle, a view can be created arbitrary. However, there are some views that make more sense than others. In this case, three views are chosen according to the meta model segments used to build the composite DSL. Naturally, these views reflect the three main aspects addressed by the DSL: A, B and C. In addition comes the General View (ABC) which covers the entire composite meta model. A model can be created in any of the four language views. Notice that simplified class diagrams are used in the figures since types of relations, multiplicities and other properties are irrelevant for the sake of this example. The colours identify refined constructs. That is, A2 is refined by the B language; A3 is refined by the C language.

Four meta models are available in the installation package of the example composite DSL. These correspond to the views identified. Please refer Figure 15.2 for an overview of these meta models.

Consequently, developers can easily create models that describe the different aspects of the language by instantiating each of the respective meta models. For instance, a developer with expertise knowledge of the aspect C only has to relate to the constructs A3, C1, C2 and C3. The other constructs of the composite DSL are irrelevant. Subsequently, the different models can be merged to create a model conforming to the composite meta model, ABC. Notice that the meta models for languages B and C have their root constructs renamed to match the constructs they refine.

Composite language views

It is possible to create composite language views. This can be achieved using the Weaver. For instance, the meta model segments addressing aspects A and B can be identified as one composite language view. Composite language views are possible because of the
partially associative Weaver. This property implies that the aspect meta models can be woven together according to different configurations and still give the same composite meta model. Obviously, there are weaving configurations that do not give the correct composite meta model. What configurations that describe a valid weaving sequence depends on the structure of the aspect meta models. In most cases, a fixed sequential order has to be fulfilled. Returning to our example, the composite meta model ABC can be created using two weaving configurations: by weaving together A, B and C sequentially, or A and B and then C. In the latter case the meta model AB represents a composite language view. A model that conforms to this view / meta model can be merged together with a model of C to create the final composite DSL model. Valid weaving configurations can be identified using expressions as exemplified below.

\[ A + B + C = (A + B) + C \]

The constructed custom view AB for the example DSL is shown in Figure 15.3.

![Figure 15.3 Custom view AB and native view C](image)

Notice that the Model Transformer and ATL Preprocessor will always be compatible with meta model segments that are parts of a legal weaving configuration. That is, for our example, meta models A, B, C and AB are supported by the tools since models conforming to all these meta models are legal building blocks that can be used to create the final composite model. Note that composite language views increase the flexibility of modelling. Related aspects can be observed and modelled simultaneously by creating a custom view. This will not compromise the overview since only relevant aspects are woven together.

As can be seen in Figure 15.4, five different views are available for the composite DSL including the custom made AB view (created using the Weaver). These views correspond directly to meta model segments. Two models \(ab\) and \(c\) have been created according to the AB view and C view respectively. Together these views cover the entire composite meta model. Consequently, a model conforming to the composite meta model can be created from the \(ab\) and \(c\) models. That is, convergence of the different views is required. This is achieved using the Model Transformer. Refer Figure 15.4.
The result of the model transformations is the model named `a1_transformed.xmi`. (\textit{A1} is the root construct of the composite meta model.) `a1_transformed.xmi` conforms to the composite meta model, here represented as `ABC.ecore`. An overview of the input and output models of the transformation process is shown in Figure 15.5.

Providing GUI-based tools for creating language views would increase usability of the LDE Platform. This would make it easy to specify what meta model segments that should constitute a view. Legal weaving configurations could also be displayed in an intuitive manner. A mock-up of such a tool is shown in Figure 15.6.
As can be seen in Figure 15.6, the different aspects / languages / meta model segments of a composite DSL are shown in the left box. Dependencies of a given language segment is shown at the right of the arrow. For instance, A depends on B. This implies that one or more constructs of A has to be refined by B in order for A to yield a consistent view. Consequently, creating a language view using the A aspect requires the B segment to be added as well to avoid creating inconsistencies.

A valid composite language view consisting of the aspects A, B, C and D has been created. Refer the right box. All dependencies of these segments are fulfilled. Note, however, that E has a dependency to F which has not been added to the language view. Thus, the E aspect is not included in the language view (it is required by C). Briefly speaking, this is possible since F may refine one or more constructs of E that are not used to refine constructs of C. Moreover, these constructs may be optional. As might be expected, modelling according to the custom language view equals creating a model of a composite meta model comprising the segments A, B, C, D and E (a version not refined by F). This meta model has to be created by the Weaver. Thus, clicking the OK button could have initiated weaving of the needed composite meta model.

Let us assume that F refines constructs in both C and E. That is, both C and E depend on F. In such case, F must be added to the language view to avoid creating inconsistencies. Refer Figure 15.7 for an illustration of the two discussed scenarios. In the left diagram there are no direct dependencies between the language C and F. On the contrary, in the right diagram F is used to elaborate the construct C3. Consequently, F must be included to achieve a valid view. (The languages A, B, and D are excluded from the figure.)
Additional information like weaving sequence can be presented in GUI. This is illustrated for the C segment in Figure 15.6. As can be seen, E and F have been woven together yielding an updated E version. This new E version has been woven together with C and D to create a new C version. Future work may deal with automatic detection of construct dependencies. The purpose of such detection is to provide information on dependencies directly as shown in the editor. Hence, this will simplify the construction process of legal language views. This is related to the Coordination Method of Domain-Specific Multimodelling [33].

Note that the concepts of model recycling and modelling according to language views only work if the composite DSL meta model has been created using non-destructive weaving. That is, the weaving process does not remove existing constructs, rename constructs or otherwise alter the meta model properties or structure other than what is present in the aspect meta models. More work has to be carried out to investigate how destructive meta model weaving can be supported. Note that the concepts of model recycling and language views are applicable to general meta models and models.
Decentralised development

The amount of information available in our fast-paced world increases in an exponential fashion. In addition, economic assertion of every aspect of business results in high competition. As a consequence of this, specialisation has become more and more important. Such specialisation can be observed within most genres of industry, especially within the manufacturing business. However, the software industry is not an exception from this trend. Specialisation is directly related to decentralisation of knowledge and expertise. Tasks that earlier could be carried out in-house are outsourced to other departments or companies. Naturally, this also impacts modelling and programming languages design to some degree. Community-based language development founded on a contribution model has been used with success for languages like UML and Java. In this chapter we will elaborate on these issues.

Objective: Decentralised design and development of languages, reuse of languages
Contribution: The concept of composite DSLs, The LDE Platform, Designing a composite DSL
(Extractor, Injector, Weaver, Runtime Environment)

16.1 THE IMPORTANCE OF DECENTRALISED DEVELOPMENT

Domain-Specific Languages reflect domain-specific concepts in an accurate manner. Clearly, need of expertise knowledge is crucial when designing a language for a certain kind of tasks. Without such knowledge, it is not possible to create a sound and expressive language. It may be argued that DSLs are small languages. And, thus, a small group of people may possess the necessary expertise. However, this is not entirely correct. In fact, what determines the need of expertise when designing a language is the abstraction level at which this language operates. Specifically, what makes things interesting is when a language provides constructs on different abstraction levels. As an example, one developer can have the necessary knowledge to create a language for some kind of problem domain at a relatively high abstraction level. However, if one or more of this language's constructs are elaborated in more detail, the initial developer may not possess the necessary knowledge. For instance, we earlier used an example language named Car. This language had a construct named Engine which was refined using an explicit language for modelling of engines. It is not obvious that the developer of the Car language knows the details on how engines work. On the contrary, there may be a developer somewhere in the world with the exact knowledge of this. A more wise choice would be to have this developer create the Engine language. As a consequence, the second version of the Car language would yield a more correct approach to modelling of cars and their engines. This observation promotes decentralised language development. One objective of this thesis is to investigate a possible approach of supporting such decentralised development. This can be achieved using the proposed LDE Platform.
16.2 A DECENTRALISED LANGUAGE DEVELOPMENT PROCESS

A decentralised language development model is illustrated in Figure 16.1.

As can be seen in Figure 16.1, a language titled A is created as a common effort. Specifically, a composite language is created by weaving a primary language with two compatible aspect languages. These aspect languages can be defined in another department or company. Clearly, more aspect languages can be used. Essential tools supporting decentralised language development are the Extractor, Injector and Weaver. Furthermore, the Runtime Environment supports execution of the composite language’s models. Decentralised development has several forms. Here we saw how third-party languages are developed based on a set of requirements and formal interfaces. Another variant of decentralised development exists in the context of online communities and voluntary work. A thorough discussion of this is found on the accompanying CD.

16.3 ARBITRARY LANGUAGE INTEGRATION

We have earlier described the local Language Repository. Basically, this is the collection of all installed languages of the LDE Platform. That is, all the available languages that can be combined to create new languages. This concept can be taken further by introducing a remote Language Repository. A remote Language Repository is a remote collection of languages that can be downloaded on a need-to-have basis using a technology like Concurrent Versions System (CVS) in a similar fashion that is used for plug-in installation in Eclipse. This concept can be further elaborated by developing functionality for accessing and searching the available languages from within the Integrated Development Environment (in this case exemplified by Eclipse).

We will illustrate how community-based language development can be supported by the LDE Platform from a technical point of view. A possible language development scenario using a remote Language Repository and a third-party language could follow the steps below. The Car language is used in this example. The third-party aspect language used in this scenario is developed independently of the primary language. This is on the contrary to the earlier Simulator scenario where a language was designed on request especially to be compatible with the primary language.

1. Let us assume that the Engine concept of the Car language needs to be refined to support more details. Thus, the remote Language Repository is searched for languages that can be used to model engines. A list comprising five relevant languages is returned.
A language named *CarEngine* is selected from this list based on the associated meta data and language description. It is downloaded into the local Language Repository of the LDE Platform. For the sake of clarity, we assume that CarEngine is equal to the earlier introduced Engine language except for the root concept which is named *CarEngine*. Please refer the section *First example of a composite DSL* in chapter *The concept of composite DSLs* for details.

2. Namespace issues and concept renaming are resolved, as described in the sub section *The LDE Tool Suite ➔ Weaver* of chapter *The LDE Platform*. Moreover, the language CarEngine is incorporated into the Car language using the Weaver. Experience has indicated that, in most cases, the abstract syntaxes of two related languages can be merged by concept renaming and weaving. If necessary, it is also possible to create a set of *additional constructs* in the primary language that provides a bridge between the two languages. In order to unify the two meta models in this case, the *CarEngine* construct can be renamed to *Engine* in a copy of the CarEngine meta model (to avoid alter the native CarEngine meta model). Notice that renaming a meta model concept does not affect the corresponding semantics class which is denoted by the *semantics* class attribute.

3. CarEngine has an unknown, arbitrary semantics. Furthermore, it does not implement any of the API methods as imposed by the Car language. Obviously, it is not compatible with the Car language. In essence, there are three approaches in order to achieve such compatibility, as discussed in the next paragraphs.

**CarEngine.CarEngine implements Car.IEngine**

We have earlier illustrated how third-party language semantics can be developed and linked to the semantics of a primary language. That is, the third-party language has to implement a set of public API methods. The same approach can be used here. Hence, the most intuitive way is to let the *CarEngine* class of the CarEngine’s semantics module implement the *IEngine* interface of the Car language. This process is earlier described in the section *Scenario 3 - Defining a third-party language* of chapter *Designing a composite DSL*. An advantage with this approach is that no in-depth insight of the CarEngine semantics is needed in order to integrate the languages’ semantics. This is because the methods from the *IEngine* interface only refer to concepts of the Car language. Moreover, these methods can invoke the available API methods of the *CarEngine* class directly from within the class itself.

**Car.Engine implements CarEngine.ICarEngine**

An alternative approach is to let the *Engine* class of the Car language implement the *ICarEngine* interface. This makes it possible to define a custom engine semantics using the semantics of the CarEngine concepts directly. That is, the semantics of the actual root class, here *CarEngine*, is discarded to be replaced by a new semantics. Clearly, this approach requires a thorough insight into the semantics of the CarEngine language.

**Subtyping of CarEngine.CarEngine and implementing Car.IEngine**

A third option is to subtype the *CarEngine* class of CarEngine and at the same time implement the *IEngine* interface. The subclass can be stored within the context of the Car semantics module. Subtyping makes it possible to integrate the semantics of the two languages without altering the CarEngine language in any way. It can be argued that this approach is more correct with respect to preserving integrity and third-party software
encapsulation. Furthermore, it is possible to use the native downloaded CarEngine package repeatedly in several composite DSLs since the semantics is not edited. An interesting property of subtyping is that properties of the supertype can be accessed in two manners: either by invoking inherited methods or by accessing inherited supertype attributes and references directly. Clearly, invoking supertype methods from within the subtype do not require in-depth insight of the supertype’s logic (semantics). However, if necessary, it is possible to override the native semantics and provide a new. Hence, subtyping is an easy and efficient way of integrating the semantics modules of languages, and promotes the use of an established object-oriented mechanism for specialisation. This latter approach would thus be suggested as the default way of integrating the semantics of arbitrary DSLs.

One concept of the Car language, named Engine, is here refined using the CarEngine language. It is quite possible that several constructs are refined at the same time in a similar manner. For instance, the Car language could initially feature two separate engine-specific constructs that have to be refined by corresponding concepts of the CarEngine language. Clearly, several classes of a language’s semantics module can be subtyped in a similar fashion as described if necessary.

The Runtime Environment supports all alternatives of semantics integration as described above. Be aware that all communication between the semantics modules of Car and CarEngine in the first and third alternative is performed using the methods of IEngine. Naturally, this is because the Engine concept of Car is elaborated using the CarEngine language. Additional methods can be added to IEngine as they are required to achieve an adequate communication.

Some code examples are provided due to the importance of the semantics integration process. Subtyping is used as the preferred method. Notice the values of package declarations and import statements of the two classes.

<table>
<thead>
<tr>
<th>Table 16.1 - Excerpt from the CarEngine class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. package language.carengine.classes;</td>
</tr>
<tr>
<td>2. import language.carengine.interfaces.ICylinder;</td>
</tr>
<tr>
<td>3. import language.carengine.interfaces.IType;</td>
</tr>
<tr>
<td>4. public class CarEngine implements ICarEngine</td>
</tr>
<tr>
<td>5. {</td>
</tr>
<tr>
<td>// Methods from ICarEngine</td>
</tr>
<tr>
<td>6.     public void setCylinders( ICylinder cylinder ) {}</td>
</tr>
<tr>
<td>7.     public Iterable&lt;ICylinder&gt; getCylinders() { return null; }</td>
</tr>
<tr>
<td>8. }</td>
</tr>
</tbody>
</table>

Eight methods are declared in the ICarEngine interface. These represent the public API of the CarEngine class. An excerpt of CarEngine is found in Table 16.1. (Methods are given an empty definition for the sake of clarity.)
Six methods are declared in the `IEngine` interface. These methods are used to communicate between the semantics modules of Car and CarEngine. This communication comprises information on horse power, torque, engine volume and valves count. Empty definitions are used for two of the `IEngine` methods in Table 16.2. As described, there are two approaches of utilising the inherited properties of `CarEngine`:

- The implemented methods from `IEngine` can invoke inherited methods from `CarEngine` to acquire necessary information
- Inherited attributes and references are used directly. This includes the process of overriding one or more inherited methods from `CarEngine` to provide a custom `CarEngine` semantics

Examples of semantics integration between the languages are found in Table 16.3 and Table 16.4. Specialisation of the `CarEngine` semantics can be seen in Table 16.5.
Table 16.5 - Example 3: overriding `getCylinders()` to create a custom semantics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>public class CarEngineSub extends CarEngine implements IEngine</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>public Iterable&lt;ICylinder&gt; getCylinders()</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>{</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>// Some additional, new code</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>...</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>return cylinders;</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>}</code></td>
</tr>
<tr>
<td>9.</td>
<td><code>...</code></td>
</tr>
<tr>
<td>10.</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Obviously, these examples are trivial, but they sum up how semantics integration of arbitrary languages is performed. Note that the `semantics` attribute of the meta model construct `Engine` has to be updated to point to its new semantics class. In this case, the correct value of this attribute is: `language.car.classes.CarEngineSub`

The Runtime Environment takes care of the appropriate instantiations and resolves method invocations to ensure that the correct method definitions are used.
In this chapter we will investigate some of the most essential properties of frameworks, DSLs and composite DSLs. Central goals are to get an overview of advantages and disadvantages using the diverse approaches of software development seen in the light of this thesis’ objectives, and to put DSL development in the right context.

**Objective:** Evaluation of frameworks and DSLs  
**Contribution:** The Ecommerce framework, The concept of composite DSLs

### 17.1 Frameworks versus DSLs

There are some distinct differences between development using frameworks and DSLs. The most evident difference is the abstraction levels at which the two approaches operate. A framework is in most cases created using a GPL. That is, the framework concepts are defined using general constructs of the GPL meta model. Software development using a framework implies creating an application model in the same GPL and linking this model with the model constituting the framework. A conceptual overview of this is found in Figure 17.1. F model and A model are abbreviations for framework model and application model.

![Figure 17.1 Overview of development using a framework](image)

On the contrary, a DSL operates on the meta model level. Consequently, development using a DSL basically means creating an application model conforming to the DSL meta model. Please refer Figure 17.2.

![Figure 17.2 Overview of development using a DSL](image)

As can be observed, a framework captures the problem domain knowledge in a model of a GPL. Thus, the interpretation of the framework concepts is expressed using GPL semantics. A DSL, on the other hand, captures problem domain knowledge in its meta model and corresponding domain-specific semantics. Some of the advantages using DSLs and frameworks, related to the topics of this thesis, are identified below.

**Advantages using DSLs:**
- DSLs are more expressive with regard to their problem domain
- It is not possible to create arbitrary general purpose models
- Models can be validated against problem domain-specific constraints / knowledge
Advantages using frameworks:

- Concepts of the framework can be extended and redefined using subtyping
- Common familiar GPLs are used to create frameworks
- Frameworks may utilise proved modelling concepts

**DSLs are more expressive with regard to their problem domain**

DSLs have domain-specific language constructs. These constructs are created with one superior goal: they should reflect problem domain concepts as accurately as possible. As a consequence, the user of a DSL is equipped with a palette of accurate building blocks that can be used to model their concerns. In essence, a DSL only contains constructs that are closely related to the problem domain. Furthermore, these constructs are integrated and related in a manner that resembles relations present in the problem domain. This increases the language’s usability. Clearly, the semantics of DSLs is custom-made to support computation of specific concerns.

**It is not possible to create arbitrary general purpose models**

One premise for achieving high software quality is that models are created accurately and correctly. An essential property of a DSL is the limited types of models that can be created in the language. Specifically, valid models can only use domain-specific constructs. These constructs have to be combined according to the meta relations present in the language’s meta model. As a consequence, it is not possible to represent concerns outside of the target problem domain. From another point of view, the restrictions in a DSL increase the chances of creating sound and correct models.

**Models can be validated against problem domain-specific constraints / knowledge**

A clear difference between frameworks and DSLs is the degree of formality used to express domain-specific concepts. Frameworks use generic constructs to provide the necessary abstractions. On the contrary, DSLs provide explicit language constructs for the same abstractions. Thus, a DSL model can be validated against its meta model to verify the correctness of problem domain-specific concerns. (This includes verification of static semantics that imposes additional requirements on the model.) Such validation is not feasible using frameworks. The reason for this is that an application model is verified against the general syntax and semantics of the GPL. Consequently, it is not possible to verify if problem-domain constraints are fulfilled. The only way to ensure this is to develop additional validation mechanisms using the generic constructs of the GPL.

We will elaborate on the identified advantages of frameworks. These advantages can also be observed as shortcomings of traditional DSLs.

**Concepts of the framework can be extended and redefined using subtyping**

Many frameworks are built using an object-oriented language. This promotes the use of subtyping to extend and elaborate concepts to meet requirements that are not fulfilled by the default framework concepts. A traditional DSL does not have functionality for refinement of the language concepts. Language refinement must be performed explicitly by the language’s developer. Clearly, using a framework is advantageous in contexts where a high degree of flexibility and customisation is required.

**Common familiar GPLs are used to create frameworks**

Most developers have some knowledge of the major General Purpose Languages used to write frameworks. Thus, from a technical point of view, this speeds up the framework
learning process. On the other hand, a DSL provides the user with new and unfamiliar syntax. Using a DSL efficiently may thus take some time. Naturally, this depends on the language complexity and size. (It may be argued that DSLs are often small in comparison to GPLs, and consequently, take less time to master.)

**Frameworks may utilise proved modelling concepts**

There are many concepts and methodologies available that support a developer in the model creation process. Examples of popular modelling concepts are state machines, components and generics / templates. These concepts can be applied as required in a framework. As a result, the framework can accommodate all types of problem domains in an efficient and intuitive manner. On the contrary, a DSL is expressed using meta modelling. At present, most meta modelling deals with flat class diagrams. This may potentially result in implementation of proprietary solutions in order to solve certain modelling tasks. A concrete example of this is found in the section *Using third-party semantics* in chapter *Aspects of composite DSLs*. Recall that the Webshop language’s meta model provides a class named `Function`. An object of this class can relate to an arbitrary number of other `Function` objects using associations, as denoted *navigations* in the meta model. This gives a mechanism for navigating between functions. However, large models may seem complex because of the many navigation routes between functions. It would be better to provide a separate state machine for handling this navigation. This state machine could either be integrated into the existing meta model or provided as an auxiliary model.

Some work has been done on applying traditional model concepts to the genre of meta modelling. We will later discuss how components can be realised in the context of meta modelling.

### 17.2 TRADITIONAL DSLs VERSUS COMPOSITE DSLs

A composite DSL is built using a set of aspect DSLs. There are two ways to use a composite DSL. We will refer to these approaches as *static* and *dynamic* modelling. These terms must not be confused with statically and dynamically typed / scoped languages.

**Static modelling**

Traditional DSLs can be referred to as static languages in the sense that they can not be customised to adapt to new requirements. Thus, only pre-factoried constructs can be used in the modelling process, referred to as *static modelling*. In fact, if a DSL does not provide the necessary constructs to model a certain concern, it simply can not be used.

**Dynamic modelling**

Composite DSLs build on a different foundation than traditional DSLs. Consequently, they are compatible with *dynamic modelling*. This term basically means that the language is adapted dynamically to support more details and altered problem domains. We have earlier illustrated the mechanisms that support such customisation.

We will relate composite DSLs to the discussion on frameworks and DSLs. As noted, a disadvantage of DSLs is the lack of flexibility. A DSL may provide some basic options for customisation. However, the DSL is still closely related to the problem domain this language was initially created for. As we have seen, a composite DSL is not limited to this initial problem domain. The principles behind composite DSLs support reuse of
languages. An advantage of reuse is that developers can stick with languages they are familiar with. Then, if a DSL does not meet the requirements, it can be extended instead of discarded. Clearly, the process of learning a refined language is easier than learning a brand new language.

### 17.3 Differences in Development Methodologies

Obviously, the engineering processes using a framework, DSL or composite DSL are quite different. A traditional DSL is the most rigid and static entity of the three alternatives. Frameworks and composite DSLs have some common properties with regard to how these tools can be extended and customised. Using a framework implies instantiating framework concepts to model problem domain-specific concerns. A framework is in most cases acquired as a pre-defined unit. Extensions (subtypes of framework classes, etcetera) to the framework are often kept privately by the developers whom create the extensions. This means that other developers utilising the framework may have to perform the same extensions to model a given kind of tasks. On the contrary, extensions to a composite DSL are kept as integral parts of the language. Thus, new language versions can easily be distributed for several people to use. Furthermore, a refinement to a composite DSL will in theory always give a sound new language version, as long as the aspect language used is sound and correct. Refer sub section Core aspects ➤ Semantic coherence of chapter Aspects of composite DSLs.

Writing extensions to a framework can introduce errors and inconsistencies. Additionally, it is not possible to validate in a generic way if a created application model is correct and sound with regard to problem-domain specific knowledge incorporated into the framework. Thus, software engineering using frameworks is more error-prone than using DSLs and composite DSLs.

### 17.4 Core Properties

A partial general requirements analysis for DSLs is presented in [30]. This analysis includes identification of ten core requirements. We will briefly list each requirement and apply them on the traditional conceptions of frameworks, DSLs and composite DSLs. (Note that the requirements are slightly adjusted.) The goal is to create an overview of similarities and differences which can be used to identify strengths and weaknesses of the different approaches. In Table 17.1, the term entity will be used as a common descriptor for frameworks, DSLs and composite DSLs.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformity</td>
<td>Correspondence between the entity concepts and important problem domain concepts.</td>
</tr>
<tr>
<td>Orthogonality</td>
<td>The ability to express a one-to-one correspondence between an entity concept and a problem domain concept.</td>
</tr>
<tr>
<td>Supportability</td>
<td>Indicates the ability to provide tool support for development of the entity, and modelling management (creation and debugging of models / programs, etcetera).</td>
</tr>
<tr>
<td>Integrability</td>
<td>Describes how easily the entity and its tools can be integrated with other entities and tools.</td>
</tr>
<tr>
<td>Extensibility</td>
<td>Illustrates how easily the entity and its tools can be extended to support additional concepts / constructs.</td>
</tr>
<tr>
<td>Longevity</td>
<td>A developed entity should be usable (and used) for a sufficient period of time to justify the development effort and costs of the entity (and show a payoff for all stakeholders). Furthermore, this is critical to ensure development of supporting tools.</td>
</tr>
<tr>
<td><strong>Simplicity</strong></td>
<td>Describes if an entity is created and presented in a straightforward and simple manner, which is desirable.</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Quality</strong></td>
<td>Quality, in this regard, describes whether the entity has internal concepts to ensure that quality aspects like security and reliability are addressed.</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Illustrates the degree of (internal) support for handling large entity models.</td>
</tr>
<tr>
<td><strong>Usability</strong></td>
<td>Describes a set of sub requirements like accessibility and understandability. In short, how usable the entity is.</td>
</tr>
</tbody>
</table>

**Conformity**

Frameworks, DSLs and composite DSLs can all be said to have a high degree of conformity. In frameworks, classes are traditionally used to reflect problem domain concepts, while class hierarchies are used to represent problem domain structure. DSLs and composite DSLs provide first-class constructs that correspond to problem domain concepts. Problem domain structures are captured using meta relations.

**Orthogonality**

All approaches are orthogonal by nature. Traditionally, a framework class is used to identify a single problem domain concept. In language design, a first-class construct is preferably related to one problem domain concept.

**Supportability**

Most frameworks are created using a GPL. A clear advantage using GPLs is the vast number of tools available. Proprietary DSL development, on the other hand, is not compatible with these tools. Consequently, new tools have to be created when needed. Composite DSLs development, as described in this thesis, supports reuse of tools since languages have to comply with a set of fundamental technical requirements. These requirements (and a set of tools) are embodied by the LDE Platform. Moreover, existing tools compatible with Ecore, Java, Kompose and ATL can be used as well. Notice that the tools referred includes both tools for framework / language development and modelling / programming.

**Integrability**

Frameworks can in most cases be integrated with other frameworks or libraries. This requires the frameworks / libraries to be created in the same GPL. Or, eventually, technologies for bridging can be used to combine entities defined on different language platforms. A traditional DSL is more rigid in this respect and integrating a DSL with other languages and frameworks can prove to be more difficult. Clearly, this all depends on how proprietary the DSL is and if common technologies are used as part of the DSL foundation. Integration of composite DSLs with frameworks and other languages can more easily be achieved since the LDE Platform is built using standard technologies. This is especially true when it comes to integration of several composite DSLs. Furthermore, frameworks and libraries can easily be used to define a composite DSL’s semantics.

Integration with tools can be argued in the same manner; common technologies and platforms increase the integrability.

**Extensibility**

We will differentiate between two types of extension: extensions made by the framework / language developer and extensions performed by the user of a framework / language.

Since frameworks are built using general languages, they can easily be extended by the
framework developers to meet new requirements. In most cases, extensions can be added to a framework without inducing co-evolution issues. In short, users of a framework do not necessarily need to update their instantiation code if a new framework version is installed. Frameworks also support custom user extensions, for instance, by subtyping framework classes.

A DSL, on the other hand, can be more challenging to update since the constructs of the language are, in comparison to framework concepts, naturally more tangled (due to the higher abstraction level). Extensions to a language also introduce severe co-evolution issues between different versions of the language and its models. Usually, a DSL can not be extended by its users.

Composite DSLs support both types of extensions. As is the case with DSLs, constructs in a composite DSL are tangled. The tools of the LDE Platform are designed to preserve all dependencies, thus, simplifying language refinement. Moreover, user-driven extensions are promoted. Software engineers / language users can perform the extensions they need without depending on new language versions from the language developers. This can also be seen as ad-hoc extensions to a language. Co-evolution issues are dealt with using model transformations.

**Longevity**

Longevity can be considered in the light of integrability and extensibility. Consequently, frameworks and composite DSLs are likely to serve their purposes for a longer period of time, with respect to traditional DSLs, because of their more flexible nature. Changing problem domains and requirements can be reflected by the framework or composite DSL by performing the necessary customisations or extensions. This includes user-performed updates and alterations.

**Simplicity**

A framework can be difficult to comprehend and relate to. Knowing what concepts to instantiate and how to fulfil the necessary dependencies are not straightforward tasks. It is also easy to introduce coding errors. In most cases, these issues have to be addressed by proper documentation. On the contrary, a DSL can be easier to manage because no instantiation code is needed. Instead, first-class constructs are combined according to the meta model and static semantics in order to create the desired model. In fact, a properly designed DSL will constrain how constructs can be combined. Thus, errors are avoided. The same argumentation applies to composite DSLs. However, composite DSLs can be difficult to manage due to its evolving nature and mixed abstraction levels. Hence, an important feature of these languages is the ability to model according to language views.

**Quality**

Both frameworks and languages can potentially incorporate concepts to increase computational quality. Whether or not such functionality is required depends on the type of language and reflected problem domain. Notice that quality in this regard includes addressing cross-cutting concerns.

**Scalability**

There exist diverse tools for modelling and programming. These can be applied when using / instantiating frameworks. There is, however, no explicit functionality within frameworks for handling large instantiation code. The same applies for large models in
DSLs and composite DSLs. As mentioned, model transformations support viewing of a composite DSL according to several perspectives. This simplifies creating large models and keeping the overview.

**Usability**

Usability can be analysed according to several criteria. According to the arguments presented under *Simplicity*, DSLs and composite DSLs can be considered to have a higher degree of usability than frameworks. However, proprietary DSLs can be difficult to integrate with other technologies, and they can also have impaired longevity due to their rigid nature. In the long term, this reduces the usability. On the other hand, frameworks can easily be integrated with other technologies and updated as needed. Composite DSLs are flexible, but language evolution can induce difficulties with respect to retaining overview. Consequently, usability depends on the task and concerns that need to be addressed. In other words, frameworks and languages should be evaluated in the light of their presumed usage scenarios.

We will sum up these results using quantitative weighting, as can be seen in Figure 17.3. This is included merely to present a more visual overview of similarities and differences in core properties. The quantisation is coarse and based on our general conceptions of frameworks, DSLs and composite DSLs. Another distribution can be argued for. Specifically, a thorough empirical analysis of existing frameworks and languages could possibly yield a slightly different result. However, this is out of the scope of this thesis.

![Figure 17.3 Evaluation of approaches](image)

Notice that proprietary DSLs are used as reference for the diagram; EDSLs are not included.

154
18 Meta components

The main points of this thesis have been design and development of composite DSLs. In the light of the proposed mechanisms of weaving languages together it is interesting to investigate how components can be realised within the discipline of meta modelling. Meta model components would provide an intuitive way of encapsulate languages. Some identified concepts of the LDE Platform are used throughout the discussion.

Objective: Investigation of component-based meta modelling
Contribution: (The concept of composite DSLs)

18.1 TRADITIONAL COMPONENT MECHANISMS

Before we start our investigation of meta model components, it is beneficial to have a clear understanding about what a component is. A component is, in the traditional sense, a partially independent part of a system or program [22]. That is, a component’s correctness should not depend on other parts of the system or program. Two important concepts are used to describe a component. These are the component’s interface and specification. A component’s interface describes the means of interacting with the component. In other words, this is the parts of a component that are visible to other parts of a system or program. The specification, on the other hand, describes the role and behaviour of the component; what the component can be used to achieve via its interface. Clearly, one important design principle of a component is its isolated implementation. This implies that the component’s structure and logic can be redefined as long as both the interface and specification are fulfilled. Seen from another perspective, it is irrelevant how a component is implemented as long as it does what its specification says as observed through its interface. Consequently, a component can be optimised independently, or easily substituted with another component that has the same interface and specification (or at least provides the same functionality and does not introduce new dependencies). Obviously, a client using a component does not need to know how the component is implemented or functions internally.

A component can be exemplified by a class as defined in Java or C++. Naturally, a class can have several methods. These methods, including the methods’ signatures and return types, represent the class’ interface. A class has a set of properties and a unique behaviour. The properties are accessible via the class’ methods. It is of lesser importance how the class processes a given task, as long as the result is correct and available within a reasonable amount of time. For instance, a method for multiplication of two numbers should always give the correct answer for a given set of argument values.

According to UML 2.0 [18] there are two types of interfaces. These are referred to as provided interface and required interface. A provided interface is used to specify a service that is available to a client of the component. On the other hand, a required interface specifies a service needed by the component in order to function properly. Two interfaces of different types are connected using an assembly connector. Refer Figure 18.1. We will later build on this notation to represent meta components. UML also supports super- and subcomponents. Clearly, components are highly modular entities.
Nesting of classes is a familiar concept in object-oriented programming. In short, a class can be declared within another class. Thus, the inner (non-static) class can access the data fields and methods of the enclosing class in a similar manner as is the case for statements in a non-static method. Note, however, this requires the programming language to be statically scoped. An advantage using inner classes is a higher degree of encapsulation. For instance, an object (of a top-level class) may be composed of several other objects. Let us assume that these objects are only of interest to the composite object, or that they do not have an independent existence. Consequently, it would make perfect sense to declare their respective classes directly within the top-level class. As a result, the classes are declared in their logical context. Moreover, this kind of nesting can continue on an arbitrary number of levels to group related concepts together. There is, however, a distinct difference between traditional nesting of classes used in object-oriented programming and meta model nesting of classes. A model class, for example in Java and C++, is an instance of a meta model class concept. In Java, this meta model concept is named JavaClass. On the other hand, a meta model class, also referred to as a meta class, works on a higher abstraction level. Such a class is an instance of a meta-meta model class concept. In Ecore this concept is named EClass. Refer Figure 18.2. It is important to be aware of this difference because it imposes certain requirements with respect to formalisation. Notice that the term class is used to describe an instance / object of a meta model or meta-meta model class concept.

As can be seen in Figure 18.2, JavaClass and Construct represent two constructs as found in two different languages. In this case, the constructs belong to Java and a custom language, respectively. Hence, models in the languages may include instances of these constructs. For instance, in Java this is achieved by writing a class declaration, as
exemplified with class T {} in the figure. Instantiation of the custom language’s construct can be achieved in different ways depending on the concrete syntax provided for the language.

A composite DSL’s meta model evolves in order to capture more detail. This may compromise the general overview and make it more difficult to relate to the language. An approach for dealing with this is to create models according to language views. In this regard, it is interesting to see what meta model components can bring to the table. A meta model component can be used to encapsulate a complete language. This can improve the usability and flexibility of a composite language because its constructs are logically grouped together.

Nesting of meta model classes (meta model nesting) has been identified as a relevant manner of defining meta model components. Ecore has been used throughout this thesis as the meta-meta model. However, Ecore does not provide means of nesting classes. Consequently, it is necessary to introduce a mechanism for nesting of meta classes. This can be achieved by revising the Ecore meta-meta model. Alternatively, a new meta-meta model can be defined. This meta-meta model works as a middle layer between the Ecore meta-meta model and the language meta models. Figure 18.3 illustrates this further.

A simplified meta-meta model that supports nesting of meta classes is proposed in Figure 18.4. It conforms to the Ecore meta-meta model. Each concept of the model has been prefixed by ‘M’ to avoid confusion with Ecore concepts and concepts of meta models. Only a small selection of attributes is included in the model. Notice that the mechanism supporting nested classes is composition / containment. Composition imposes a rigid life cycle dependency between the outer and inner classes; the inner classes should be destroyed if the outer class is destroyed. The Ecore meta-meta model can be refined to support nested classes in a similar manner.
As can be seen in Figure 18.4, the reflective containment reference mDeclaredClasses makes it possible to build a composite MClass instance by referencing other MClass instances. We assume that these instances are created within the scope / namespace of the top-level / enclosing class, resembling an inner classes structure. Thus, the referred MClass instances can read attributes and references, and invoke operations on the enclosing MClass instance. Consequently, the enclosing instance works as a container for the sub-level instances and has a concise interface as specified by its operations.

### 18.3 Meta Component Definition

Let us see how nested meta classes can be used to deduce a meta model component definition. Specifically, the definition must describe a component’s interface and specification. In short, the idea is to give a meta model class a dual role. Firstly, it should constitute an entity used to reflect a problem domain concept. Secondly, an alternative proposed usage is the role as a meta model component. In other words, a meta model class can either be used as a traditional class or as a component. In fact, the only difference between a class and component in this regard is that a component encapsulates a set of additional concepts. Notice that here we use the term meta component for a meta class that supports / uses nesting. Obviously, the component is still a class, but its role is a component.

We will motivate this idea with a couple of points.

- A class is a well-proved modular concept
- No special component structures are necessary
- Components can easily be introduced by refining classes
- Classes and components can exist in the same namespace

**A class is a well-proved modular concept**

As mentioned, a class is a good example of a component because of its distinct interface and behaviour. Both the interface and behaviour of a class are represented using operations. These operations can often easily be differentiated depending on their role.
Moreover, it is easy to extend a class’ interface to adapt to new parts of a system if needed.

**No special component structures are necessary**

There are many component technologies available like COM (Microsoft) and CCM (Corba). These technologies build on proprietary component models. Instead, by identifying that a class is in itself a component, such proprietary technology can be avoided. Moreover, most developers are familiar with the class concept.

**Components can easily be introduced by refining classes**

According to the idea sketched above, a meta model class always has one of two possible roles. Hence, a class can alternate between being a basic class and a component. This alternation is easily achieved by adding or removing inner classes. For instance, there might be several concepts of a meta model that only relate to one given root concept. There is no reason these concepts should not be encapsulated in their proper context. That is, represented as inner classes within the root class. Additionally, a meta model class can easily be refined with inner classes taking the role as a component in situations where a meta model concept needs to be elaborated to meet its requirements. Notice that this will not break any dependencies with other classes of the meta model since the class’ / component’s interface has not been altered. Furthermore, the component interface can easily be extended by adding the necessary operations.

**Classes and components can exist in the same namespace**

In essence, a meta model component is an elaborated class. Thus, basic classes and components can work together as a coherent whole. In fact, as illustrated in the above paragraph, classes can have alternating roles within a meta model according to the concepts they reflect. Basic classes are used to represent simple concepts, while components are used to represent composite and more complex concepts, including languages and cross-cutting concerns. Both basic classes and components can be used as language constructs. That is, both entities may be instantiated and used in models.

A meta component can be defined by the following description:

*A meta component is a composite meta model class that is elaborated using a set of declared inner classes or meta components. The meta component’s interface comprises a set of top-level operations as found declared in the outer enclosing class. Its behaviour is the accumulated behaviour of the enclosing class and all sub-level classes, as can be observed by invoking the available interface operations.*

Notice that the above definition describes a meta component as a recursive entity. This is because a meta component can declare a set of inner classes that are themselves meta components. Hence, nesting of classes can occur on an arbitrary number of levels.
18.4 DESIGNING A META COMPONENT

We will explain the technical aspects of meta components using an example. We will see how the previously introduced Car language can be represented as a meta component. Note that the design principles and patterns discussed are generic.

![Figure 18.5 Nesting of Car concepts](image)

The first step in creating a meta component for the Car language is to create a composite `Car` construct / meta model where all its logical parts are declared as inner classes. Please refer Figure 18.5. The initial meta model conforms to the meta-meta model found in Figure 18.4. As can be seen, six `MClass` instances (meta classes) are declared as inner classes of `Car`. However, the nesting relation in the diagram does not express how a `Car` instance is composed of the other concepts. Furthermore, additional relationships between the constructs are not expressed in the meta model. Please refer the meta model of the original Car language, as found in the section First example of a composite DSL of chapter The concept of composite DSLs. Hence, the necessary Car meta model structure is not present. This structure is of utter importance in order to formalise the language and to parameterise the valid models. Traditionally, inner classes are used as a way of encapsulating model concepts within an outer enclosing class. These concepts are created freely by a software engineer. Consequently, there is no need to formalise the legal interrelationships between these concepts, or the relationships between the inner concepts and the concept represented by the outer class. However, as pointed out, working with nesting on the meta model level requires such formalisation. Therefore, the required meta model structure has to be created by instantiating the `MReference` concept repeatedly. This resembles traditional meta modelling. The complete meta model for the Car language, as described in the meta-meta model of Figure 18.4, is found in Figure 18.6. It has the same structure as the original Car language’s meta model (in addition to the nesting relation).
The final meta model describes a Car component. The component’s interface is the operations of the composite Car class (and the Car class itself). A detailed overview of this class with a set of likely operations is found in Figure 18.7.

As can be seen in Figure 18.7, the interface of the Car component comprises nine operations. Thus, all communication with other meta model constructs are performed using these operations. Naturally, these operations have to be implemented in the corresponding semantics. An excerpt of a basic one-level implementation of the Car component’s semantics can be seen in Table 18.1.
As might be expected, the Car component’s semantics may be structured using inner classes. Here, only the Engine class is included. A subset of the component’s interface is found at lines 8 and 9. Notice how the methods getHp() and getTorque() access the Engine class’ properties hp and torque. Line 10 defines a method for creating objects of the inner class Engine. This can eventually be incorporated into a factory structure.

However, the identified semantics structure above does not support nesting of classes in more than one level. In fact, inner classes should not be used as part of the semantics when multi-level nested components are used. The reason for this is that inner classes compromise the flexibility and modularity when it comes to integration and merging of several components’ semantics. A suggested manner of implementing semantics for multi-level nested components is illustrated in Table 18.2 (some error-handling is omitted).

<table>
<thead>
<tr>
<th>Table 18.1 - Excerpt of the Car component’s semantics for single-level nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. public class Car</td>
</tr>
<tr>
<td>2.</td>
</tr>
<tr>
<td>3. private String name;</td>
</tr>
<tr>
<td>4. private String model;</td>
</tr>
<tr>
<td>5. private Chassis chassis;</td>
</tr>
<tr>
<td>6. private Engine engine;</td>
</tr>
<tr>
<td>7. private Vector&lt;Seat&gt; seats;</td>
</tr>
<tr>
<td>8. public Double getHp() { return engine.hp; }</td>
</tr>
<tr>
<td>9. public Double getTorque() { return engine.torque; }</td>
</tr>
<tr>
<td>10. public void createEngine() { engine = new Engine(); }</td>
</tr>
<tr>
<td>11. class Engine</td>
</tr>
<tr>
<td>12.</td>
</tr>
<tr>
<td>13. private double hp;</td>
</tr>
<tr>
<td>14. private double torque;</td>
</tr>
<tr>
<td>15. }</td>
</tr>
<tr>
<td>16. }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 18.2 - Excerpt of the Car component’s semantics for multi-level nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. public class CarMulti</td>
</tr>
<tr>
<td>2.</td>
</tr>
<tr>
<td>3. private String name;</td>
</tr>
<tr>
<td>4. private String model;</td>
</tr>
<tr>
<td>5. private HashMap&lt;Part, Class&gt; classes;</td>
</tr>
<tr>
<td>6. private HashMap&lt;Part, IC&gt; ics;</td>
</tr>
<tr>
<td>7. public enum Part</td>
</tr>
<tr>
<td>8.</td>
</tr>
<tr>
<td>9. Chassis, Colour, Engine, Seat, Wheel, Window</td>
</tr>
<tr>
<td>10. }</td>
</tr>
<tr>
<td>11. class IC</td>
</tr>
<tr>
<td>12.</td>
</tr>
<tr>
<td>13. private Part part;</td>
</tr>
<tr>
<td>14. private Vector&lt;Object&gt; instances;</td>
</tr>
<tr>
<td>15. private IC() { instances = new Vector&lt;Object&gt;(); }</td>
</tr>
<tr>
<td>16. }</td>
</tr>
<tr>
<td>17. public CarMulti()</td>
</tr>
<tr>
<td>18.</td>
</tr>
<tr>
<td>19. classes = new HashMap&lt;Part, Class&gt;();</td>
</tr>
<tr>
<td>20. ics = new HashMap&lt;Part, IC&gt;();</td>
</tr>
</tbody>
</table>
A car consists of several parts, as identified in the enum at lines 7 till 10. This enum is used to address the semantics classes corresponding to the inner classes of the meta model Car concept. Refer Figure 18.6. IC (Instance Container) is used to store the instances of a given semantics class. For instance, a car comprises four Wheel instances according to the Car meta model. Obviously, these instances are not necessarily equal and must be stored separately. The semantics class definitions, corresponding to the inner classes of the meta model Car concept, are added at lines 21 till 23 using `addClass(...)`. These are later used to instantiate the semantics classes. Note that `addClass(...)` can be used externally to add...
class definitions. Here, however, the classes are specified directly in the constructor since they reside in the same package as CarMulti.createInstance(...) at lines 29 till 39 is used to create an instance of one of the (inner) semantics classes. This method would be invoked from the Runtime Environment based on the content of the Car model (conforming to the Car component). The component’s interface is defined between lines 40 and 53. It translates to five methods, including a method named t(...), is included merely to illustrate invocation of methods with parameters. Notice how the interface methods forward the method call to the appropriate semantics class; here Engine. Reflection is used to invoke methods in the semantics classes, as can be seen at lines 56 through 60. There are five parameters that are used to specify invocation of a method. Specifically, part addresses the semantics class / car part, id specifies which model instance the method should be invoked on (for example wheel 3), name identifies the method, while arguments and parameterTypes correspond to the method’s signature. Notice that the structure / logic of CarMulti supports multi-level meta model nesting. The same structure can be used for the (inner) semantics classes, for instance Engine.

A small example will illustrate the use of the Car component’s semantics. A simplified semantics class for Engine is shown in Table 18.3.

<table>
<thead>
<tr>
<th>Table 18.3 - A simplified Engine semantics class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>public class Engine</code></td>
</tr>
<tr>
<td>2. <code>{</code></td>
</tr>
<tr>
<td>3. <code>public Double getHp() { return 140.0; }</code></td>
</tr>
<tr>
<td>4. <code>public Double getTorque() { return 320.0; }</code></td>
</tr>
<tr>
<td>5. <code>public Double t(Integer a, Double b) { return a + b; }</code></td>
</tr>
<tr>
<td>6. <code>}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 18.4 - Example use of the Car construct’s / component’s semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CarMulti cm = new CarMulti();</td>
</tr>
<tr>
<td>2. cm.createInstance(CarMulti.Part.Engine, 0);</td>
</tr>
<tr>
<td>3. System.out.println(&quot;Horse powers: &quot; + cm.getHp());</td>
</tr>
<tr>
<td>4. System.out.println(&quot;Torque: &quot; + cm.getTorque());</td>
</tr>
<tr>
<td>5. System.out.println(&quot;t(3,4.5): &quot; + cm.t(3,4.5));</td>
</tr>
</tbody>
</table>

The code of Table 18.4 shows a manual invocation of the Car component’s / CarMulti’s methods. Method invocations and orchestration of components would ultimately be performed by the Runtime Environment. Notice that only one instance of the Engine class is created. This instance is given the id 0, as can be seen on line 2. Three methods of the Car component’s interface are invoked at lines 3 through 5. The result is shown in Figure 18.8.

```
Horse powers: 140.0
Torque: 320.0
t(3,4.5): 7.5
```

Figure 18.8 Executing program using the Car component

It can be tempting to try using subtyping for definition of multi-level semantics. However, since a component can have more than one inner class, this is not feasible. Note that the current version of the Runtime Environment does not support meta components.
18.5 Meta Modelling Using Components

We have illustrated how to create the abstract syntax and semantics of meta components. The next step is to shed light on how and why meta components should be used in meta modelling. Specifically, a component can take two roles: as a construct or an auxiliary component. A construct can be used as part of a model, whereas an auxiliary component can be used to store language-specific data. That is, it is not possible to instantiate this entity for use in a model (declared as abstract). As illustrated, meta components are ideal for encapsulation of language constructs. In essence, there are three major advantages to this.

Increased organisational value
Firstly, a language’s meta model becomes more intuitive and easier to comprehend because of segmentation according to aspects / parts. Moreover, on the contrary to traditional meta modelling, meta components make it possible to model / create languages in two dimensions. The reason for this is because a meta component supports declaration of inner meta components. That is, an internal multi-level architecture for meta models (at the M2 level). We emphasise that this notion must not be confused with multi-level meta model architectures like MOF and Ecore.

Figure 18.9 Two-dimensional meta modelling

Figure 18.9 illustrates how a language named A can be built in a two-dimensional manner. It encapsulates two components / languages known as B and C. B is constructed using two classes, B1 and B2, and one auxiliary component X. C is composed of two components; a language named D and a component named Y. D comprises two constructs known as D1 and D2. As noted, a component can have the role as a language construct. This is the case for the components A, B, C, D and Y. Recall that a component is still a class that can be viewed as the root class of the encapsulated meta model segment / language.

There are several relations between components and classes in Figure 18.9. In detail, C uses a service provided by B, B1 and B2 use services provided by the auxiliary component X, and D has a bidirectional communication with the Y component. Note that only relations between two components, or one component and one class, are shown in the diagram. The meta model structure can be expressed using a standard class diagram. Each component in the figure is indexed to illustrate its context. We stress that all entities in the figure reside on the M2 level of the MOF / Ecore meta model architecture.

Promotion of modularity
A component is a highly modular entity. Consequently, a component can be substituted with another component as long as their interfaces are conjunct. This is valuable in
language design where a component can be used to encapsulate a complete language. A composite language can be built by combining several sub-order components / languages. Moreover, if a sub-order language is not accurate enough, it can quickly be substituted with another. This gives a high degree of flexibility and also promotes reuse of meta models (and models).

A composite language composed of three components / languages is shown in Figure 18.10. The provided interfaces are tagged with names. Car and Engine comprise a set of basic inner classes. Compressor, on the other hand, is constructed using two basic classes (CHRA, Impeller) and three components (Turbine, CA and CB). Notice that Turbine declares an inner component named CC. CC is only accessible via Turbine since it resides on a lower level than the rest of the Compressor’s entities. In this case, CHRA, Impeller and Turbine constitute the Compressor constructs. Thus, these can be used to model a compressor. CA and CB are auxiliary components. CC is an auxiliary component used to define Turbine. Car, Engine and Compressor are themselves constructs. Entities that are defined as language constructs have their name in purple.

There are two ways to model relations between components. In Figure 18.10 an interface notation is used where required and provided interfaces are connected. However, since a component is also a class, traditional meta modelling relations like references and inheritance can be used as well. This includes references where the multiplicities are greater than 1. Thus, a component can be linked to several instances of another component through the same set of interfaces. Refer Figure 18.11. Notice that the interface notation has been used between classes and components in the component diagrams to identify a dependency. As mentioned, traditional class diagrams are used to describe the internal structure. A combination of the diagrams can also be used.
Modelling according to aspects

We have earlier discussed how models can be created according to selected language views. A similar approach can be used for components in meta modelling. This is achieved by creating a model for each component that is also a language construct. For the preceding example this yields four models conforming to the components: Car, Engine, Compressor and Turbine. As might be expected, if CC (or CA, CB) had been a construct (on the contrary to an auxiliary component) of Turbine, a model conforming to CC had to be created / loaded as well. The reason it is possible to create models separately is because all interactions between models are formalised by the component interfaces. An example will illustrate this further.

Together these comprise a model for the composite language. The models can be interpreted separately (by an updated version of the Runtime Environment). In fact, they do not need to be merged in order to be processed. All relations between the models are preserved by the integrated semantics classes, as described by linking component interfaces. Clearly, a compiler / parser for meta components has to resolve the nesting relations in order to put classes and components in the correct contexts.

As we have seen, the interfaces of meta components constitute a set of class operations. An operation is an instance of a generic meta-meta model concept. Hence, it can be necessary to specify additional constraints. For instance, there might be certain conditions for a component to work properly, or, there might be components that should not be combined. OCL invariants [35] can be used to specify this kind of additional static semantics. An invariant works like a predicate that can be used to assess whether certain properties and conditions are fulfilled. Work has been performed on OCL-constrained
meta model conformance for MOF [36]. Initially, the conformance relation between a model and its meta model was not formally defined when OCL expressions were used. As a consequence, the effect of OCL constraints on the semantics of a model type could not be determined. The work identified has addressed this issue. Specifically, it introduces a meta model specification as a pair \((M, C)\), where \(M\) is a MOF meta model and \(C\) is a set of meaningful OCL constraints. Moreover, proposed algebraic semantics for meta model specifications has extended the notion of the structural conformance relation to include OCL constraints. Consequently, the result of this work can likely be applied for Ecore meta models. This makes it possible to verify formally if constraints related to component instantiation and interaction are fulfilled. Note that the abstract syntax of a component is described using a class. This implies (is a proof) that a component conforms directly to the meta-meta model class concept.

18.6 PRACTICAL META MODELLING USING COMPONENTS

Meta components introduce a new approach of language design. However, tool support is critical in order to present the technology in an efficient, intuitive and clear manner. One central question is how modelling environments and IDEs can support meta components. Specifically, it should be possible to focus on one component / aspect at the time, and still have the benefits of using diverse development tools. A first step is to identify the required functionality of two-dimensional modelling. Thus, an initial mock-up of an Eclipse perspective is found in Figure 18.13.

![Figure 18.13 Eclipse perspective for component-based meta modelling](image-url)
Six views constitute the proposed development perspective. These are **Outline, Context, Overview, Interfaces, Class** and **Component**. A brief explanation of each view follows.

**Outline**
The main view of the perspective is the Outline view. It is used to access classes and components of a composite language. Here, the project file with the language from Figure 18.10 is opened. All entities of the language are stored in a package named `languages.car_v3_0_0.stable`. In this case, there are three components named `Car`, `Compressor` and `Engine`. `Compressor` comprises two classes and a component known as `Turbine`. `Turbine` is defined using a component named `CC`. Refer Figure 18.10 for details. `Turbine` has been selected in the Outline. Thus, the other views centre on this component. Components that can be instantiated and used in a model are marked in purple. Classes are by default language constructs. According to traditional meta-modelling, they can be defined as abstract to avoid being instantiated (and used in models).

**Context**
Contextual information is available in the Context view. This view relates to the parent package or component. In this case, `Turbine`'s role within the `Compressor` component can be viewed. In detail, an object of the `CHRA` construct is composed of an `Impeller` object and an object of the `Turbine` (component) class.

**Overview**
A verbose one-level meta model “can be seen” in the Overview view. This meta model includes all attributes, references, operations and eventually static semantics. Moreover, the `Compressor` concept is represented in the meta model as a class. Thus, the Overview can be used to perform meta modelling. Notice that the Overview is the same regardless of what entity of the `Compressor` component that is selected.

**Interfaces**
A component’s or class’ public interfaces / operations can be viewed in the Interfaces view. This view makes it possible to address dependencies in order to integrate entities. Five interfaces of the `Turbine` can be seen in the current view. This includes information on operation name, type and meta information / description.

**Class**
Class provides the developer with a standard one-level class diagram of `Turbine`. That is, the inner classes and components. (Here, only one component is encapsulated by `Turbine`.) The `Turbine` entity is represented in this diagram as well. In this view, `Turbine` is considered and represented as an elaborate class. Refer Figure 18.6 for details on how a component is constructed using inner classes. The Class view can be used for meta modelling.

**Component**
An alternative to Class is the Component view. It treats `Turbine` as a component. The Component view can be used to build a language from components. All components of the composite Car language can be viewed by selecting the language package `languages.car_v3_0_0.stable`. 

169
Additional views could include support for creation of models according to the composite language or a language part, debugging and template functionality. Furthermore, functionality for loading of resources within the views and a tailored search mechanism could also be valuable. Notice that each view has shortcuts for navigating, sorting, etcetera. A similar perspective can be used to support composite DSL design and development as discussed throughout this thesis.

There is one limitation with regard to the instantiation model of Ecore meta models. Specifically, the current instantiation mechanism of an Ecore meta model treats instances as objects. That is, the instances of a meta model class only have properties found in objects. Objects can not be further instantiated. In the case of meta-meta modelling it is desirable that instances can represent classes as well. This implies that properties like attributes and references are available on the instances. Consequently, the instances are available for further instantiation. This is especially important when a middle layer meta-meta model is created and used. A similar limitation has been identified for UML.

18.7 COMPONENT HIERARCHY
Encapsulation, as available using inner meta classes, supports the notions of super- and subcomponents. A subcomponent always resides within another component. By this technique, a component hierarchy can be created. In the context of language design, this allows a language to reside within another language in the context where it naturally belongs. For instance, the Engine language could be encapsulated within the Car language. Thus, the Engine language is a sub language to Car. Clearly, this structure results in a language hierarchy consisting of several layers. As might be expected, the abstraction level is likely to decrease when layers are traversed top-down.

18.8 WEAVING OF COMPONENTS
We have defined a meta component’s interface as its operations. However, in the light of topics discussed in this thesis, it makes perfect sense to consider the entire top-level class as the component’s interface. This makes it possible to weave components together in a similar manner as described for meta classes. Also, this includes weaving of a component with a class. As a consequence, a component can be used either by invoking its operations or by weaving it with another entity.
19 Related work

This chapter gives an overview of related work to topics of this thesis. It is not possible to discuss all related research. Thus, we have narrowed it down to five different, but relevant cases.

19.1 CASE 1: MOF 2.0 META MODEL COMPONENTS

Article name: “Formal Definition of MOF 2.0 Metamodel Components and Composition”

Relevance: We have earlier discussed the possibilities associated with defining meta models. This was discussed in the light of a promoted approach where meta model nesting is used to realise meta components.

Another way of defining meta components is elaborated in [9]. This work discusses how the MOF 2.0 standard can be extended with modularisation concepts. Specifically, an updated version of the MOF package concept is suggested in order to define proper components. This includes specification of export and import interfaces. These interfaces support composition of meta models (components). In more detail, export interfaces are used to identify submodels of meta model components that should be visible outside of the component. Import interfaces provide a mechanism for parameterisation of meta model submodels. These submodels can be instantiated and replaced by exported elements from other components. Furthermore, a meta model composition operator is defined. This operator can be used to create new components by instantiating and relating existing components with each other. Binding relationships between the export and import interfaces are performed using graph morphisms.

The work is motivated by the fact that definition and maintenance of complex Visual Domain-Specific Modelling Languages, like UML, are inconvenient using the MOF 2.0 standard. Moreover, the lack of a mechanism for differentiating between externally visible and internal structures of a component in native MOF 2.0 is also noted. The approaches discussed are general and compatible with any meta model conforming to the MOF architecture. The suggested meta component definition is flat. That is, all components are defined at the same modelling level (not to be confused with meta levels). The concepts and ideas are theoretical by nature. Claimed contribution includes:

- Support for definition of libraries of reusable meta model components
- Mechanism for definition of parameterised language descriptions
- A meta model composition operator

19.2 CASE 2: TECHNIQUES FOR META MODEL COMPOSITION

Article name: “Techniques for Metamodel Composition”

Relevance: Reuse of languages has been an important topic of this thesis. In order to support this, a complete platform for language design and development has been suggested. This platform represents a common foundation that can be used to unify languages.

Techniques for reuse and composition of meta models are discussed in [7]. Specifically, the different approaches of combining meta models are elaborated, including Meta model Merge, Meta model Interfacing and Class Refinement. A modelling language design
environment is envisioned to address the different tasks of language design. It is argued that meta model reuse will result in benefits like avoidance of duplication of effort, high-quality reusable meta model fragments, recognition of meta modelling patterns and best-practices, and more efficient development of Domain-Specific Modelling Languages. In addition, a meta model composition method titled Template Instantiation is described. In short, this concept comprises a technique for defining abstract meta model templates that can be instantiated and used in domain-specific meta models. Several meta modelling templates have been identified. Some of these are composition hierarchies of composite and atomic objects, state chart style modelling and proxy meta modelling where a reference type can be defined for a class with the same role and properties of another class.

Important points motivating the work are the observations that there are few best-practices for meta modelling and no public collection of reusable meta models. Moreover, tools for a variety of composition techniques are identified as essential in order to support reuse of meta models.

Claimed contribution comprises:

- Identification and discussion of meta model composition approaches
- A new approach for meta model composition known as Template Instantiation
- Elaboration on the issue of composite meta model semantics

19.3 Case 3: When and How to Develop DSLs

Article name: “When and How to Develop Domain-Specific Languages”

Relevance: Defining DSLs according to best-practices ensures efficient development and optimal solutions. Some methodologies for composite DSL design have been discussed in this thesis. Important points of these methodologies were focus on cohesive languages and reuse.

General methodologies for DSL development are addressed by [23]. This work identifies patterns for all major phases of traditional DSL development including the decision, analysis, design, implementation and deployment phases. In addition, a discussion on analysis tools and language development systems is included.

The work is motivated by the observation that DSL development is difficult and requires both domain knowledge and language development expertise. Moreover, it is claimed that DSL development is more varied than GPL development and that language support, standardisation and maintenance of DSLs may potentially become serious and time-consuming issues. Different aspects of DSL language design are thoroughly discussed. Furthermore, the article features an extensive survey of technologies used in DSL development. Claimed contribution is:

- Improved and extended work on DSL patterns
- Identification of tools and systems for DSL development
- Elaboration on open problems that need to be addressed

19.4 Case 4: Modelling with Multiple DSLs

Article name: “An Integrated View on Modeling with Multiple Domain-Specific Languages”
Relevance: The interactions between the languages of a composite DSL have been a central point in this thesis. These interactions are based on weaving of abstract syntaxes and linking of semantics modules. Moreover, modelling according to language views has been elaborated.

A method for modelling using multiple DSLs is suggested in [37]. One central point is how to deal with overlapping concerns between DSLs. Two types of mechanisms for integrating DSLs are discussed: soft references and semantic connections. Soft references are achieved by using a matching string key for attributes in models of different DSLs. That is, one DSL refers another DSL. Semantic connections occur when elements from different languages overlap. These overlaps can be expressed using semantic descriptions. Soft references merely require enforcing referential integrity. Semantic connections, on the other hand, impose requirements regarding a reference’s semantic validity. It is stated that current tools do not provide consistency checking of constraints related to semantic connections. The suggested method for development with multiple DSLs comprises three steps: Identification, Specification and Application. Identification is used to find connections between DSLs. The specification step comprises encoding of the connections for analysis purposes in order to find suitable representations of the DSLs, their models and connections. Two approaches for specification based on inference are discussed. Application comprises using the encoded connections to: determine dependencies and relationships between artefacts, perform consistency checking and provide modelling guidance.

Motivation for the work includes the observations that there is little known about the interaction between several DSLs of a single system and that complex systems require the use of multiple DSLs. Specifically, it is stated that using multiple DSLs is necessary to handle separation of concerns and to offer several views on a system. Claimed contribution is:

- A method for development with multiple DSLs
- Tools supporting the method
- Elaboration on case studies of development with multiple DSLs

19.5 CASE 5: POLYMORPHIC EMBEDDING OF DSLs

Article name: “Polymorphic Embedding of DSLs”

Relevance: Composition and reuse of DSLs have been discussed. Specifically, an aspect language can be incorporated into numerous primary languages. Moreover, a DSL’s semantics can be customised to address special-case scenarios. Refer the discussion of EDSLs on the accompanying CD.

A method for embedding DSLs within a host language is discussed in [38]. This includes elaboration of functionality for composition and reuse of DSL programs and their interpretations. A central point is abstraction over the implementation of domain-specific operations and types.

The motivation behind the work includes the fact that traditional EDSLs are restricted to a single interpretation. This prevents flexible composition and reuse of languages. Moreover, analysis and optimisation is not straightforward. Claimed contribution is:

- An approach for embedding DSLs that supports multiple interpretations
- Elaboration on how domain-specific analysis and optimisation can be done
- Illustration of how polymorphic embedding supports composition
20 Conclusions

Here we present conclusions on the work of this thesis and an overview of contributions. This includes a description of known weaknesses. In a later chapter we will identify central points of future work.

20.1 SUMMARY

The work of this thesis has comprised addressing issues regarding design and development of composite Domain-Specific Languages, evaluation of DSLs with respect to frameworks, and meta modelling using components.

20.1.1 COMPOSITE LANGUAGE DESIGN AND DEVELOPMENT

Language Driven Development is an increasingly popular approach in software engineering. However, as motivated, there are not many practical solutions of unifying languages and ensuring that best-practices are followed. Moreover, changing and evolving requirements may potentially imply a tedious and inefficient development process since changes can not easily be reflected in languages.

To address these issues, a complete platform for design, development and model execution has been constructed. This platform includes tools, editors and runtime environments that address the shortcomings of traditional Domain-Specific Language use and development. Aspect-oriented weaving has been identified as a mechanism for supporting dynamic language composition.

Specifically, the platform support:

- Standardised language design and structuring
- Unification and reuse of languages based on a common platform and meta model architecture
- Aspect-oriented customisation and extension of languages to meet new requirements
- Evolving languages by recycling models
- Modelling according to aspects / language views
- Decentralised development and language integration
- Execution of models conforming to composite languages

As we have seen in this thesis, composite language design provides software engineers with an efficient approach in software development. Important key principles are focus on flexibility, integrability and reusability.

Recycling of models

Evolving languages results in incompatibility issues between models and meta models. Specifically, changed language properties imply new language versions. As a consequence, models of an earlier language version are likely not to conform to the new language version’s meta model. This can be addressed using model transformations where a model conforming to an earlier language version is merged with a set of aspect language models. The result of the transformation process is a model that conforms to the updated language version, and thus, reflects new language properties.
Modelling according to aspects / views
Aspect-oriented weaving is a mechanism of incorporating aspect languages into a primary language. This mechanism also supports modelling according to custom language views. Consequently, software engineers can focus on language segments that reflect concepts and concerns within their area of expertise by creating models according to certain views. Subsequently, these models can be transformed to achieve a composite model conforming to the composite DSL, known as convergence of multiple views.

Decentralised development
Specialisation and distribution of knowledge complicate software development. Domain expertise is often possessed by stakeholders that do not necessarily have software development skills. Hence, good communication between domain experts and software developers is crucial. If this communication fails, the result can be poor or, in worst case, useless software. Two aspects of decentralised software development have been addressed in thesis. Firstly, aspect languages can be created in a decentralised manner and woven together to constitute a rich composite language. Secondly, models conforming to each aspect language can be created by different parties and subsequently merged together.

Community-based cooperation is a typical example of decentralised development. This kind of development can be supported by providing a community website with communication tools, including a database for available aspect languages - a remote Language Repository. This way, aspect languages can be downloaded on a need-to-have basis and woven together with a primary language to provide functionality for a given type of concerns.

20.1.2 FRAMEWORKS AND DSLs
Frameworks, DSLs and composite DSLs are all highly usable approaches in modelling problem domain concerns. There are scenarios where frameworks are better suited than DSLs, and vice versa. Frameworks provide a high degree of customisation and flexibility, as illustrated with the Ecommerce framework. However, framework instantiation is questionable since this requires manual writing of instantiation code. Conversely, traditional DSLs are static in nature and do not provide mechanisms for customisation. There is, however, no known reason why a DSL should not be created in a manner that supports reuse and composition as illustrated by the work of this thesis. Thus, we will narrow the alternatives down to frameworks and composite DSLs.

Whether to use a framework in preference to a composite DSL basically depends on the required freedom of modelling and integration with other technologies. Important questions to ask in resolving these issues are: Should developers be able to provide proprietary solutions, or should all models conform to pre-defined problem domain constructs? Is it necessary for the framework or language to interoperate with other tools, platforms and technologies? If freedom and support for proprietary solutions are highly desirable a framework may be the best choice. With regard to integration issues, this ultimately comes down to how flexible and non-proprietary the LDE Platform is. Implementing a platform using a common GPL is the first step in ensuring a flexible solution. For instance, the proof-of-concept LDE Platform of this thesis is implemented in Java. Consequently, it can easily be integrated with other Java technologies. The same applies to the semantics of the compatible languages. This has been illustrated by creating a model-driven web shop as a J2EE Servlet.
A clear advantage using composite DSLs is verification of models directly against incorporated problem domain knowledge. This ensures models of higher quality and simplifies the modelling process. Consequently, more stakeholders of a project can be involved closely in the modelling process, which yields more accurate results and contentment. In addition, no manual instantiation code has to be written using DSLs. In addition, support for modelling according to aspects advocates using composite DSLs.

20.1.3 META MODELLING USING COMPONENTS
Mechanisms for creation of and modelling with meta components have been elaborated. These mechanisms are based on meta model nesting and a dual interpretation of meta classes. Specifically, a meta class can have two roles: a traditional meta class or a meta component. Meta components can be used to group language features together. For instance, all constructs of a language can be defined by means of inner meta classes of an enclosing meta class. This increases encapsulation and modularisation. Moreover, a component can be designed to address a certain aspect in a similar manner as an aspect language. An important observation is that a component is still a meta class. Hence, components can be used directly as language constructs providing means of hierarchical meta modelling. In other words, languages can be created in two dimensions. The first dimension corresponds to classical flat meta modelling, while the second dimension reflects nesting of meta classes / components. Meta model nesting provides a mechanism for defining super- and subcomponents in the proper contextual setting. That is, within an enclosing class / component.

20.2 WEAKNESSES
Weaving languages together yields more complex languages that address aspects of composite problem domains. Moreover, concepts on different abstraction levels are mixed together. It may be argued that overview is lost as more details are captured by a language. Countermeasures of this observation are modelling according to language views, meta components and use of namespaces. Hence, the different parts of a composite DSL can still be evaluated and used separately.

20.3 CONTRIBUTIONS
An approach for design and development of composite DSLs has been elaborated and verified in this thesis. We have shown that aspect-oriented language weaving is an optimal mechanism for extending and customising languages in an intuitive and pragmatic manner. This mechanism also makes it possible to model concerns according to custom language views / aspects, where different stakeholders / parties can create independent aspect models of a composite language. Semantic coherence between languages has been ensured using weaving of abstract syntax and explicit interfaces for semantics integration. An important aspect of the solution is support for unification and reuse of languages. Moreover, execution of models conforming to composite DSLs is supported by the Runtime Environment. This is possible using dynamic proxies and generic method invocation dispatching.

Diverse scenarios regarding software evolution and decentralised development have been demonstrated. Specifically, tools of the LDE Platform have been used to verify that the scenarios identified can be dealt with in a straightforward manner. This is an important result seen in the light of Language Driven Development. Specifically, we have
illustrated that support for evolving languages can be achieved using model transformations. We believe that the ideas and principles behind the transformation processes described provide the foundation for building a complete transformation system for models of more rich and complex languages. Moreover, decentralised development of languages has been addressed using an aspect-oriented and incremental development model where remotely created aspect languages are incorporated into a primary language. Experience has indicated that languages can be created as a common effort by focusing on separation of concerns and standardisation of language interaction points.

We have related the discussion on composite language design to the concept of components. We have seen how meta components can be realised using meta model nesting. A solution for defining multi-level semantics has been provided as well. We believe that using meta components in the language design process opens up a new world of meta modelling yielding increased modularity, contextual awareness and a higher organisational value. This includes capabilities for defining super and sub languages and language hierarchies. An important observation of the promoted approach is that meta classes and components can co-exist within a language meta model. Thus, language constructs can be expressed using a combination of these entities. At present, there is no similar known work performed on multi-level meta components.

Currently, frameworks are the most popular approach in software engineering. The main reason for this is a framework’s high degree of flexibility and support for customisation. An evaluation of frameworks and composite DSLs has been performed. We hope and believe that composite DSLs can provide software engineers with a strong alternative to frameworks, due to the dynamic and aspect-oriented nature of these languages.

Composite DSLs are a technology-independent concept. Hopefully, ideas and principles elaborated can serve as foundation for other future language development platforms and frameworks.

20.4 CRITICAL ASSESSMENT
Evolving languages and modelling according to views have been addressed using model transformations. There are, however, complex scenarios that are difficult to support. Specifically, destructive weaving and coordination of multiple tangled views are challenging. More work has to be done on model transformations to investigate how to resolve these issues.
21 Future work

Suggested revisions and extensions to concepts and implementations have been identified in previous chapters. Here, we will sum up what we consider is the essential continuation of the work presented in this thesis.

21.1 SOLIDIFICATION OF WORK

In general, future research should address solidification of the presented work. This includes elaboration on concrete design methodologies and standardisation of best-practices comprising language development kits and templates, and revision of proof-of-concept applications.

21.2 LDE PLATFORM 2.0

The first version of a Java compatible LDE Platform was developed as part of this thesis. This version does not support meta components, as defined using nested meta models. Clearly, functionality for meta components is desirable for a future version of the LDE Platform. To a large extent, it should be feasible to integrate meta components with the different concepts, mechanisms and tools discussed in previous chapters. Thus, meta components could be an optional modelling technology to the traditional flat meta modelling.

In addition, it is desirable with functionality for automatic concept renaming and processing of namespaces. More work has to be performed to investigate how the Model Transformer can support transformation scenarios where destructive meta model weaving has been used. Also, this tool should support all kinds of meta relations. Finally, development of perspectives, wizards and more comprehensive editors would increase the value of the platform with respect to usability.

Consequently, the LDE Platform 2.0 will be a powerful tool for language design and development supporting extensible languages, decentralised development, recycling of models, modelling according to language views, meta components, and more.

21.3 INCREASING INTEGRABILITY

Implementations of tools and the Runtime Environment are based on standardised technologies like Ecore, Java, Kompose (Kermeta) and ATL. Further integration with other technologies can be achieved by creating bridging technologies. There are many interesting opportunities in this regard, including distribution of a language’s semantics to modules written in arbitrary host languages, and integration with UML 2.0. Other relevant updates include support for other IDEs than Eclipse and standalone spin-off products that can be closely integrated with other technologies. MOF 2.0 support is also desirable.
Bibliography

BOOKS


SCIENTIFIC ARTICLES, SPECIFICATIONS AND MANUALS


179


[24] Anna Gerber, Kerry Raymond, MOF to EMF: There and back again, Cooperative Research Centre for Enterprise Distributed Systems (DSTC), University of Queensland, Brisbane.


[32] Arthur Boronat, José Meseguer, Algebraic Semantics of OCL-constrained Metamodel Specifications, Department of Computer Science, University of Leicester, and Department of Computer Science, University Illinois.


WEB RESOURCES


APPENDICES
A Delivered material / CD contents

This appendix describes the delivered material and the contents on the accompanying CD.

A.1 DELIVERED MATERIAL
The delivered material comprises the thesis and an accompanying CD with all source code. A complementary document is available on the CD. This elaborates on a selection of topics including software evolution, composite DSL terminology, community-based development, social aspects of cooperative work, alternative weaving sequences, language versioning, code generation, embedded languages and more.

A.2 CONTENTS ON ACCOMPANYING CD
Ten folders are found on the CD. These are described in Table A.1.

<table>
<thead>
<tr>
<th>Folder:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>eclipse bundle</strong></td>
<td>Contains a complete Eclipse bundle with:</td>
</tr>
<tr>
<td></td>
<td>- Source code for all proof-of-concept applications</td>
</tr>
<tr>
<td></td>
<td>- Examples</td>
</tr>
<tr>
<td></td>
<td>- Installed plug-ins</td>
</tr>
<tr>
<td></td>
<td>- Installed technologies like EMF, Kermeta, Kompose and Ecore Tools</td>
</tr>
<tr>
<td><strong>plug-ins</strong></td>
<td>All plug-ins created including tools and editors</td>
</tr>
<tr>
<td><strong>atl preprocessor</strong></td>
<td>The ATL Preprocessor</td>
</tr>
<tr>
<td><strong>standalone runtime environment</strong></td>
<td>The standalone Runtime Environment as used by the Webshop application</td>
</tr>
<tr>
<td><strong>ecommerce framework</strong></td>
<td>Source code for the Ecommerce framework and graphical components</td>
</tr>
<tr>
<td><strong>web shop application</strong></td>
<td>The Webshop Servlet application as a deployable WAR file</td>
</tr>
<tr>
<td><strong>document definitions</strong></td>
<td>DTDs and XML Schemas of LDE Platform file types</td>
</tr>
<tr>
<td><strong>thesis</strong></td>
<td>Final version of the thesis as a PDF document</td>
</tr>
<tr>
<td><strong>complementary document</strong></td>
<td>Includes more information on a selection of topics</td>
</tr>
<tr>
<td><strong>runtime log</strong></td>
<td>Complete example of runtime log</td>
</tr>
</tbody>
</table>