

Sanna Sivonen

Domain-specific modelling language
and code generator for developing
repository-based Eclipse plug-ins

VTT PUBLICATIONS 680

Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins

Sanna Sivonen



ISBN 978-951-38-7094-2 (URL: <http://www.vtt.fi/publications/index.jsp>)

ISSN 1455-0849 (URL: <http://www.vtt.fi/publications/index.jsp>)

Copyright © VTT 2008

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 3, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax +358 20 722 4374

VTT, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100, FI-90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Text preparing Tarja Haapalainen

Sivonen, Sanna. Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins [Sovellusaluekohtainen mallinnuskieli ja koodigeneraattori tietokantapohjaisten Eclipse-laajennusten kehittämiseen]. Espoo 2008. VTT Publications 680. 89 p.

Keywords model-driven development, software product family, variability

Abstract

Eclipse is an open source platform for tool integration which can be extended by writing plug-ins that utilise the extension points provided by the Eclipse platform. Eclipse plug-ins are written in the Java language and the plug-in development work can be time consuming especially if multiple plug-ins are developed for the same application domain. Model-driven development is about focusing on models rather than computer programs in software development. Domain-specific modelling follows the principles of model-driven development by promoting the use of domain-specific modelling languages (instead of general-purpose modelling languages).

The aim of this research is to develop a prototype graphical domain-specific modelling language (DSML) and a code generator for creating repository-based plug-ins for Eclipse. The purpose of DSML is to raise the level of abstraction and thus speed up the development process of several similar Eclipse plug-ins compared to hand writing the plug-ins in Java language. Also people not familiar with Java (i.e. end users) could build their own extensions with the language defined in this work.

Developed DSML is demonstrated by generating the source code of an existing repository-based Eclipse plug-in. The plug-in that is used in the demonstration is an open source software architecture knowledge management tool called Stylebase for Eclipse, which has been developed at VTT Technical Research Centre of Finland. The Stylebase for Eclipse is a software product family, which has a number of variation points. Since the Stylebase for Eclipse tool is already developed once with the traditional code-centred approach, it is possible to compare the model-driven approach with the code-centred approach in this particular case.

The case example shows that DSML and the code generator defined in this work can be used for generating repository-based Eclipse plug-ins. The code generator generates a fully functional Eclipse plug-in so the generated code does not have to be edited manually after its generation. Also variability in the software product family can be handled in a more flexible way in the model-driven approach.

Sivonen, Sanna. Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins [Sovellusaluekohtainen mallinnuskieli ja koodigeneraattori tietokantapohjaisten Eclipse-laajennusten kehittämiseen]. Espoo 2008. VTT Publications 680. 89 s.

Avainsanat model-driven development, software product family, variability

Tiivistelmä

Eclipse on avoimen lähdekoodin alusta, jota käyttäjät voivat laajentaa hyödyntämällä Eclipse-alustan tarjoamia laajennuspisteitä. Eclipse-laajennukset kehitetään Java-ohjelmointikielillä ja kehitystyö voi olla haastavaa, erityisesti kehitettäessä useita hieman toisistaan poikkeavia sovelluksia samalle sovellusalueelle. Malliohjattu ohjelmistokehitys keskittyy ohjelmiston malleihin lähdekoodin sijasta. Sovellusaluekohtainen mallintaminen on eräs tapa toteuttaa malliohjattua ohjelmistokehitystä. Sovellusaluekohtaisessa mallintamisessa käytetään sovellusaluekohtaisia mallinnuskieliä yleiskäyttöisten mallinnuskielten asemesta.

Tämän tutkimuksen tavoitteena on määritellä sovellusaluekohtainen mallinnuskieli ja koodigeneraattori tietokantapohjaisten Eclipse-laajennusten kehittämiseen. Mallinnuskielen ja koodigeneraattorin tarkoituksena on nostaa ohjelmistomallin abstraktiotasoa ja siten nopeuttaa tietokantapohjaisten Eclipse-laajennusten kehittämistä verrattuna laajennusten manuaaliseen ohjelmointiin. Myös käyttäjät, jotka eivät hallitse Java-ohjelmointikieltä, voivat kehittää tietokantapohjaisia laajennuksia Eclipseen tässä työssä kehitetyn mallinnuskielen avulla.

Työssä kehitetyn sovellusaluekohtaisen mallinnuskielen ja koodigeneraattorin käyttöä havainnollistetaan generoimalla olemassa olevan tietokantapohjaisen Eclipse-laajennuksen lähdekoodi. Laajennus, jota käytetään havaintoesimerkissä, on arkkitehtuuritietämyksen hallintaan käytettävä Stylebase for Eclipse -työkalu, joka on VTT:n kehittämä avoimen lähdekoodin työkalu. Stylebase for Eclipse on tuoteperhe, jolla on useita varioituvuuspiisteitä. Koska Stylebase for Eclipse on aiemmin kehitetty koodikeskeisellä sovellusten kehittämistavalla, voidaan tässä työssä vertailla mallikeskeistä tietokantapohjaisten Eclipse-laajennusten kehittämistä koodikeskeiseen tapaan.

Havaintoesimerkki osoittaa, että työssä kehitettyä sovellusaluekohtaista mallinnuskieltä ja koodigeneraattoria voidaan käyttää Eclipse-laajennusten kehittämiseen. Koodigeneraattori tuottaa toimivan Eclipse-laajennuksen, joten generoitua koodia ei tarvitse editoida manuaalisesti generoinnin jälkeen. Mallikeskeisessä kehitystyössä myös tuoteperheen varioituvuuden hallinta on joustavampaa.

Preface

This work was done in the Product Family Architectures team at VTT Technical Research Centre of Finland during the year 2007. The work forms part of an international joint research project called Model-driven development of highly configurable embedded Software-intensive Systems (MoSiS). VTT's objective in MoSiS is to develop transformation techniques and supporting concepts to support automated or semi-automated architecture model transformations. Further, the project concentrates on modelling and visualisation of extra-functional software properties.

I would like to thank my supervisor Mari Matinlassi from VTT for instructing me with my work and all the co-workers at VTT for creating a fun and relaxed atmosphere in which to work. I would also like to thank Juha Rönning and Jukka Riekkilä from the University of Oulu for their valuable comments during the work. I would like to express my gratitude also to MetaCase staff for answering my questions quickly. Last, but certainly not least, I would like to thank my family and friends for their love and support, and especially my boyfriend Tommi for listening to my worries and being the light of my days.

Oulu 18.2.2008

Sanna Sivonen

Contents

Abstract	3
Tiivistelmä	5
Preface	7
List of abbreviations	10
1. Introduction.....	13
2. Model-driven development of software product families.....	16
2.1 Non-MDD development.....	16
2.2 Model-driven development	17
2.2.1 MDD with Model-Driven Architecture approach.....	17
2.2.2 MDD with domain-specific modelling approach.....	18
2.2.3 Comparison between MDA and DSM	20
2.3 Software product families.....	21
3. Tools and technologies	23
3.1 Eclipse	23
3.2 Metamodelling tools.....	26
3.2.1 DSL Tools for Visual Studio.....	27
3.2.2 Generic Modelling Environment.....	28
3.2.3 Generic Eclipse Modelling System	29
3.2.4 MetaEdit+.....	29
3.2.5 Comparison of the tools	31
3.3 Relational databases	33
4. DSML for developing repository-based Eclipse plug-ins	36
4.1 Requirements for DSML	36
4.2 Approach for defining DSML	37
4.3 Domain analysis	37
4.3.1 Eclipse concepts.....	38
4.3.2 Stylebase for Eclipse concepts.....	39
4.4 Concepts in DSML	40

5. Implementation of DSML.....	42
5.1 Workbench graph	44
5.2 Preference graph.....	46
5.3 Database graphs.....	47
5.4 View graph	50
5.5 Dialog graph.....	53
6. Implementation of the code generator	59
6.1 Code style	60
6.2 The structure of the code	60
6.3 The structure of the code generator	62
7. Case example: Stylebase for Eclipse	65
7.1 Case description.....	65
7.2 Modelling the Stylebase for Eclipse tool.....	67
7.2.1 HSQL database graph	68
7.2.2 MySQL database graph.....	69
7.2.3 Preference graph.....	70
7.2.4 View graph.....	70
7.2.5 Dialog graph.....	71
7.3 Generating the source code	73
7.4 Comparison of the MDD approach to the code-centred approach	74
8. Discussion.....	76
8.1 Experiences on DSML design	76
8.2 Experiences in DSML implementation	77
8.3 Evaluation of DSML	79
9. Conclusion	83
References.....	84

List of abbreviations

BAT	File extension for Microsoft Windows batch file
BNF	Backus-Naur Form, DSL for expressing context free grammars
CIM	Computation Independent Model, a model of a system that contains no information of the structure of systems
COM	Component Object Model, Microsoft's object-oriented programming model
DSL	Domain-Specific Language, a language that is specific to a certain application domain
DSM	Domain-Specific Modelling, a modelling paradigm that uses DSMLs
DSML	Domain-Specific Modelling Language, graphical DSL
EMF	Eclipse Modelling Framework, modelling and code generation framework in Eclipse
FAQ	Frequently Asked Questions
FCO	First Class Object, a first class entity in the GME metamodelling tool
GEF	Graphical Editing Framework, a framework for creating graphical editing plug-ins for Eclipse
GEMS	Generic Eclipse Modeling System, a metamodelling tool for Eclipse
GIF	Graphics Interchange Format, an image file format
GME	Generic Modelling Environment, a metamodelling tool
GMF	Graphical Modelling Framework, a framework for developing graphical editors for Eclipse based on EMF and GEF

GMT	Generative Modelling Technologies, an Eclipse project which aims at producing a set of prototypes for model-driven engineering
GOPRR	Graph-Object-Property-Port-Role-Relationship, a metamodeling language used in the MetaEdit+ metamodeling tool
HSQL	Hypersonic SQL, a project that has been used as the basis for HSQLDB
HSQLDB	A relational database management system implemented entirely in Java
HTML	HyperText Mark-up Language, a DSL for creating Web pages
IDE	Integrated Development Environment, a GUI workbench that combines the necessary tools for application development
IP	Internet Protocol, protocol that is used in the Internet to route packages
JAR	Java Archive, a tool for compressing files
JDT	Java Development Tools, a tool set in Eclipse for Java development
JPEG	Joint Photographic Experts Group, an image file format
MDA	Model Driven Architecture, OMGs initiative to provide a framework for MDD
MDD	Model-Driven Development, a software development approach where the focus is on models rather than code
MERL	MetaEdit+ Reporting Language, a language that is used for defining the code generator in the MetaEdit+ metamodeling tool
MOF	Meta-Object Facility, a standard defined by OMG for defining metamodels
MoSiS	Model-driven development of highly configurable embedded Software-intensive Systems, an ITEA 2 project
MVC	Model-View-Controller, an architectural pattern
MySQL	An open source relational database management system

OCL	Object Constraint Language, a language for defining constraints
OMG	Object Management Group, computer industry consortium that develops standards
PDE	Plug-in Development Environment, a tool set in Eclipse for developing Eclipse plug-ins
PIM	Platform Independent Model, a model of a system that contains no information specific to the implementation platform
PNG	Portable Network Graphics, an image file format
PSM	Platform Specific Model, a model of a system that contains information about the specific technology that is used in the implementation
RDBMS	Relational DataBase Management System, a database management system that is based on the relational model
SQL	Structured Query Language, DSL for performing relational database queries
SWT	Standard Widget Toolkit, widget toolkit for Java
TIFF	Tagged Image File Format, an image file format
UI	User Interface, point of communication between the computer and the user
UML	Unified Modelling Language, a general-purpose modelling language defined by OMG
XMI	XML Metadata Interchange, an OMG standard for serialising objects into XML
XML	eXtensible Mark-up Language, an open standard for exchanging structured data

1. Introduction

Modern-day software systems are constantly becoming more complex, which entails challenges to software development projects. Software needs to be developed at low cost but it also needs to be of high quality. To solve this paradox, new, more efficient, software development approaches need to be adopted. Traditionally, general-purpose modelling languages such as the Unified Modelling Language (UML) are used at some phase of the software development life cycle, but the full benefits of modelling are usually not achieved, since 100 percent code generation from general-purpose models is usually not possible [1 p. 117, 2]. Model-driven development (MDD) is about focusing on models rather than computer programs and automating the model-to-code transformation [3]. Domain-specific modelling is well suited to the MDD paradigm. Instead of using a modelling language that can model anything, domain-specific modelling is about using a domain-specific modelling language (DSML) that is restricted to a certain application domain. DSML uses *domain concepts* rather than programming language concepts for representing software models. Using DSML, full code generation from the models is possible, and the generated code typically has fewer errors than manually written code [2, 4].

Eclipse is an open source platform, which users can extend by writing plug-ins [5 p. 5]. Eclipse plug-ins are developed with the Java language by using extension points defined in the Eclipse platform. Although there are several wizards in Eclipse that help in the plug-in development process, building an Eclipse plug-in requires a lot of work, especially from a person not familiar with Eclipse technology or Java language. This is particularly so if several variant applications are developed for the same application domain with manual coding taking up a lot of development time. There are also several rules that should be obeyed while developing Eclipse plug-ins, and mastering all those rules may be difficult for the inexperienced plug-in developer [5 p. 15].

Previously, a few tools have been developed that are able to generate Eclipse plug-ins semi-automatically, such as the Generic Eclipse Modelling System (GEMS) [6], which is a tool for Eclipse that allows for the rapid development of graphical modelling plug-ins. GEMS enables the developers to specify the rules for DSML using a metamodel and the tool generates a graphical modelling plug-

in that enforces the rules from the metamodel. However, the structure of the generated plug-in is always similar and only the metamodel can change. It is not possible to add for example, one's own dialog boxes to the generated plug-in. Another tool that can generate graphical modelling Eclipse plug-ins is the Merlin Generator [7]. The Merlin Generator is an Eclipse plug-in, which is based on the Eclipse Modelling Framework (EMF) project [8]. The Merlin generator provides code generation and model transformation tools as well as a Graphical Editing Framework (GEF) [9] generator that can generate GEF plug-ins from a UML model. In GEMS and the Merlin Generator the main goal is not to automate Eclipse plug-in development and they do not address the generation of general Eclipse plug-in concepts, such as preference pages, dialog boxes and views. [6, 7]

Since there are already existing tools that help in the development of Eclipse plug-ins in the graphical modelling domain, in this work a slightly different approach and domain has been selected. The objective of this work is to develop prototype DSML and a code generator for developing several varying repository-based Eclipse plug-ins. In this work, a repository-based Eclipse plug-in means an Eclipse plug-in that utilises a database in a way that the end user does not have to know the physical structure of the database. Repository-based Eclipse plug-ins form a product family with several variation points:

1. Database type
2. Database content (tables, columns, column data types)
3. Operations to the database
4. How the information in the database is represented to the user.

The work includes the following steps:

1. **Selecting a suitable metamodeling tool.** A tool comparison is performed and a suitable tool is selected to support the construction of DSML and the code generator.
2. **Defining the scope of the domain.** The scope of the domain should be restricted to repository concepts and a small set of Eclipse UI concepts. The scope is influenced by the existing source code of the Stylebase for Eclipse plug-in, since the code generator has to be able to generate the source code of the tool.

- 3. Developing DSML (i.e. defining the syntax of DSML).**
 - a. Defining the domain concepts. This includes defining objects, their properties, and the relationships between the objects.
 - b. Defining the constraints for the language.
 - c. Defining symbols for the concepts.
- 4. Developing the code generator (i.e. defining the semantics of DSML).**

The code generator is defined by specifying which kind of source code corresponds to which concepts in the language. The development of the code generator is similar to the development of a compiler for a 3rd generation programming language.
- 5. Case example.** The case example that is used is the Stylebase for Eclipse architectural knowledge management open source tool developed at VTT [10, 11]. DSML and the code generator are used for generating the source code of the Stylebase for Eclipse plug-in. The variability binding times in the model-driven approach are compared with the traditional approach.
- 6. Publishing DSML and the code generator under an open source licence.**
 - a. Selecting a suitable open source licence.
 - b. Applying the licence to the metamodel and the code generator.
 - c. Publishing DSML and the code generator in the existing Stylebase for Eclipse open source community [11].

Defining DSML is done iteratively, that is steps 2–5 are repeated several times to build up the modelling language to its full strength.

The benefits of DSML and the code generator defined in this work are the raised level of abstraction and reduced development time for a repository-based Eclipse plug-in. DSML can be used to generate a large number of different repository-based plug-ins. DSML can also be extended in the future, if new features are needed.

2. Model-driven development of software product families

The software development process is a complex and challenging process to manage. Every stakeholder of a software product has specific requirements and expectations. Most software projects fail to deliver what has been promised on time and according to budget [1 p. 3]. New software development paradigms are emerging that aim to speed up the traditional code-centric software development process. Model-driven development (MDD) is such an approach that emphasises the importance of models in software development. MDD can be realised in many ways, in this chapter, the Model Driven Architecture (MDA) initiative and domain-specific modelling (DSM) are presented as a means to support MDD. Domain-specific modelling is particularly suited for a software product family situation, where similar applications are being built on the same problem domain.

2.1 Non-MDD development

Traditional code-centric software development often follows the well-known waterfall model, which was originally presented by Royce in [12] as a bad model for software development. The nature of the waterfall model is sequential, and the main phases of the model are requirements definition, design, coding, and testing. This approach usually includes the construction of some kind of models (e.g. UML) of the software in the design phase. Unfortunately, these models are not always utilised properly when the implementation starts and the final implementation may not even follow the original model of the system. Even if some code is automatically generated from the models, it is often just skeleton code, which needs to be completed manually. Of course it is useful to have the models as documentation of the software, but the models become easily outdated when changes are made to the code. Maintaining the models in synch with the code can be very time consuming and use up expensive resources in a software project.

2.2 Model-driven development

The key idea in model-driven development is to focus on models in software development (rather than computer programs). Fully automated MDD enables the generation of complete programs from models, which means that the models take the role of implementation languages. For MDD to be successful, the models must be abstract, understandable, accurate, predictive, and inexpensive. Abstraction helps in understanding a complex system by removing or hiding irrelevant details. The model that is left after removing the details must be easy to understand. Accuracy is also essential: the model must be a correct representation of the modelled problem. Predictiveness means that the model should predict the modelled system's interesting properties either through experimentation or through some type of formal analysis. The final property, inexpensiveness, is important because it makes no sense to construct models if the modelling costs more than the building of the modelled system. [3]

2.2.1 MDD with Model-Driven Architecture approach

Model-Driven Architecture is an Object Management Group (OMG) initiative, offering a conceptual framework to support MDD. The first definition document of MDA was released in 2001. The current version is 1.0.1 which is described in [13]. The foundations of MDA consist of three ideas: direct representation, automation, and open standards. Direct representation means shifting the focus of software development away from the technology domain towards the concepts of the problem domain. MDA defines the concept of platform as a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented [13]. MDA provides an approach for specifying a system independently of the platform with the Platform Independent Model (PIM), specifying platforms, choosing a particular platform for the system using the Platform Specific Model (PSM), and transforming the system specification into one for a particular platform (PIM to PSM transformation). [13, 14]

The aim in MDA is to automate the transformations between the different models of the system. Automation means using tools to mechanise software

development tasks that do not depend on human intelligence. For instance, if a model of the system is constructed and it needs to be translated manually to source code, there is no increase in development time and cost. Instead, automating the model-to-code transformation increases speed and reduces human error. [14]

Standards perform a key role in promoting automation. Open standards encourage vendors to produce tools and eliminate diversity. MDA includes several OMG standards that enable the model-driven approach, such as the Unified Modelling Language (UML), the XML Metadata Interchange (XMI), and the Meta-Object Facility (MOF). XMI facilitates interchange of models via XML documents. MOF is a standard, which enables metadata management and modelling language creation. It is commonly thought that UML is the basis of MDA, but it is actually MOF compliance that is formally required for the “MDA Compliant” label. However, in practice, UML is the key enabling technology for MDA and the basis of most development projects. [13–15]

2.2.2 MDD with domain-specific modelling approach

A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [16]. DSL raises the level of abstraction by using concepts familiar to domain experts [1 pp. 142–143]. There are countless numbers of popular DSLs such as BNF, HTML, and SQL to name but a few. In this work the term domain-specific modelling language (DSML) is used instead of domain-specific language (DSL). Domain-specific *modelling* language is a domain-specific language that can be used for constructing a graphical model of a software system. This definition also emphasises the MDD approach by restricting itself to graphical models and excluding programming languages.

Domain-specific modelling (DSM) is about using domain-specific modelling languages to create models, and generating code from the models with a code generator. The definition of DSML and the code generator should be done by a domain expert, who is familiar with the domain and able to produce high quality code for the domain. This process can be considered equal to the definition of a compiler for a 3rd generation language. When DSML and the code generator are

defined by a true expert, DSM speeds up the software development work and decreases the number of errors in software products. The DSM approach entails the initial work of defining DSML and the code generator, thus if only one application needs to be developed, it may be faster to use traditional methods. However, if several applications are continuously being built in the same problem domain (e.g. in a software product family), the benefits of DSM become greater than the effort required to define DSML and the code generator. This phenomenon is presented in Figure 1. This is the key issue to take into account when the adoption of domain-specific modelling is considered. After the initial effort of implementing DSML and the code generator, development of new applications is said to be 5–10 times faster than with traditional methods. [4, 17]

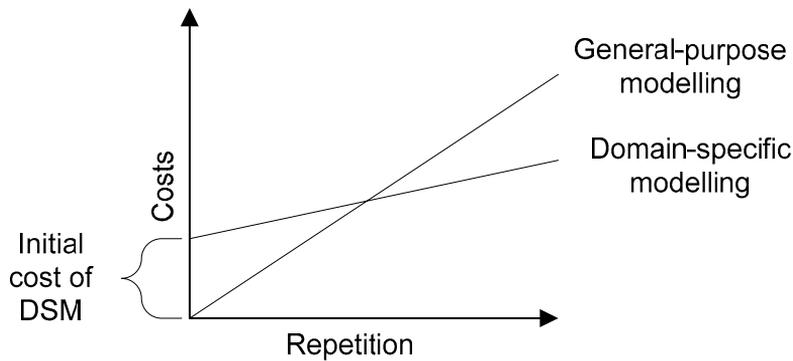


Figure 1. Domain-specific modelling versus general-purpose modelling.

A domain-specific language can be constructed by using metamodels or by using context-free grammar [18]. Usually graphical languages are developed by using metamodels and textual languages are defined by their grammar [19]. Since in this work, we aim to construct a graphical language, metamodeling is used. Simply put, a metamodel is a model of models [13]. A metamodel including the main domain concepts is first constructed to result in DSML and then DSML is used to model the actual problem. There are several approaches identified for defining the metamodel (i.e. DSML) [4]:

1. **Domain expert’s or developer’s concepts.** DSML is constructed from the domain concepts, for example, insurance products. This approach raises the level of abstraction far beyond programming language concepts.

2. **Generation output.** The design of DSML is driven by the required code structure, for example XML. This approach does not raise the level of abstraction very much and it is difficult to model behaviour.
3. **Look and feel of the system built.** If the design of the product can be understood by seeing, touching, or hearing, the end-user product concepts can be used as modelling constructs.
4. **Variability space.** If DSML is designed for modelling a product family, DSML can be constructed by expressing variability. Languages for static variability (configuration) are easy to define but behavioural variability is more challenging.

The definition of DSML and the code generator needs proper tool support, which is discussed in Section 3.2.

2.2.3 Comparison between MDA and DSM

Both MDA and DSM can be considered as a means for MDD and fundamentally they both aim at the same thing: speeding up the software development process and reducing the number of human-made errors. This is done by raising the level of abstraction from the solution domain to the problem domain. The aim is to have a model that is independent of the implementation platform.

MDA promotes more the use of open standards, whereas DSM tools often use proprietary languages. Often, companies do not want to commit to a certain tool vendor, so the open standards provided by the MDA approach may be more appealing than proprietary DSM tools. However, the tool interoperability promoted by MDA standards such as MOF and XMI is not fully achieved since many MDA tools use their own dialects of the standards [20]. There are also multiple versions of the standards, which creates even more diversity. Because of the many standards that need to be followed, the transition to using the MDA approach may be overwhelming. Adopting a proprietary DSM environment can thus be more appealing but the risks of committing to a certain tool vendor need to be accepted (e.g. tool vendor going out of business).

2.3 Software product families

As stated before, managing the complexity of modern software systems is a difficult task. Developing software products from scratch each time is not a feasible solution. Instead, reusing previously developed components should be considered. The software product family approach is about reusing software in a predefined way and thus making the software development process more effective. The term software product line is an alternate term to software product family. In this thesis, the term software product family is preferred. A software product family is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. It should be noted that software reuse as such does not form a product family. In the software family approach, the software reuse is planned, enabled, and enforced. The software product family approach can be divided in two phases: domain engineering and application engineering. Domain engineering means developing reusable core assets and application engineering means building complete products from those core assets. These phases may occur in either order: new products can be developed from existing core assets, or core assets can be extracted from existing products. [21 pp. 5–15]

A central concept that relates to software product families is software variability. Software variability is the ability of a software system or artefact to be efficiently extended, changed, customised, or configured for use in a particular context [22]. Variability can be expressed by using variation points. A variation point identifies one or more locations at which the variation will occur [23 p. 99]. For example, if an application needs to be localised for a number of languages, the language can be represented as a variation point and the applications in different languages would then be called variants.

In the past variability in software systems has been represented implicitly. In a modern-day software product family approach, an implicit representation of variability is not enough. A system-independent method for representing and normalising variability in the application engineering phase is presented in [24]. Variability is realised by a certain variability technique, e.g. makefiles or pre-processor directives. Variability realisation techniques are characterised by their introduction and binding phase in the software life cycle. A frame of reference is

presented, which uses variability sets for representing variability. Figure 2 presents the variability sets. A variability set relates to a certain combination of an introduction and binding phase. A variability realisation technique is a member of exactly one variability set and a variability set may consist of zero or more variability realisation techniques. For example, makefiles can be used as a variability realisation technique. If a variation point is introduced at the requirements phase and makefiles are used as the variability realisation technique, the variability is bound at the link phase so the corresponding variability set is (1,5). Maximum variability occurs in the variability set (7,7) which means that the variability is introduced and bound as late as possible. [24]

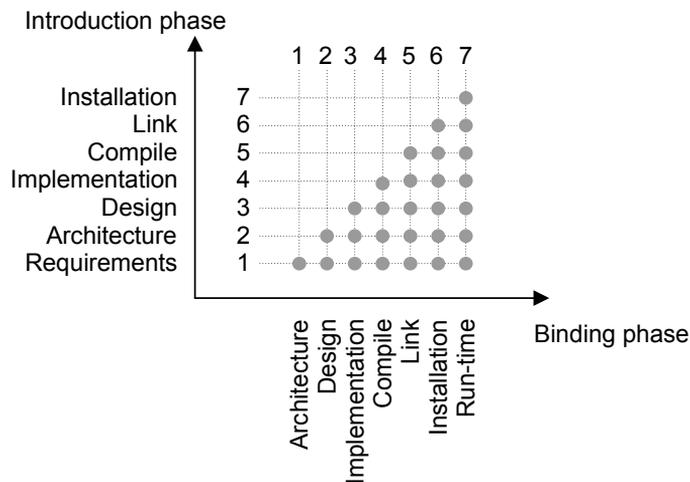


Figure 2. Variability sets in the software life cycle.

3. Tools and technologies

This section presents the tools and technologies that are used in this work. First, the Eclipse platform is presented. Second, metamodelling tools are introduced and compared, and a suitable tool for implementing DSML and the code generator is selected. Finally, a short introduction to relational database technology is provided.

3.1 Eclipse

Eclipse is an extensible platform for tool integration, and a set of basic tools built on that platform. Eclipse is also an open source project which delivers the Eclipse technology. The key word in Eclipse is interoperability, since nearly everything is an extension to the platform. The Eclipse platform includes a collection of extension points where the extensions can be plugged-in, which is why Eclipse extensions are commonly called plug-ins. The Eclipse community is large and it includes a wide range of people. The community can be thought of as a pyramid, which is illustrated in

Figure 3 [5 p. 2]. The largest group of people in the Eclipse community are the users, who use Eclipse as an integrated development environment (IDE). Second, there are the configurers, who customise Eclipse for example, by rearranging perspectives. Next, there are the extenders, who extend the functionality of Eclipse by writing their own extensions. An extender can become a publisher by publishing the extension to others. Enablers enable other people to extend their own extensions by providing an extension point to others. At the top of the Eclipse community are the committers, who take part in the development of the global Eclipse release. [5 p. 2]

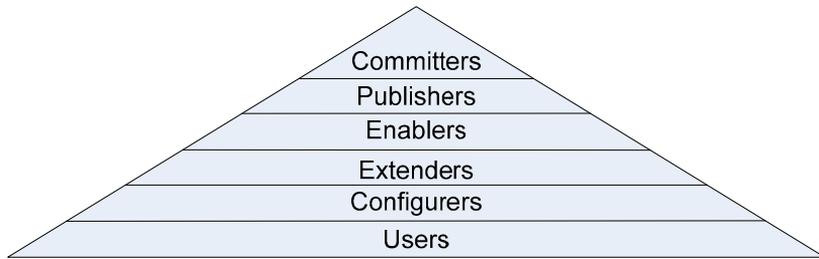


Figure 3. The Eclipse community.

The Eclipse environment is divided in three layers, which are presented in Figure 4 [5 p. 3]. The bottom layer is called the platform, which defines the common infrastructure of Eclipse. The middle layer, the Java Development Tools (JDT) layer, adds a Java integrated development environment to Eclipse. The Plug-in Development Environment (PDE) is the top layer and it extends JDT by providing support for developing plug-ins to Eclipse. [5 p. 3]



Figure 4. The three layers of Eclipse architecture.

Figure 5 shows an overview of the architecture of the platform layer [5 p. 6]. The platform is decomposed into the Core and the User Interface (UI) layers. The Core layer includes the Runtime component and the Workspace component. The Runtime component is responsible for discovering all available plug-ins and loading the plug-ins when needed. The Workspace manages the projects that the user creates. The Workspace is mapped directly to the file system and there is no intermediate repository. [5 p. 6, p. 295]

The UI layer consists of three components. The first component is the Standard Widget Toolkit (SWT), which is designed to provide efficient, portable access to the user interface facilities of the operating systems on which it is implemented [25]. If a widget is not available on a particular platform, SWT implements the widget in Java. SWT focuses on providing widgets, layouts, and event handling functionality. SWT can be used to create either stand-alone Java applications or Eclipse plug-ins [26 p. 130]. The second component is the JFace UI toolkit,

which provides higher level application support not provided by SWT. JFace covers a set of smaller frameworks for viewers, actions, dialogs, and wizards, among others. The third component in the UI layer is the Workbench, which utilises both the SWT and the JFace components. The Workbench defines the Eclipse UI paradigm, including views, editors, and perspectives. [5 pp. 325–345]

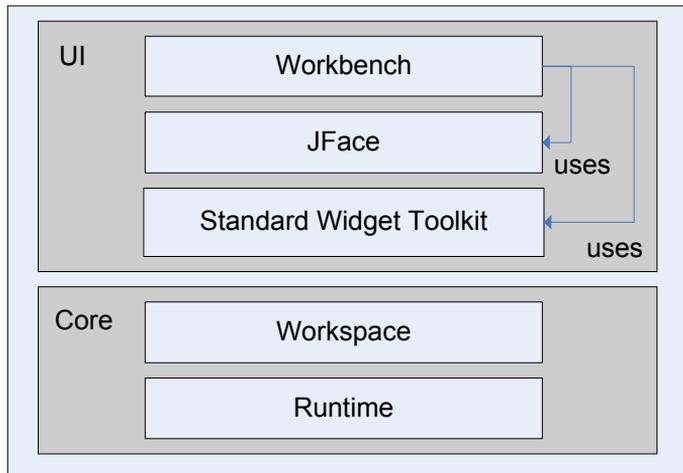


Figure 5. The structure of the Eclipse platform.

When Eclipse plug-ins are developed by hand, PDE automates most of the plug-in structure development. The main task of the plug-in developer is to write the Java source code for the plug-in. Since in this work, the aim is to develop a code generator able to generate an Eclipse plug-in, it is necessary to know the inner structure of an Eclipse plug-in.

A plug-in is represented as a directory. Figure 6 presents an example of the contents of a very simple plug-in directory. The compiled source code of the plug-in is contained in the com directory. The name of the com directory comes from the first part of the Java package name. The META-INF directory contains the MANIFEST.MF file, which describes the runtime aspects of the plug-in (such as identifier, version, and plug-in dependencies). The src directory is optional and it includes the Java source code of the plug-in. Eclipse consists of a very small kernel surrounded by hundreds of plug-ins. If the Java class files of each plug-in would be loaded, it would take far too long for Eclipse to start. That is why Eclipse applies the lazy loading rule, which means that plug-ins are only

loaded when needed. The plugin.xml file describes the extensions and extension points of the plug-in. The plugin.xml file enables the implementation of the lazy loading rule by providing the necessary information of the plug-in without loading the whole plug-in. If a plug-in provides extension points for other plug-ins, the plug-in directory contains a directory called schema, which contains XML descriptions of the details of each extension point. The plug-in directory can also contain resources (such as icons). The plug-in is installed to Eclipse by placing the plug-in directory into the plugins directory under the Eclipse home directory. The plug-in directory can be also compressed to a Java Archive (JAR) file. [5 p. 17, 26 pp. 101–104]

Name ▲	Size	Type	Date Modified
com		File Folder	30.8.2007 15:09
META-INF		File Folder	30.8.2007 15:09
src		File Folder	30.8.2007 15:09
plugin.xml	1 KB	XML Document	30.8.2007 15:09

Figure 6. The contents of a simple plug-in directory.

3.2 Metamodelling tools

The metamodelling tool is in an essential role in domain-specific modelling. This section discusses some of the available tool support for defining DSML and the code generator. The definition of the language means defining the syntax of the language (i.e. what kind of constructs the language contains and what their relations are). In the code generator, the domain model is interpreted and the constructs in the model gain a meaning, so the code generator provides the semantics for the language. The following things are considered when the tools are evaluated:

1. Tool provider
2. Supported platforms
3. Licence
4. Documentation and support
 - User's guide for the tool
 - Tutorials
 - Instructions for the code generator definition

- E-mail support
- 5. Metamodelling language
- 6. Constraint definition possibilities
- 7. Code generation possibilities
 - Generator definition language
 - Generator output language.

The metamodelling tool should be available for the Windows platform, have the required documentation and tutorials available, be easy-to-learn, be easy-to use (easy to define the metamodel and the code generator), and be able to generate Java code and all the other necessary structures of an Eclipse plug-in.

3.2.1 DSL Tools for Visual Studio

Microsoft's Domain-Specific Language Tools are available as a plug-in to Visual Studio 2005. Thus the DSL tools plug-in is a commercial tool and only available for the Windows platform. The DSL tools toolkit contains the metamodelling environment, modelling environment, as well as code generation support. Documentation is available online at [27]. [28]

The definition of the metamodel includes three steps [29]:

1. **Defining the domain model with a graphical designer.** The domain model notation represents the structure of the domain (i.e. the syntax of DSML). The metamodelling paradigm is similar to object-oriented programming, since there are classes with properties and the classes can be connected by embedding, inheritance, and reference.
2. **Defining the graphical domain constructs in XML.** The domain constructs are defined in an XML document. Managing XML documents takes a lot of development time, and no supporting tools exist [28].
3. **Defining the relationship between the domain model and the graphical constructs.**

There is also a DSL Designer wizard, which can be used for creating a fully configured solution. The wizard provides templates for languages (e.g. task flow diagrams and class diagrams). [29]

Constraints are expressed with the help of the Validation Framework and C# coding is required for writing the constraints. Code generation is based on text templates. The target languages for code generation are restricted to Visual Basic and C#. [18, 28]

3.2.2 Generic Modelling Environment

The Generic Modelling Environment (GME) is a configurable framework for creating DSM environments. GME has been developed at the Institute for Software Integrated Systems at Vanderbilt University. GME is available only for the Windows platform. GME is free and open source (licensed with the GME licence [30]). GME had been designed for flexibility and customisability. A user's guide and tutorials (also code generator definition tutorials) are available online at [31]. [32]

The metamodelling paradigm is based on UML. The main metamodelling concepts used by GME are Folders, Models, Atoms, Sets, References, Connections, Roles, Constraints, and Aspects. Models, Atoms, Sets, References, and Connections are all First-Class Objects (FCOs). A relational database is used for storing the metamodel. GME supports the visual drawing of the modelling concepts with a COM component called decorator. Simple bitmaps can be used as icons for the concepts but the default implementation of the decorator can be replaced by writing a proprietary COM-based component. [32–34]

Constraints are defined using a predicate expression language based on the Object Constraint Language (OCL). This language enables the representation of complex relational constraints and also rules for the containment hierarchy and the values of properties. The main focus in GME is the construction of a graphical modelling tool and the model interpretation is not the concern of GME. GME is component-based and the component model is COM, so the primary languages for model interpretation are C++ and Visual Basic, but any COM-enabled language can be used (e.g. Java and Python). There is also a report language which provides simple reporting capabilities through the definition captured in a simple text file. Output language is not restricted. [32, 34]

3.2.3 Generic Eclipse Modelling System

The Generic Eclipse Modelling System (GEMS) is a metamodeling tool for the Eclipse environment. GEMS is an open source project, which is licensed under the Eclipse Public License [35]. GEMS is a part of the Eclipse Generative Modelling Technologies (GMT) project [36]. GEMS is implemented as an Eclipse plug-in and it utilises several Eclipse projects, such as the Eclipse Modelling Framework [8], Draw2D, and GEF [9]. GEMS is also transitioning to utilise the Generic Modelling Framework (GMF) project [37]. Since GEMS is an Eclipse plug-in, it is available on any Java-enabled platform.

The metamodeling language in GEMS includes the following concepts: entity, attribute, connection, inheritance, and aspect. The metamodel is created graphically by creating entities and connecting them to each other. The modelling plug-in is created by invoking the DSML plug-in generator that is provided by GEMS. [38]

Constraints can be written in Java, OCL, and Prolog. Similarly to GME, GEMS is also mainly focused on creating the metamodel and the modelling plug-in. However, GEMS provides extension points for writing code generators. The generator is also implemented as an Eclipse plug-in so the generator definition language is Java. [39]

3.2.4 MetaEdit+

MetaEdit+ is a repository-based tool for developing domain-specific modelling languages and code generators. MetaEdit+ is a commercial tool and available for all the major platforms including Windows, Linux, Mac OS X, HP-UX, and Solaris [40]. Extensive documentation and tutorials are available online at [40].

MetaEdit+ employs the GOPRR metamodeling language, which is an extension to the GOPRR metamodeling language [41]. The name of the language comes from the names of the metatypes: Graph, Object, Property, Port, Relationship, and Role. The GOPRR language adds the concept of port to the GOPRR language. An object is a thing that exists on its own and it typically appears as a shape in a diagram. Relationship is a connection between two or

more objects and it typically appears as a line between objects. Role defines how an object participates in a relationship and it typically appears as the end point of a relationship: for example, an arrowhead. Port is a specific point in an object to which a role can connect, for example, an Amplifier object might have a port for analog input and a port for digital input. Graph is a diagram consisting of objects and their relationships (with specific roles). An object in a graph can be decomposed into a new graph or linked to other graphs via the explosion structure. An object can have zero or one decomposition graphs but several explosion graphs. Decomposition and explosion structures enable the layering of DSML. All five of the above mentioned metatypes can have properties and properties can only be accessed as parts of the other metatypes. MetaEdit+ includes a tool for each of the metatypes. In the object tool, for example, it is possible to create a new object or edit the properties of an existing object. [40–42]

MetaEdit+ provides two metamodelling possibilities: form-based metamodelling and graphical metamodelling. A graphical metamodel is described as a diagram using a visual notation for the GOPPRR language. Graphical metamodelling is suitable only for simple languages. Form-based metamodelling uses the tools that are defined for the metatypes for creating new types and defining connections between them. Form-based metamodelling provides more precision and better scalability. [42]

MetaEdit+ provides multiple constraint definition possibilities. When graphs are defined with the Graph tool, relationship cardinalities can be defined in the Graph Bindings tool. Additional constraints for graphs can be defined with the Graph Constraints tool. MetaEdit+ enables the provision of many types of constraints: connectivity constraints (how many times an object may participate in a relationship), occurrence constraints (how many times an object may occur in a graph), uniqueness constraints (objects of the same type must have a unique value for a property) and ports property constraints (port properties must have the same or a different value). Also, regular expressions can be used for constraining user input. [40]

MetaEdit+ supports code generation from the domain-specific models with the Generator Editor tool. Code generators are defined in the Generator Editor tool using the proprietary MetaEdit+ Reporting Language (MERL). The MERL language is a simple scripting language with commands for navigating through

design models, extracting information from design elements and outputting it to a window or a file. [40]

3.2.5 Comparison of the tools

Table 1 presents a summary of the evaluated tools. All the tools are available for the Windows environment, but only GEMS and MetaEdit+ are available for several platforms. Consequently, the platform does not restrict the selection of the metamodelling tool in any way, since the required platform is Windows.

The DSL tools environment has all the necessary documentation available. Yet, the learning curve of DSL tools is said to be steep [43]. The output languages in code generator are Visual Basic and C#, and Java is the required output language in this work, which rules out the DSL tools environment.

GME is mostly aimed at the development of a DSML editor and code generation support is not the main concern [34]. The UML-based metamodelling paradigm is not very appealing. GME is said to be suitable for simple prototyping of DSML editors [33]. In this work, a sophisticated and mature metamodelling tool is required with good code generation support, so GME is also excluded.

The GEMS environment is an open source tool that has the necessary documentation and tutorials available. However, there is not enough support for code generation, since only extension points are provided for accessing the models. That is why GEMS is not selected as the metamodelling tool either.

MetaEdit+ is a tool with a proprietary metamodelling paradigm and generator definition language. Since MetaEdit+ tool is not tied to UML in any way, it offers very flexible possibilities for DSML definition. The definition of DSML does not require any manual coding and the code generation support is very good. The output language of the generator is not restricted to any specific language and there is also a generator debugger. The documentation and support in MetaEdit+ is very good and the learning curve for the tool seems reasonable. MetaEdit+ fits the requirements of this work, so MetaEdit+ is selected as the metamodelling tool for this work.

Table 1. DSM tool comparison.

		DSL tools	GME	GEMS	MetaEdit+
Provider		Microsoft	Vanderbilt university	Eclipse GMT project	MetaCase
Platform		Windows	Windows	Java-enabled platforms	Windows, Linux, Mac, HP, Solaris
Licence		Commercial	GME licence	EPL	Commercial
Documentation and support	User's guide	Yes	Yes	Yes	Yes
	Tutorials	Yes	Yes	Yes	Yes
	Instructions for the code generator definition	Yes	Yes	Yes	Yes
	E-mail support	Yes	Yes	Yes	Yes
Metamodelling language		Object-oriented	Based on UML	Ecore	GOPRR
Constraint definition language		C#	OCL	Many languages	Restricted set of constraint definition possibilities
Code generation	Generator definition Language	Templates for generating	C++ , Visual Basic, COM enabled languages	Java	MERL language
	Output language	Visual Basic, C#	Any language	Any language	Any language

3.3 Relational databases

Database is a collection of related data [44 p. 2]. There are several database models, such as the hierarchical, network and relational models. This work utilises the relational database model. The most fundamental concept of a relational database is the table, which consists of rows and columns. Each column specifies a field, which has a data type, for example varchar or int. Rows contain different values for the fields specified by columns. A database table usually has a primary key, which consists of one or more columns. A primary key is used to identify a row in the database, so in each row the primary key must have a unique value. A table may also have a foreign key, which references a column in another database table. [44 p. 6]

SQL is a domain-specific language, which is used to manage and interact with data in a relational database. SQL is made up of a set of statements that define the structure of a database, store and manage data within that structure, and control access to the data. The most common SQL operations in a relational database are insert, select, update, and delete. [44 p. 14]

The insert statement is used to insert data to a table. Presented below is an example of an insert statement:

```
INSERT INTO table (column [,...]) VALUES (value [,...])
```

The update statement is used to update a value in the database. Presented below is an example of an update statement:

```
UPDATE table SET column = new_value WHERE column operator value
```

The where clause at the end of the update statement can be used to specify a condition for the update operation. Only those rows fulfilling the condition are returned in the result set. The condition of the where clause is formed by using operators; some of the operators that can be used are presented in Table 2. [44 p. 278]

Table 2. Operators in the where clause.

Attribute	Meaning
=	Evaluates to true if both arguments are equal
<>	Evaluates to true if both arguments are not equal
>	Evaluates to true if the value of the first argument is greater than the value of the second argument
<	Evaluates to true if the value of the first argument is less than the value of the second argument
>=	Evaluates to true if the value of the first argument is greater than or equal to the value of the second argument
<=	Evaluates to true if the value of the first argument is less than or equal to the value of the second argument
LIKE	Searches for values similar to a specified value

The delete statement deletes all the rows in the database which fulfil the condition given in the where-part of the statement. The syntax of the delete statement is as follows:

```
DELETE FROM table WHERE column operator value
```

The update, insert, and delete statements do not return any data from the database. For retrieving data from the database, the select statement is used. The syntax of the select statement is the following:

```
SELECT column FROM table WHERE column operator value
```

In this work, two relational database management systems (RDBMS) are utilised, namely MySQL and HSQLDB. MySQL is a popular, open source RDBMS developed by MySQL AB. MySQL runs on multiple operating systems. MySQL requires installation before it can be used, so it is not transparent to the user of the repository-based application. [44 p. 1]

HSQLDB (HSQL database) is lightweight RDBMS, which is implemented with the Java language. HSQLDB can be run in server mode or in-process mode.

Server mode means that the database can be accessed over the Internet, whereas the in-process mode means that the database is used only locally. HSQLDB can be totally transparent to the application end-user: it can be started from a Java application and the user does not have to use any database management tools. [45]

4. DSML for developing repository-based Eclipse plug-ins

This chapter describes the design of the domain-specific modelling language for developing repository-based Eclipse plug-ins. First, some requirements are set for DSML. Second, the approach for defining DSML is selected. Then domain analysis is made to find out the relevant domain concepts. Finally, the scope of DSML is stated.

4.1 Requirements for DSML

Table 3 presents the requirements that the domain-specific modelling language should satisfy [1 pp. 142–143, 40]. The requirements are given IDs from R1 to R5. R1 means that the scope of the domain should be clearly declared so that it is easy to decide whether DSML answers a certain need. R2 emphasises that familiar concepts should be used so that people who are familiar with Eclipse would understand the language concepts with reasonable effort. R3 says that DSML should have easy-to-use notation, which is important because it has to be easier to use DSML than to develop the plug-in using Java language. R4 ensures that the user is not able to construct incorrect models with DSML. An important aspect when designing DSML is to make sure that DSML can be extended, since in the future there may be a need for new features or even the domain itself may need to be extended. The extensibility requirement is stated in requirement R5. This can be realised by introducing multiple layers to the language. It is possible to present a simple language in a single layer, but when the number of language concepts starts to grow, multiple layers become necessary.

Table 3. Requirements for the domain-specific modelling language.

ID	Requirement
R1	The domain of DSML should be explicitly stated
R2	The language concepts must be understood by people familiar with the domain
R3	The notation should be easy to use
R4	DSML should have well-defined constraints
R5	It should be possible to extend DSML (i.e. DSML should be composed of multiple layers)

4.2 Approach for defining DSML

The target platform in this work is Eclipse, which forms a solid foundation for the modelling concepts. There is a restricted set of building blocks for Eclipse plug-ins, and Eclipse plug-ins have a familiar look and feel. The developed Eclipse plug-ins are also repository-based, so some repository related concepts need to be included in DSML. Approaches for defining DSML were introduced in Section 2.2.2. In this work approaches number one (domain expert's or developer's concepts) and number three (look and feel of the system built) are applied in the definition of DSML. The challenge in defining DSML with approach number three is relating other types of modelling elements and constraints to the look and feel concepts [4]. Also the variability space approach is considered when the scope of the domain is defined.

DSML is developed iteratively, which means that first just a few modelling concepts are defined and some code is generated from them. Each iteration adds some concepts to the language and extends the code generator to support the new features.

4.3 Domain analysis

The purpose of the domain analysis is to find the relevant concepts in the domain that can be used in the construction of the domain-specific modelling language. Because DSML has to be able to generate the source code of the Stylebase for Eclipse plug-in, it is important that the Eclipse concepts used by the Stylebase for Eclipse tool are included in DSML. Some other Eclipse concepts not included in the Stylebase for Eclipse tool can also be included in DSML, but the first priority is to include the concepts necessary to generate the Stylebase for Eclipse plug-in. It is also necessary to identify the database operations that are needed for fulfilling the functionality of the Stylebase for Eclipse tool.

4.3.1 Eclipse concepts

Figure 7 presents a screenshot from Eclipse with some Eclipse concepts pointed out. Two basic concepts in Eclipse are the view (number 1 in Figure 7) and the editor (2). The combination of various views and editors visible within the Eclipse workbench are known as a perspective. Multiple perspectives can be open at one time, but only one is visible. Every perspective has its own set of views but open editors are shared by all open perspectives. Only a single instance of a view can be open in a perspective, while any number of editors of the same type (e.g. Java editors) can be open at the same time in a perspective. Views are used for navigating resources and modifying the properties of a resource. A view can contain a view toolbar (3) and a view pull-down menu (4). A toolbar consists of toolbar buttons (5). A view can also contain a context menu (6), which pops up when the right mouse button is clicked in the view. Any changes made within a view are saved immediately, whereas editors follow the open-save-close model. The user can select a certain view using the show view dialog, which is opened from the Window menu in the menu bar (7). In order to reduce the visual clutter in the show view dialog, views should be grouped using view categories [46]. Plug-ins can contribute actions in many places such as the main menu bar (7), toolbar (8), the view toolbar (3), the view pull-down menu (4), and context menus (6). [5 p. 127]

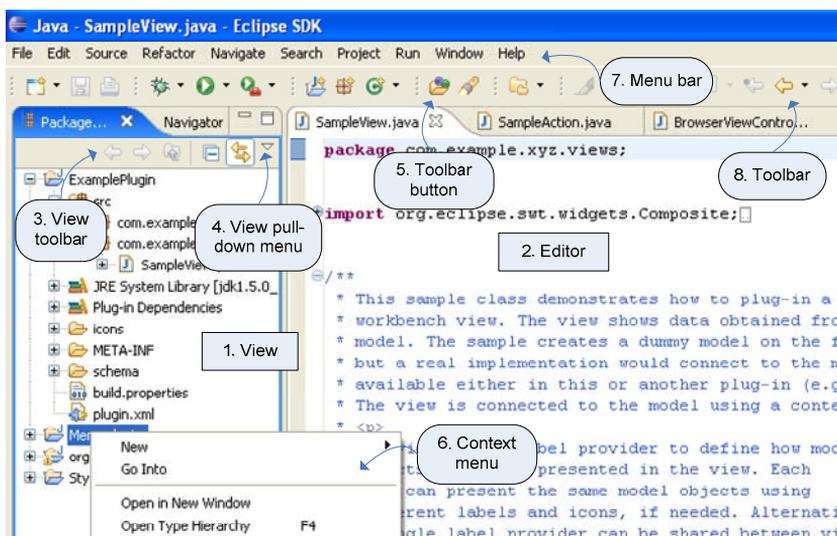


Figure 7. Eclipse user interface concepts.

Plug-ins can provide preference pages that contain information needed by the plug-in when it executes. The user can configure the execution of the plug-in by changing the values in the preference pages, and the information stored in the preferences is saved across multiple Eclipse sessions. The contents of a preference page can be defined by creating SWT widgets, but there are also helper classes, called field editors, that create the widgets and implement the value setting and retrieval code for the most common preference types [46]. [26 p. 451]

4.3.2 Stylebase for Eclipse concepts

Since the case example that is used in this work is the Stylebase for Eclipse product, the relevant Eclipse concepts used by the Stylebase tool need to be identified. The Stylebase for Eclipse architecture knowledge management tool uses an HSQL and a MySQL database to store design and architectural patterns. The tool uses Eclipse views to present information in the databases. Figure 8 presents a screenshot of the Stylebase tool. The main view is called Stylebase and it presents the patterns in the database. In the upper right corner of the view there are toolbar buttons (number 1 in Figure 8), which perform different actions to the database when selected. The view has also a context menu, which contains actions related to a pattern in the database. In Figure 8, the content of a remote MySQL database is presented. When the leftmost toolbar button (number 2 in Figure 8) is selected the view switches to show the content of the HSQL database. In addition to the Stylebase view, the Stylebase tool contributes the Pattern diagram and the Pattern guide views, which are used for rendering the image of a pattern and the HTML description of a pattern, respectively. The Stylebase plug-in does not provide any custom editors. In the Eclipse preferences dialog, the Stylebase for Eclipse tool contributes database related preferences as well as information about the file types of the pattern files.

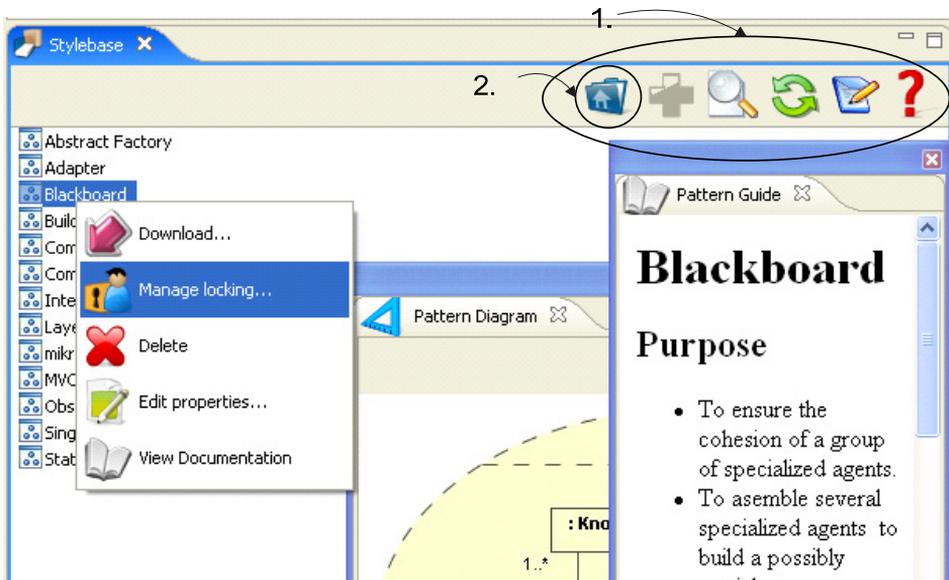


Figure 8. The Stylebase for Eclipse tool.

Since the selected domain is repository-based Eclipse plug-ins, the structure of the repository and the operations to the repository need to be modelled with DSML. The basic concepts and operations of relational databases were discussed in Section 3.3. The concepts of database table and database column are enough for representing the structure of the database. The Stylebase for Eclipse tool needs all the basic database operations to operate. Also operations for transferring data between the file system and the databases are needed, as well as transferring data directly between the two databases.

4.4 Concepts in DSML

In the previous section, the possible domain concepts were analysed. From these concepts, a restricted set of concepts need to be selected. The selection was based on the Stylebase for Eclipse domain analysis i.e. which concepts are needed to model the Stylebase tool. The selected concepts are presented in Figure 9. First, there is the repository, which contains the data. Second, there are the Eclipse views that are used for representing data in the repository. Third, there are the dialog boxes that are used for collecting user input and performing

operations to the database. Preference pages can be defined that contain database parameters and values that can be used to initialise dialog text fields. It is not possible to define one's own custom editors with this DSML.

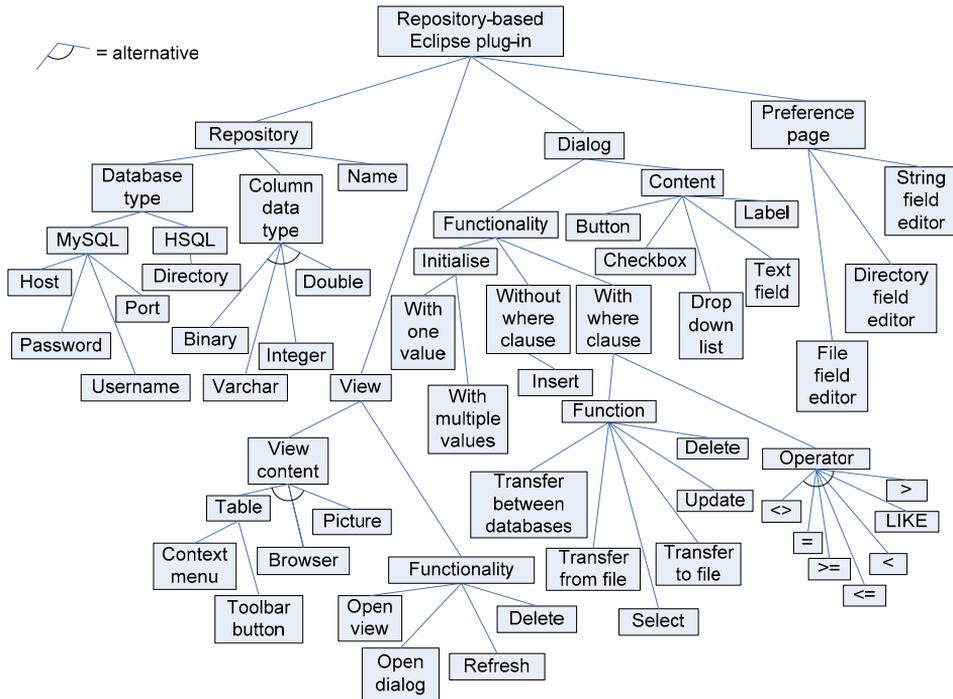


Figure 9. The selected concepts for DSML.

5. Implementation of DSML

In this chapter the implementation of the domain-specific modelling language is described. As stated in Section 3.2.4, MetaEdit+ provides two alternatives for the definition of a domain-specific modelling language: form-based metamodelling and graphical metamodelling. In this work, form-based metamodelling was used because of the better precision over graphical metamodelling. In form-based metamodelling, there is a proprietary tool for defining each of the metatypes of the GOPRR language (Graph tool, Object tool, Port tool, etc.). Each object, relationship, role, and property is defined in their respective tools and then the concepts are bound together in the Graph tool. In the Graph tool, the following tasks are carried out:

1. Defining the name and properties of the graph.
2. Declaring which previously defined objects, relationships, and roles may appear in the graph.
3. Defining which relationships can appear between which objects. This includes also specifying the roles of each object in the relationship. Also cardinalities can be assigned to the roles which specify how many times a role may be used in the relationship.
4. Defining possible decomposition and explosion graphs for the objects in the graph (in this work, only the decomposition structure is used, which means that the details of an object are described with a proprietary graph).
5. Defining constraints for the graph (e.g. how many times an object may occur in the graph).

The finalisation of the language includes also defining the graphical symbols for the objects, roles, and relationships. The symbols are defined with the Symbol Editor tool, which can be opened from the Object, Role, or Relationship tool. The definition of symbols is a very important task, since it has a major effect on the usability of DSML.

Figure 10 illustrates the structure of DSML. DSML consists of six graphs (presented as grey rectangles in Figure 10): the Workbench graph, the HSQL database graph, the MySQL database graph, the View graph, the Preference graph, and

the Dialog graph. The Workbench graph is the top-level graph of DSML. The five rounded rectangles under the Workbench graph rectangle present the objects that can be used in the Workbench graph. An object can be decomposed into a graph, which forms the basis for the layering of DSML. The decompositions of the objects in the Workbench graph are presented as rectangles under the rounded rectangles. It is important to use constraints so that the modeller is not able to create an incorrect model, which was also stated in the requirements for DSML (R4). The numbers above the objects describe how many times an object may occur in the graph. If there is no number above the object, it means that the object can occur only once. For those objects, the occurrence constraint is strict and can not be any other number, e.g. a Dialog graph can not contain more than one Dialog box object. For those objects that can occur more than once, the upper limit of occurrences is set to some fixed number, just for practicality. For example, there can be only five view categories because it usually does not make sense to use too many categories for a single plug-in.

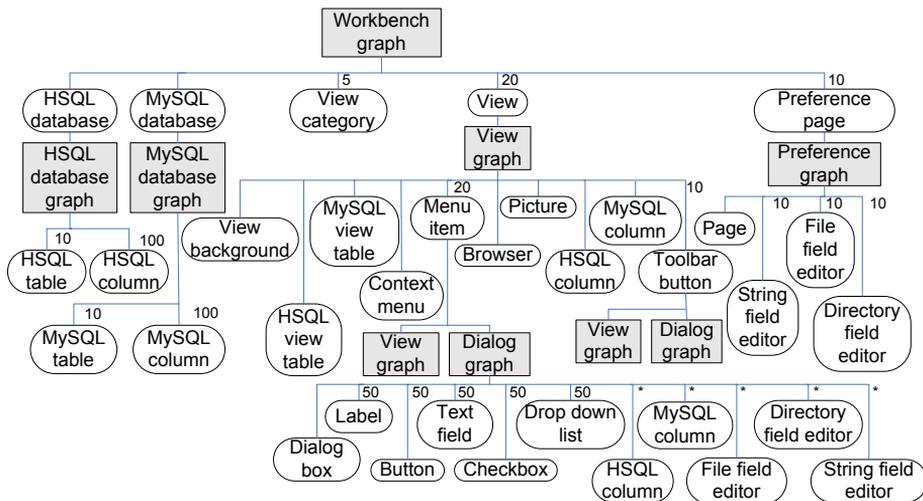


Figure 10. Overview of DSML.

The six graphs of DSML are described in the following sections, starting from the top-level graph (i.e. Workbench graph) and continuing to the decomposition graphs of the top-level graph. All the objects and their decompositions in the graph are explained, the properties of the graph are explained and the relationships between the objects are described. If there are some special constraints for relationships, they are presented when the relationships are described.

5.1 Workbench graph

Because the user interface of Eclipse is called the Workbench, the top-level graph in DSML is called the Workbench graph. The purpose of the Workbench graph is to declare the existence of views, view categories, databases, and preference pages. The Workbench graph is purely static, so there are no behaviour definition possibilities. Table 4 presents the objects that can appear in the Workbench graph.

Table 4. Objects in the Workbench graph.

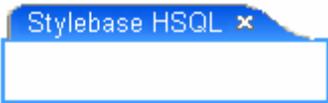
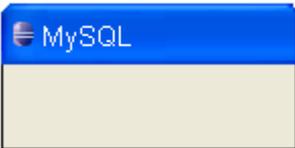
Object	Symbol	Description
HSQL database		Declares that the plug-in uses an HSQL database. Must be decomposed into an HSQL database graph, which describes the structure of the database.
MySQL database		Declares that the plug-in uses a MySQL database. Must be decomposed into a MySQL database graph, which describes the structure of the database.
View category		Declares the existence of a view category.
View		Declares the existence of a view. Must be decomposed into a View graph, which describes the content of the view.
Preference page		Declares the existence of a preference page. Must be decomposed into a Preference graph, which describes the content of the preference page.

Table 5 presents the properties of the Workbench graph. The plug-in uses a log file for logging some events at run time. The path of the log file can be specified at run time by using the Log file preference property or at the modelling phase with the Log file path constant property. Regular expressions are used for constraining user input.

Table 5. Properties of the Workbench graph.

Property name	Property data type	Regular expression for property value	Description
Destination directory	String	[C-Z]:\ [A-Z a-z\\]* (the expression can not make sure that the directory is an existing directory; it just checks the correct format)	The directory to which the code generator generates the plug-ins (most convenient to set to Eclipse plugins directory so that the generated plug-in is ready to use when Eclipse is restarted).
Log file path constant	String	None	A path to a file where the plug-in can write events at run time.
Log file preference	File field editor	None	A file field editor in preference pages from which the file path is fetched at run time.
Package name	String	[a-z]+([\.[a-z]+)*	The root package for the source code of the plug-in.

Although it is not possible to define any behaviour in the Workbench graph, there are two relationships that are used for connecting the objects in the graph. These relationships are presented in Table 6. The rectangles in the images in Table 6 present the objects that participate in the relationship. The relationships are presented as dots in the middle and the roles that are connected to the relationships are represented as lines between the relationship dot and the participating object. The name of a role is presented next to the role line and the cardinality of a role is at the end of the name. If the cardinality is one, it is not presented. The same notation is used for describing all the relationships in DSML.

Table 6. Relationships in the Workbench graph.

Relationship	Description
	Specifies which views belong to which category (it is not mandatory for a view to belong to a category, since there is the Other category in Eclipse that shows views that do not belong to any category).
	Specifies which preference page is shown as the top-level preference page in the Eclipse Preferences dialog.

5.2 Preference graph

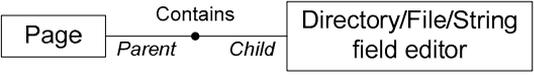
The purpose of the Preference graph is to describe the structure of a preference page in the Eclipse Preferences dialog. Table 7 presents the objects that can appear in the Preference graph. Only field editor preference pages can be created with DSML so the Preference graph includes three different types of field editors and a background object for the field editors. The Preference graph has only one property called the Page name. Each Preference graph instance can contain one Page object and a maximum of ten of each field editor.

Table 7. Objects in the Preference graph.

Object	Symbol	Description
Page		The Page object is the background for the field editors and its purpose is to increase the level of abstraction because the symbol resembles a real preference page in Eclipse.
Directory field editor	Pattern root <input type="text" value="H:\StylebasePatternRoot"/> <input type="button" value="Browse"/>	A field editor which accepts a directory as an input.
File field editor	Log file <input type="text" value="C:\Log.txt"/> <input type="button" value="Browse"/>	A field editor which accepts a file as an input.
String field editor	Database Name <input type="text" value="stylebase"/>	A field editor which accepts a String as an input.

The Page object should be connected to different field editors with the Contains relationship, which is presented in Table 8. The Contains relationship is the only relationship in the Preference graph, because the Preference graph contains only static information, which is reused in other graphs.

Table 8. The only relationship in the Preference graph.

Relationship	Description
	Connects different field editors to the Page object.

5.3 Database graphs

There are two database graphs in DSML: the HSQL database graph and the MySQL database graph. Both graphs have a similar structure but differ in the graph properties. The purpose of a database graph is to define the structure of the database. Table 9 presents the objects that can be used in the HSQL database graph.

Table 9. Objects in the HSQL database graph.

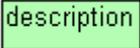
Object	Symbol	Description
HSQL table		A table in the HSQL database.
HSQL column		A column in the HSQL database.

Table 10 presents the objects that can be used in the MySQL database graph. As can be seen from Table 10, MySQL database has a similar structure as the HSQL database graph.

Table 10. Objects in the MySQL database graph.

Object	Symbol	Description
MySQL table	Table: patterns	A table in the MySQL database.
MySQL column	description	A column in the MySQL database.

The column objects in the database graphs can be reused in the View and Dialog graphs (see Sections 5.4 and 5.5) when functionality is defined. The column objects have a property called Home graph, which should be a reference to the database graph where the column is defined. The column objects also have a similar reference property to the table to which they belong. This makes it possible to access the information in the database graphs also from the other graphs where column objects are reused. The column objects have also a property for the name and data type of the column. The data type can be integer, double, varchar, or binary. There is only one relationship in the database graphs, which is presented in Table 11.

Table 11. The only relationship in the Database graphs.

Relationship	Description
	Connects database columns to the table in which they belong. Since only the Primary key and the Table roles are mandatory, a table may consist entirely of primary key columns.

There are some restrictions to the table and column names in a database. A regular expression is used in constraining user input for the name property of the table and the column objects. The expression that is used is `[A-Za-z][A-Za-z0-9_]*`, which means that the name can consist of just letters, numbers, and underscores, but can not begin with an underscore. It is also constrained such that a column and a table can be in at most one Contains relationship. Each column in a database table should also have a unique name. It is possible in MetaEdit+ to specify that a property must have a unique value, but then each column that is defined, has to

have a unique name (also columns in different tables). Thereby, the column name uniqueness in a table is not controlled in the DSML level, but checked in the code generator instead.

The main difference between the databases is the transparency: the MySQL database is not transparent to the end user as the HSQL database. The structure of the MySQL database needs to be modelled in the MySQL database graph, but the actual database tables need to be created with a separate MySQL administration tool. If an HSQL database is used, the user does not have to create the database since all the database management work is done by the generated plug-in. A MySQL database can also be remote or local, but an HSQL database can only be local. These differences do not show in DSML, but the difference in the database parameters does. The database parameters are specified with the graph properties. The HSQL database needs two parameters to operate: the directory in the local file system where the database is located, and the name of the database. Table 12 presents the properties of the HSQL database graph. The directory and the name can be specified either by using field editors in preference pages or by using a constant value. Thus the parameters can be specified either at run-time, or already at the modelling phase. If all properties are set, the field editors in the preference pages are preferred.

Table 12. Properties of the HSQL database graph.

Property name	Property data type	Regular expression for property value
Database directory preference	Directory field editor	None
Database name preference	String field editor	None
Database directory constant	String	[C-Z]\:\\\\[A-Z a-z\\]*
Database name constant	String	[A-Za-z][A-Za-z0-9_]*

Table 13 presents the properties of the MySQL database graph. The MySQL database needs five parameters: IP-address of the host, name, port, user name, and password. Each of these can be specified either by a String field editor from a Preference graph, or by a string or number constant. Primarily, the database parameters are fetched from the preferences, and secondarily from the constants.

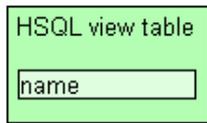
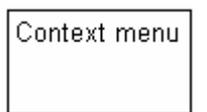
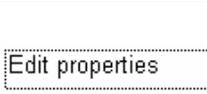
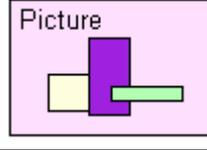
Table 13. Properties of the MySQL database graph.

Property name	Property data type	Regular expression for property value
Database host preference	String field editor	None
Database name preference	String field editor	None
Database port preference	String field editor	None
Database user name preference	String field editor	None
Database password preference	String field editor	None
Database host constant	String	None
Database name constant	String	[A-Za-z][A-Za-z0-9_]*
Database port constant	Number	None
Database user name constant	String	None
Database password constant	String	None

5.4 View graph

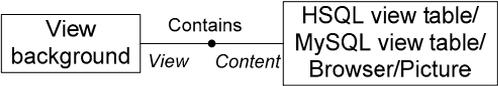
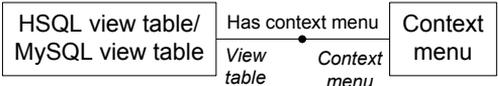
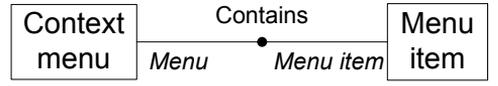
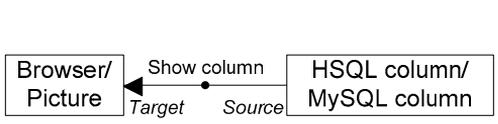
The purpose of the View graph is to describe the contents and actions in an Eclipse view. Table 14 presents the objects in the View graph. The purpose is that the HSQL and MySQL column objects that are used in the View and Dialog graphs are defined in the database graphs and reused in the View and Dialog graphs. This means that only existing column objects should be added to the View and Dialog graphs. It can not be constrained in the MetaEdit+ tool that only existing column objects can be added to a graph. However, it is possible to add also new column objects and generate a working plug-in, as long as the new column object has the same properties as an existing column object defined in a database graph.

Table 14. Objects in the View graph.

Object	Symbol	Description
View background		Background object for the other objects in the view graph.
HSQL view table		Specifies that the view shows information from an HSQL database. The name property of this object specifies which database column is shown as a table item in the view.
MySQL view table		Specifies that the view shows information from a MySQL database. The name property of this object specifies which database column is shown as a table item in the view.
Context menu		Context menu for an HSQL or MySQL view table.
Menu item		Menu item in the context menu. Can be decomposed into a View or a Dialog graph, which specifies what kind of view or dialog is opened when the menu item is selected.
Toolbar button		A toolbar button in the view toolbar. Can be decomposed into a View or a Dialog graph, which specifies what kind of view or dialog is opened when the toolbar button is pressed.
Browser		Renders an HTML document in the view.
Picture		Renders an image in the view. The image can be in GIF, JPEG, PNG, BMP, or TIFF format.
HSQL column	Presented in Table 9	Can be used for specifying which database column is shown in a browser or a picture.
MySQL column	Presented in Table 10	

The relationships that can be used in the View graph are presented in Table 15. A View graph instance may contain both an HSQL view table and a MySQL view table object, but only one of them can be connected to the View object, which means that a view can be used to show information either from an HSQL database table, or a MySQL database table, not both. A view that contains a browser or a picture has to be opened from another view that contains a view table so that content is shown in the browser or picture.

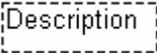
Table 15. The relationships in the View graph.

Relationship	Description
	Specifies the content of the view.
	Specifies that a view table has a context menu.
	Connects menu items to the context menu.
	Specifies the column that is shown in the browser or picture (the column has to be from the same database table which is shown in the view from which the Browser or Picture view is opened).
	Specifies that the menu item deletes the selected table item from the database.
	Reloads all the table items from the database.

5.5 Dialog graph

The purpose of the Dialog graph is to specify a dialog box that can show information in the database, accept user input and possibly perform some actions to the database. Table 16 presents the objects that can appear in the Dialog graph. In the code level, GridLayout is used for the layout of the dialog. The GridLayout lays out the widgets in rows and columns, so that each row has the same number of widgets. GridLayout lays out the widgets in the dialog in one column by default. That is why the Dialog box object contains a property called Number of columns, which makes it possible to control the layout of the dialog.

Table 16. Objects in the Dialog graph.

Object	Symbol	Description
Dialog box		Background object for the widgets in the dialog. There can be only one Dialog box object in the Dialog graph.
Checkbox		A Boolean type widget.
Drop down list		A widget that contains a list of Strings.
Label		A plain text widget which can not be edited.
Text field		A text field widget which can show text and accept user input.
Button		A button that triggers an action.
HSQL column	Presented in Table 9	Are used for specifying the targets of the database operations.
MySQL column	Presented in Table 10	
Directory field editor	Presented in Table 7	Can be used to initialise Text field widgets.
File field editor		
String field editor		

The majority of the functionality is defined in the Dialog graph, which contains eleven different relationships. Four of the relationships are binary relationships, which are presented in Table 17. A Button can not be in more than one Widget role, which means that it can be connected to the Dialog box object either with the Contains relationship, or with the Is default button relationship (not both).

Table 17. Binary relationships in the Dialog graph.

Relationship structure	Description
	Connects the different widgets to the Dialog box. Widgets can take part in only one Contains relationship.
	Defines which button in the dialog is the default button that is initially selected. A Dialog box may take part in only one Is default button relationship.
	Initialises a text field with a single value from preferences or from the database table that is shown in the view from which the dialog box was opened.
	Initialises a drop down list with all the values from a certain database column.

In addition to the binary relationships, there are also so called n-ary relationships in the Dialog graph, which have more than two participating objects. Figure 11 presents the objects that can participate in the Insert relationship, which is used for inserting a new row to the database. The Trigger role specifies the button in the dialog box, which triggers the Insert action. All the n-ary relationships in the Dialog graph have the Button object in the Trigger role. The Value to be inserted role specifies the text field in the dialog from which the value to be inserted is read at run time. The Target column role specifies the database column to which the value is inserted. A Button object can participate in only one Insert relationship.

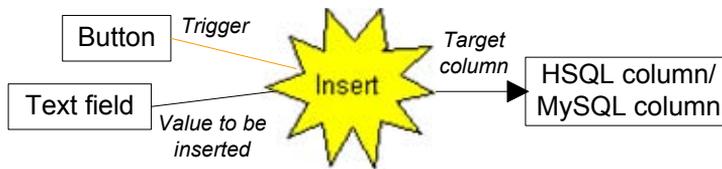


Figure 11. The Insert relationship.

The Insert relationship does not use the where clause in the SQL statement. In all the other relationships, the where clause is needed. The where clause is specified with the Condition column and Condition value roles. The Condition column role specifies the column that is used in the where clause and the Condition value specifies a widget, from which the value for the column is read. The Condition value role has a property called Operator, which specifies the operator that is used in the where clause. Both of the roles have a property called Sequence number, which connects the specific value to a certain column. This way the where clause can contain several conditions which are combined by using the AND operator. If the Condition value is not specified, the value is taken from the selected item in the view if the selected item has a column with the same name. If the Condition value role is not specified and there is no item selected, the select operation is not executed.

Figure 12 presents the objects that can participate in the Select relationship, which is used for retrieving data from the database. The Column role specifies the column to be selected. If the Column role is not used, it means that instead of selecting a specific column, table items are selected from the view from which the dialog was opened. The Target widget role specifies the widget where the result of the Select operation is written. If the target widget is not specified, the result of the select operation is shown in the view from which the dialog box was opened. However, if the select operation is specified to a different database table that is shown in the view, the results are not shown in the view. The Enable role specifies the checkbox that enables or disables the Select action at run time. The Button object may participate in only one Select relationship.

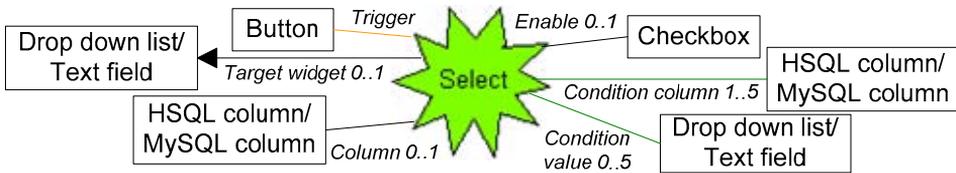


Figure 12. The Select relationship.

Figure 13 presents the objects that can participate in the Update relationship, which is used for updating an existing field in the database. The Target column role specifies the column that is to be updated and the widget in the New value role specifies the new value for the column. If the condition column and the Condition value roles are not specified, the selected view item is updated if it has a column with the same name as the target column. The Button object can participate in multiple Update relationships, so many values can be updated by pressing a single button.

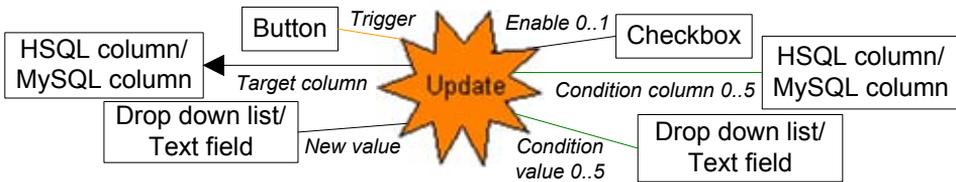


Figure 13. The Update relationship.

Figure 14 presents the Delete relationship, which is used for deleting a row from the database. A button object can participate in only one Delete relationship.

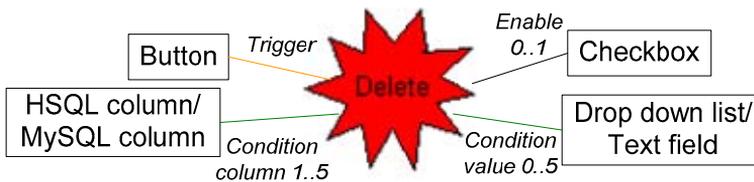


Figure 14. The Delete relationship.

In the previously presented relationships, the type of the target column did not matter, since the operation is always made to the database which is shown in the view from which the dialog box was opened. However, in the remaining

relationships, the type of the column is taken into account. The Transfer from file relationship transfers binary or character data from the file system to an HSQL or MySQL database. Figure 15 presents the objects that can participate in the Transfer from file relationship. If the Condition column and the Condition value roles are not specified, the selected view item is updated if it has a column with the same name as the target column. The Button object can participate in multiple Transfer from file relationships, so many values can be updated by pressing a single button. The File role specifies the text field in the dialog box that should contain the file path of the source file. The Target column role specifies the column in the database where the contents of the file are transferred.

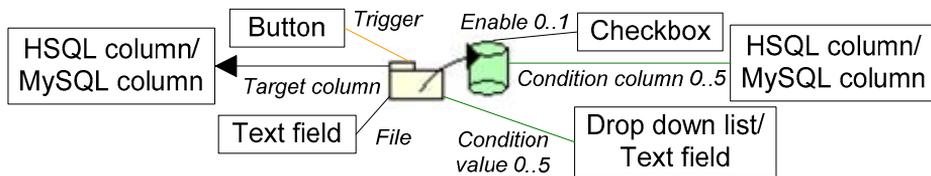


Figure 15. The Transfer from file relationship.

Sometimes it is necessary also to transfer data from the database to the file system. For this purpose, there is the Transfer to file relationship. Figure 16 presents the structure of the Transfer to file relationship. The File role specifies the text field that should contain the file path of the target file. The Source column specifies the column from which the data is transferred to the target file.

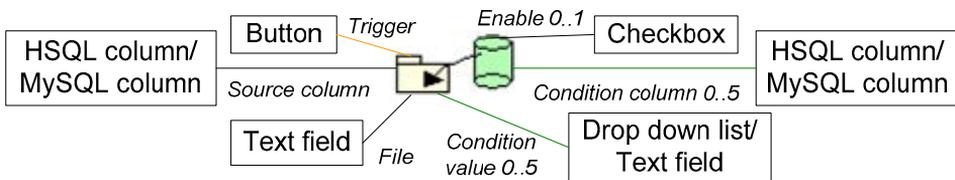


Figure 16. The Transfer to file relationship.

It would be possible to transfer data between the databases using only the Transfer to file and Transfer from file relationships (in that order) but for convenience, also a relationship for transferring data directly between the databases is provided. The Transfer between databases relationship can be used to transfer data from an HSQL database to a MySQL database or the other way

round. Figure 17 presents the structure of the Transfer between databases relationship. The Source column and the Target column roles specify the source and the target columns for the transfer. The source and the target columns have to be from different databases.

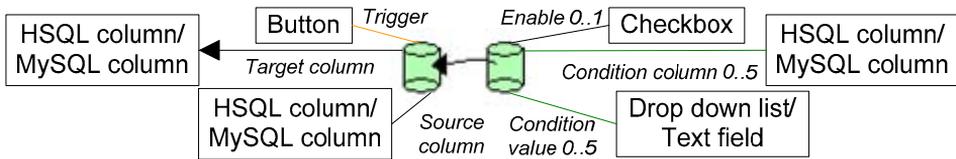


Figure 17. The Transfer between databases relationship.

6. Implementation of the code generator

The syntax of DSML defines what kind of objects can appear in a graph and how they can be connected with different relationships. The semantics of DSML is defined in the code generator. The code generator traverses the model and uses the information in the model to generate a fully functional application. The code generator is defined in the Generator Editor tool provided by the MetaEdit+ environment. The code generator consists of reports, which can call other reports. The reports are written in the MERL scripting language. An example of the MERL language is presented in Figure 18, which includes a part of the generator definition that generates the plugin.xml file. A report has to be defined for a certain graph type. For example, if a report would be defined for the Dialog graph and the context is View graph when the report is called, the generator does not find the constructs in the Dialog graph. From here on, a report is called a generator or a sub generator.

```
1 ` <extension point = "org.eclipse.ui.views">` newLine
2 foreach .View category{
3   ` <category id = "Category' objectid `` ` newLine
4   `           name = "` :Category name `"/>`newLine
5 }
6 foreach .View{
7   variable `category' write `0' close
8   do ~View>Contains~Category.(){
9     variable `category' write `Category' objectid close
10  }
11  do decompositions{
12    foreach .View{
13      ` <view `
14      if $category <> `0' then
15        `category = "` $category `` ` newLine
16      endif
17      ` class = "` $package `.view.View' objectid `` ` newLine
18      `       icon = "icons' sep :Icon path;2 %file'`` ` newLine
19      `       id = "View' objectid `` ` newLine
20      `       name = "` :View name;1 ``"/>` newLine
21    }
22  }
23 }
24 ` </extension>` newLine
```

Figure 18. An example of the MERL language.

The first priority when implementing the code generator is that the generated code should be of high quality and contain as few errors as possible. As stated before, the constraints set for DSML are not comprehensive so it is possible to construct a model that does not define the plug-in completely. For this reason, it is important that the code generator takes care of the constraints that could not be defined in the DSML level.

6.1 Code style

It is good coding practice to give classes, methods, and variables descriptive names so that the code is easy to understand by an outsider. However, since the purpose of domain-specific modelling is to develop such a code generator, which is able to generate 100 percent code that does not require manual editing it is not the first priority while developing the code generator that the generated code is easy to understand. The generator names the classes and variables by combining the type of the object and the object ID, which is a property that MetaEdit+ automatically assigns for each of the object instances. The names can also be more complicated, including possibly many object IDs. This kind of naming convention is used because Java class names can not have empty space and the object ID never contains empty space. The object ID is also unique so this procedure guarantees uniqueness of the class and variable names. This naming scheme does, however, also have a downside: objects in dialogs can not be reused because this causes duplicate names in the code.

6.2 The structure of the code

The code generator implementation is based on the existing implementation of the Stylebase for Eclipse tool, because it is easier to define the generator when there is some idea of the required generation result. The Stylebase for Eclipse tool consists of three separate plug-ins: the core plug-in, the MySQL database plug-in, and the HSQL database plug-in. The core plug-in contains most of the functionality and at least one database plug-in needs to exist for the core plug-in to execute successfully. The core plug-in applies the Model-View-Controller architectural pattern. The MVC architectural pattern divides the application into three parts: the model, the view, and the controller. The model contains the core functionality and data, the view displays the information to the user, and the

Although the basis for the implementation was the existing implementation, there were some modifications that were necessary. The Model and the database plugins required the most changes since in the Stylebase for Eclipse tool, only patterns are saved in the database and in constructed DSML, any kind of data should be possible to be stored in the database. This is why the IModelAdmin and the IDbPlugin interface had to be made independent of the data stored in the database.

6.3 The structure of the code generator

The instructions for code generator definition are that there should be a separate code framework and the generator should generate as little code as possible (i.e. glue code). Since the existing source code of the Stylebase for Eclipse tool had to be modified, it was easier to have all the necessary code in the generator and not as separate files. Of course, there is the Eclipse platform which is not included in the code generator. Also the database drivers are provided as separate jar files. Otherwise, all the necessary parts of an Eclipse plug-in are generated by the code generator.

Figure 20 presents the structure of the code generator. The code generator has a hierarchical structure where the top level generator is responsible for calling the sub generators. The names of the sub generators start with an underscore, which identifies the generator as a sub generator. The top-level generator is called generate and it creates some variables and translators, creates the plug-in directory, and calls the sub generators.



Figure 20. The structure of the code generator.

The `rmdir` and `mkdir` sub generators include commands for removing a directory and creating a directory in the file system. The `hsqldbDatabase` and the `mysqlDatabase` sub generators create the HSQL and MySQL database plug-ins, respectively. The structure of these generators is similar to the structure of the top-level generator, since all the corresponding items need to be created for the database plug-ins also. The `icons` sub generator creates the `icons` directory under the plug-in directory and then copies the icons that are used in the views, menu items and toolbar buttons to the `icons` directory. The source code generator creates all the necessary Java source files for the core plug-in. The `schema` sub generator creates the `schema` directory and an XML description of the details of each extension point provided by the core plug-in. The `pluginXML` generator creates the XML description of the plug-in. The `manifest` sub generator creates the manifest file for the plug-in. Finally, the `compile` generator creates a BAT file for compiling the source code and executes the file. The code generator generates either two or three plug-ins, depending on the number of databases the modeller has selected.

The `sourceCode` generator generates the source code for the core plug-in. Figure 21 illustrates the structure of the `sourceCode` generator. The `sourceCode` generator has eight sub generators. The first sub generator is the same `mkdir` generator for creating a directory that was used also by the top-level generator. The `framework` sub generator generates the activator class for the plug-in, some classes and interfaces to the controller package, and the system package. The `preferences` sub generator generates the preferences package, containing code for each preference page, as well as a preference initialiser, which initialises each preference page with default values. The `viewController` sub generator creates a controller class for each view. The `view` generator creates the code for each view. The `model` sub generator creates the `IDbPlugin` interfaces, `IModelAdmin` interfaces, and `ModelAdmin` classes for each view. The `tableItem` sub generator generates the `TableItem` classes and the `TableItemContainer` classes for each view table. The `viewActions` sub generator generates all the actions for each view that contains a view table. DSML does not restrict adding actions also to views that do not contain a view table, but these actions are not generated. The `viewActions` generator also generates the code for the dialog boxes. The Dialog layout definition is a little problematic. To create the dialogs from the dialog graph, the code generator goes through the `Contains` relationships and orders them by using the location. Then `GridLayout` is used for laying out the dialog. It

is possible that the layout is not the desired one, but by moving the objects slightly, the desired outcome can usually be achieved.

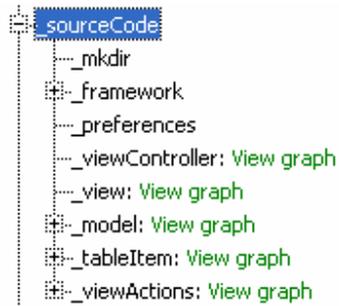


Figure 21. The structure of the source code generator.

7. Case example: Stylebase for Eclipse

This chapter validates constructed DSML with a case example. First, the Stylebase for Eclipse product family is described and the variability of the product family is analysed. Second, DSML is used for modelling the Stylebase tool and the source code for the plug-in is generated. Finally, the model-driven approach of developing a repository-based Eclipse plug-in is compared with the code-centred approach.

7.1 Case description

Stylebase for Eclipse is an open source software architecture knowledge management tool which is implemented as a plug-in to Eclipse. Stylebase is a knowledge base for storing design patterns and architectural styles as well as quality attributes associated with them. The Stylebase for Eclipse tool allows the user to browse and maintain the knowledge base. The purpose of the tool is to improve the quality of software design and enable reuse of architectural information. Stylebase for Eclipse is a product family, which possesses a number of variation points. Figure 22 presents the features in the Stylebase for Eclipse product family. The model input feature is represented with a dashed line in Figure 22, since it is a so called external feature. An external feature is offered by the target platform of the system so it is thus external to the architecture of the software [49]. In this case, the repository can accept any kind of textual model of a design or architectural pattern so the model input tool is not restricted in any way. [10, 47]

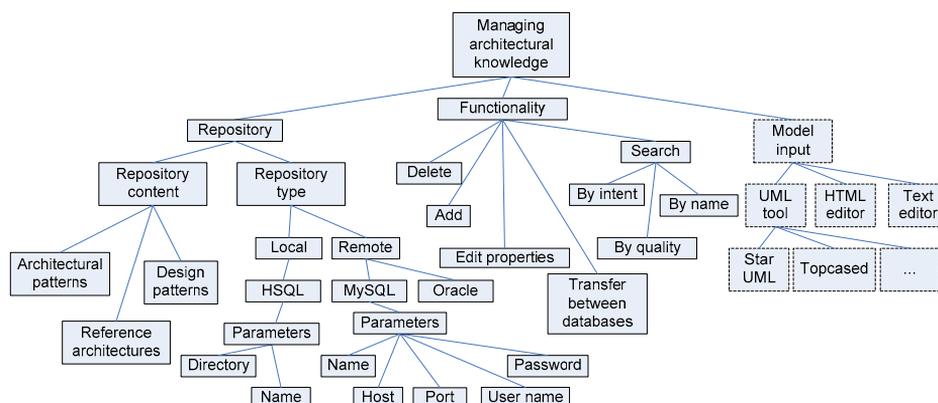


Figure 22. The Stylebase product family.

Figure 23 presents the features that are implemented in the current version of the Stylebase for Eclipse tool. The current version of the Stylebase for Eclipse tool utilises two relational database management systems: MySQL and HSQLDB. MySQL database is used as a central repository, which can be used by multiple users and HSQLDB is used to store patterns locally, and if the user wants to share his or her patterns with others, he or she can upload them to the shared MySQL database. [45]

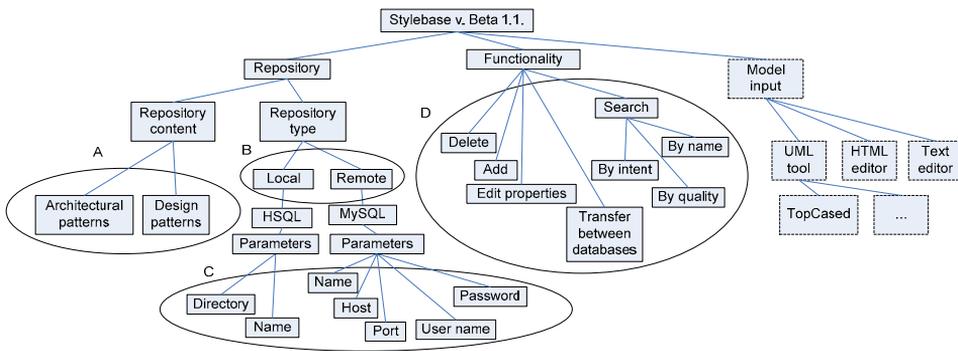


Figure 23. Stylebase for Eclipse version Beta 1.1.

The ellipses labelled from A to D in Figure 23 represent the variant sets in the Stylebase product that have a common variability realisation technique, introduction time and binding phase. The binding of the model input feature is not considered here because it is an external feature which does not have a specific variability realisation technique. Table 18 presents how these variant sets are realised in the current version of the Stylebase for Eclipse tool.

Table 18. Variability sets in Stylebase.

Variant set	Variability realisation technique	Introduction phase	Binding phase
A	Database structure design and manual coding	Requirements	Design
B	Manual coding	Architecture	Implementation
C	Eclipse preference pages	Design	Run time
D	Manual coding	Requirements	Implementation

Figure 24 presents the variant sets in the current version of the Stylebase tool with the frame of reference presented in Section 2.3.

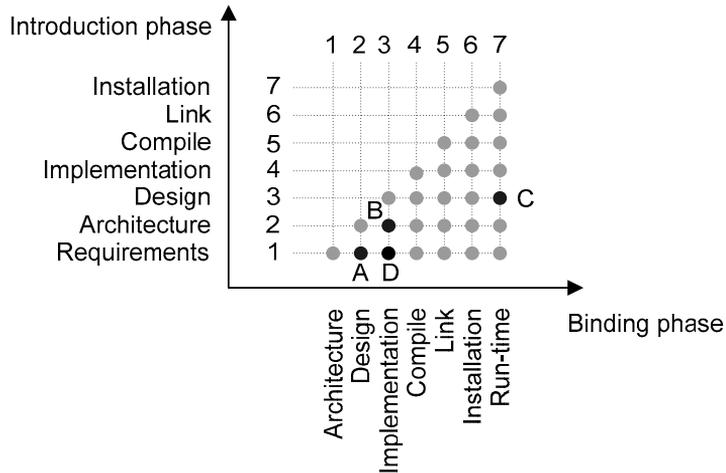


Figure 24. Variability in Stylebase.

7.2 Modelling the Stylebase for Eclipse tool

DSML and the code generator can be used for generating a large number of different repository-based Eclipse plug-ins. For example, a photo management application could be easily generated with DSML and the code generator. The application could be used for example, to manage photos associated with persons who appear in the photo. In this section, the modelling of the version Beta 1.1 of the Stylebase for Eclipse plug-in is presented, which represents just one of all the possible plug-ins that DSML is able to generate.

Figure 25 presents an instance of the Workbench graph, which is the top-level graph of DSML. The version Beta 1.1 of the Stylebase for Eclipse tool uses both an HSQL database and a MySQL database, which are declared in the Workbench graph. The original Stylebase tool uses the same view to show content from each of the databases and there is a toolbar button which changes the database. This is not possible to do with DSML defined in this work, because a view is dedicated to show information from either an HSQL database or a MySQL database by using either the HSQL view table or the MySQL view table object. Consequently,

the contents of the HSQL database and the MySQL database are presented in different views: the Stylebase HSQL and the Stylebase MySQL views, respectively. The available quality attributes for the patterns are also presented in their own views, instead of showing them in a Dialog box, as in the original Stylebase for Eclipse tool. There are three preference pages declared in the Workbench graph. The Preference page entitled Stylebase is the main preference page and the pages entitled HSQL and MySQL are shown under the Stylebase page in the Eclipse preferences dialog. All objects except the View category object have a decomposition graph.

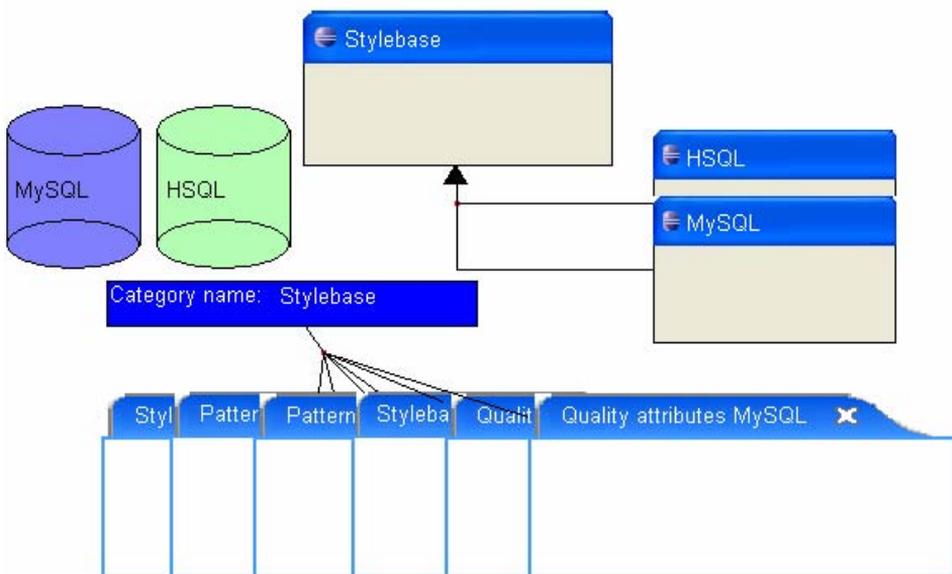


Figure 25. An instance of the Workbench graph.

7.2.1 HSQL database graph

Figure 26 presents an instance of the HSQL database graph, which describes the structure of the HSQL database that is used by the Stylebase for Eclipse tool. The database consists of three tables: patterns, quality_attributes, and attribute_definitions. The patterns table contains information about a design or an architectural pattern, such as the HTML description of the pattern. The attribute_definitions table contains the definitions of quality attributes that are

associated with the patterns. The `quality_attributes` table connects the patterns to their quality attributes and provides also the `rationale` field which can be used to explain why a pattern promotes a quality attribute. The structure of the database is not exactly the same as in the original Stylebase tool, because in DSML, the references between different tables were a bit problematic. That is why the primary keys in the `quality_attributes` table are strings that contain the name of the pattern and the attribute, instead an integer id. This makes it easier to define dialogs that show the quality attributes of a pattern.

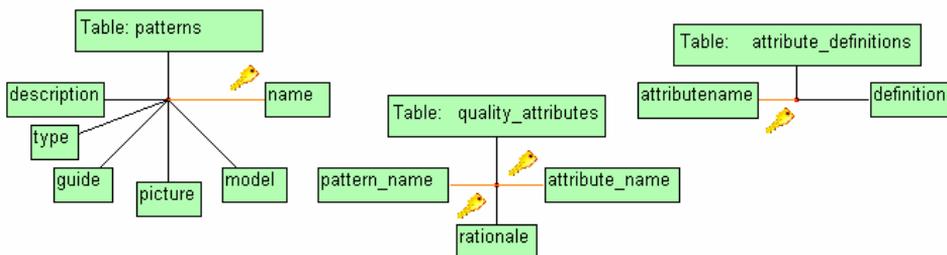


Figure 26. An instance of the HSQL database graph.

7.2.2 MySQL database graph

Figure 27 presents an instance of the MySQL database graph, which defines the structure of the MySQL database that is used by the Stylebase for Eclipse tool. As can be seen from the figure, the structure of the MySQL database is the same as the structure of the HSQL database. Again, the structure is not the same as in the original Stylebase for Eclipse tool.

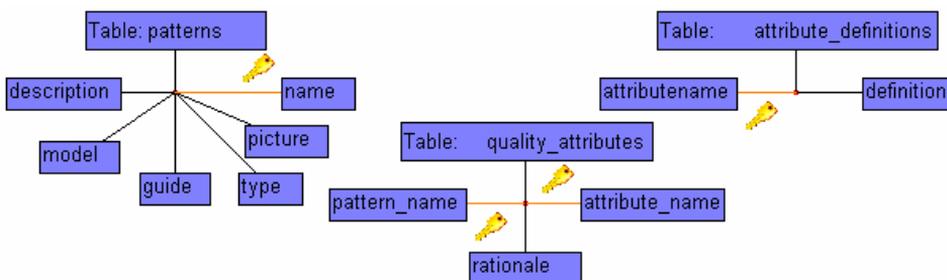


Figure 27. An instance of the MySQL database graph.

7.2.3 Preference graph

Figure 28 presents an instance of the Preference graph. This preference page includes the Page object and five String field editors that contain MySQL database related parameters. The Page object is the big background object and the String field editor objects are connected to the Page object with the Contains relationship (grey line with a red dot in the middle). The String field editors defined in this Preference graph instance are reused in the MySQL database graph properties to specify where the MySQL database plug-in can find the necessary parameters to connect to the MySQL database.

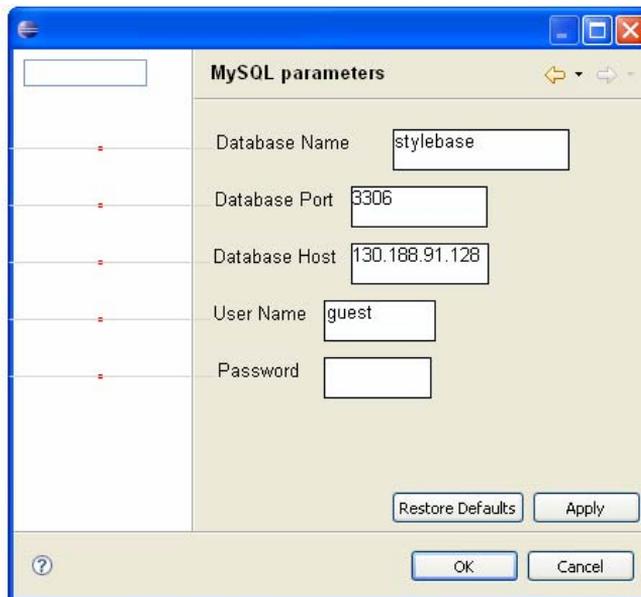


Figure 28. An instance of the Preference graph.

7.2.4 View graph

Figure 29 presents a View graph instance, which is the decomposition graph for the Stylebase HSQL view object in the Workbench graph. This View graph instance describes an Eclipse view that contains an HSQL view table. The table has a context menu, which contains eight menu items. There are more menu items than in the original Stylebase for Eclipse tool, since all the actions could

not be implemented in the same way with DSML. All the menu items, except the Delete menu item have decompositions into Dialog graphs, which describe the dialog box that is opened when the menu item is selected. There are also four toolbar buttons (number 1 in Figure 29). The first toolbar button opens a MySQL view, so it decomposes into the View graph that defines the MySQL view. The Add new pattern and Search patterns toolbar buttons have decompositions into Dialog graphs. The last Toolbar button does not have a decomposition graph, since it is connected with the Refresh relationship to the HSQL view table.

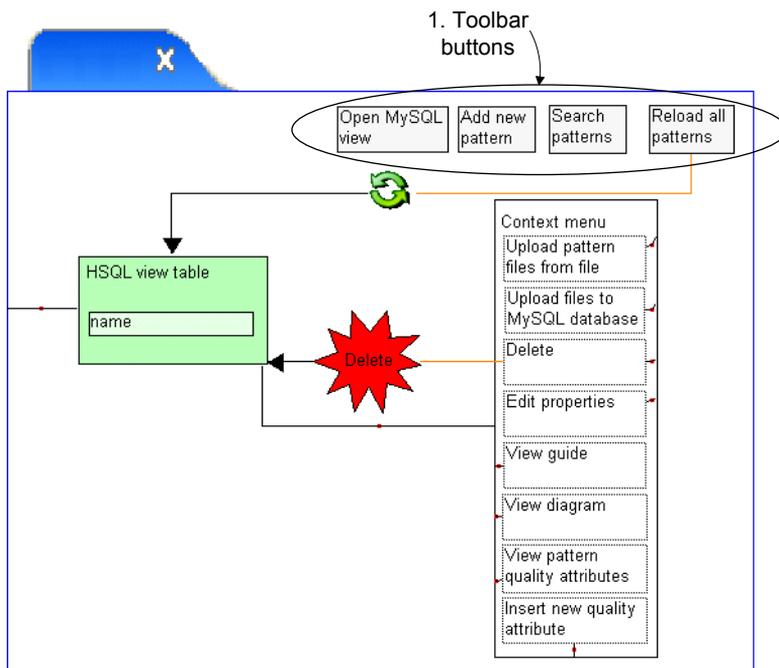


Figure 29. An instance of the View graph.

7.2.5 Dialog graph

Figure 30 presents an instance of the Dialog graph. This dialog shows the quality attributes of the pattern that is selected in the Stylebase view. The problematic nature of the references between different database tables can be seen in Figure 30. The Initialise relationship (number 1 in Figure 30) can be used to initialise a widget only if the column is in the same table that is shown in the view from

which the dialog was opened. Since the quality attributes of a pattern are in a different table than the pattern, the attributes need to be retrieved using the Select relationship. When the user presses the Button labelled Refresh attributes (number 2 in Figure 30), the quality attributes are fetched to the drop down list (number 3 in Figure 30). It is possible to use the select relationship to fetch all columns in the table that is shown in the view from which the dialog was opened, but again it is not possible to select multiple columns from another database table. Therefore, the rationale needs to be fetched using another Select relationship. When the user presses the button labelled “Refresh rationale” (number 4 in Figure 30), the rationale is fetched to the text field (number 5 in Figure 30). Because the nature of the GridLayout object, it is sometimes necessary to add empty Label objects (number 6 in Figure 30) to the Dialog to achieve a certain layout. It is not possible to reuse the objects in the dialog graph, because the variables in the code are named after the object id of the widget. However, it is possible to reuse widgets defined in other Dialog graphs. This dialog can be used also to show the quality attributes in the MySQL database, because the Insert, Update, Select, and Delete relationships do not use the type of the database column (HSQL database column or MySQL database column). However, the Transfer between databases, Transfer from file, and Transfer to file relationships use the type of the column object to determine the source or target of the operation.

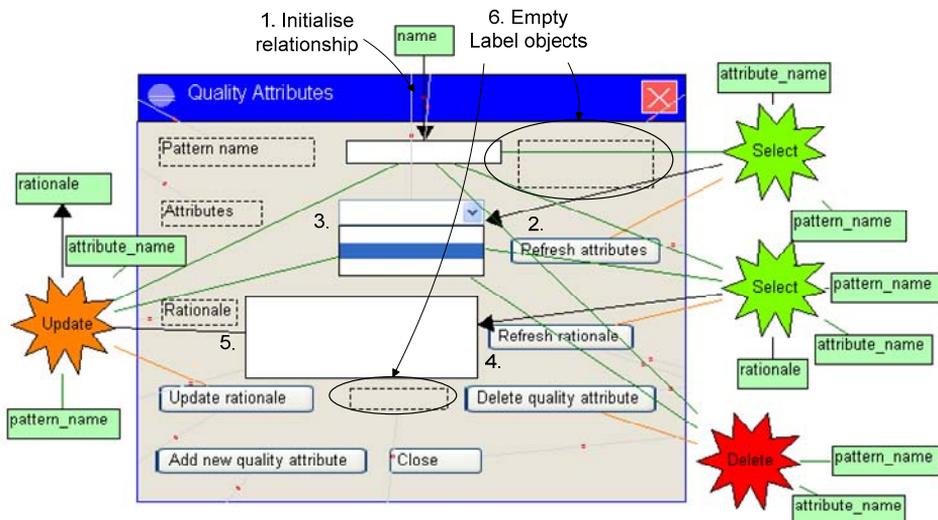


Figure 30. An instance of the Dialog graph.

7.3 Generating the source code

After all the necessary features have been modelled with DSML, the source code can be generated. The source code of the Stylebase for Eclipse tool is generated by running the top-level generator called “generate” presented in Figure 20. The necessary plug-ins are generated to the directory specified in the Destination directory property of the Workbench graph. In this case, the destination directory is set to the Eclipse plugins directory, so the Stylebase for Eclipse tool is ready to use when Eclipse is restarted. Figure 31 presents a screenshot from Eclipse with the generated tool. The dialog that is open in Figure 31 is the dialog that was modelled in Figure 30.

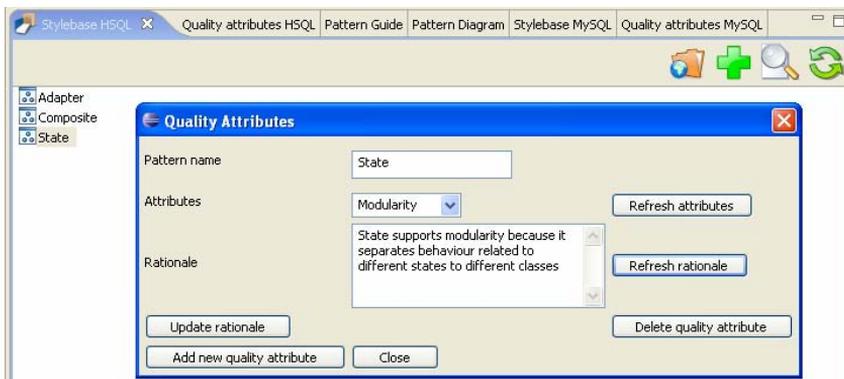


Figure 31. The generated Stylebase tool in Eclipse.

Figure 32 presents a sample of the generated code. The code defines a new button widget and it is a part of the createContents method of a dialog box. The name of the button is specified in the model (row 2 in Figure 32) as well as the size of the button (rows 4 and 5 in Figure 32).

```

1  this.button12882 = new Button(this.composite, SWT.NULL);
2  this.button12882.setText("OK");
3  GridData gd12882 = new GridData();
4  gd12882.widthHint = (int)74.0;
5  gd12882.heightHint = (int)21.0;
6  this.button12882.setLayoutData(gd12882);
7  this.button12882.addSelectionListener(new SelectionAdapter() {
8      public void widgetSelected(SelectionEvent e) {
9          View3781Dialog12669.this.setReturnCode(12882);
10         View3781Dialog12669.this.caller.handleInput();
11     }
12 });

```

Figure 32. A sample of the generated code.

7.4 Comparison of the MDD approach to the code-centred approach

Domain-specific modelling is well suited to the software product family approach and handling variability. The metamodel and the code generator definition roughly correspond to the domain engineering phase, and using the metamodel (i.e. modelling) and generating code from the model correspond to the application engineering phase. The frame of reference for representing variability presented in Section 2.3 is not directly applicable to domain-specific modelling because the variability realisation techniques and the phases of the software development process are different. In this work, a frame of reference for representing variability in DSM is introduced. The new frame of reference is presented in Figure 33 with the variant sets from Figure 22. The variant sets that were previously bound at the implementation phase can now be bound already at the modelling phase and code can be automatically generated. The variant set C can be now bound already at the modelling phase, or alternatively at run time.

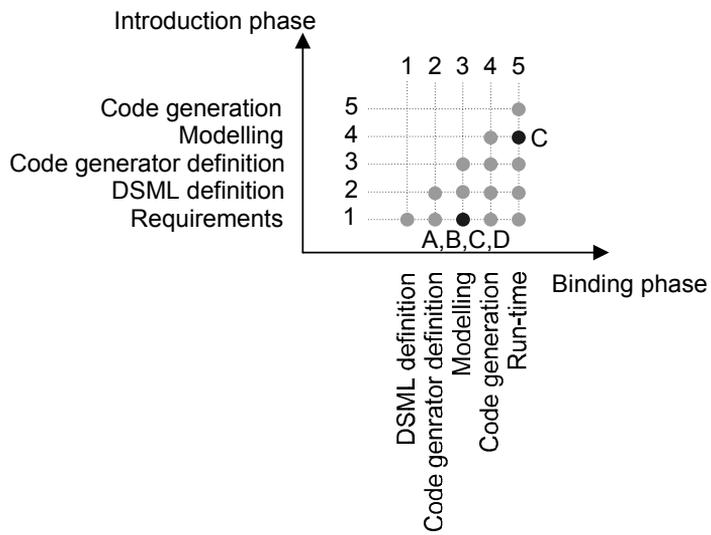


Figure 33. Variability representation in DSM.

8. Discussion

The aim of this work was to define a prototype domain-specific modelling language and a code generator for developing repository-based Eclipse plug-ins. The work included many challenges but was also very rewarding. This chapter presents some experiences during the work and analyses developed DSML and the code generator.

8.1 Experiences on DSML design

Initially, it was difficult to invent a suitable domain for the work, and even after the domain was selected, finding the relevant domain concepts was somewhat troublesome. The reason for this was probably the fact that the author of this work had no previous experience in any kind of language design. The selected approach for the work was the look and feel approach with domain concepts (see Section 4.2). It was quite easy to pick the relevant user interface concepts from Eclipse to DSML. Combining the functionality to the user interface concepts seemed daunting at first, but after initial struggles, also the behaviour part started to come together. However, the implementation of the functionality is not fully satisfying due to the complex nature of SQL queries.

It was hard not to be too greedy with the number of different domain concepts. This relates to the trade-off between generality and expressiveness: the more general the language, the less expressive it becomes [50]. Also, the difficulty of implementing the code generator increases every time a new concept is added to the language. It was necessary to leave out some concepts that were initially selected, so that the scope of the work would not become overwhelming. Consequently, when constructing DSML, the number of initial concepts should be small and the relationships in the language should be kept simple. New concepts should be added after the code generator works with the initial concepts.

8.2 Experiences in DSML implementation

The selected tool for the definition of DSML and the code generator was the MetaEdit+ environment, which was quite easy to learn but which lacked some more intuitive features. Form-based metamodelling was used in this work, which meant that all the types in DSML were defined using dialog boxes provided by the MetaEdit+ tool. This was not the most convenient way of defining DSML because every object, relationship, role, and property had to be defined in a separate tool and after that the concepts were tied together in the Graph tool. This was sometimes quite awkward when defining a graph only to realise a role was needed to some relationship. This meant having to start the role tool and generate the role and return to the graph tool. Nevertheless, form-based metamodelling was a good selection for this work because of the better precision over graphical metamodelling.

Also the functionality in the graphical diagrams was sometimes different than was expected. For example, intuitively one might expect that when an object is double clicked, the decomposition graph of that object should open. However, this was not the case; instead the decomposition graph had to be opened from a pop-up menu which appeared when the object was clicked with the right mouse button. Also the modality of dialogs was annoying sometimes when many windows were open.

The symbol definition for metatypes was also somewhat limited. It was not possible to use the icon specified by the icon path property in the object symbol. Luckily, it is possible to see the icon by selecting “Execute” from the input box, but it would have been more intuitive to the user of DSML if the icon would have been a part of the object symbol.

The code generator definition was the most time consuming task in this work. At first, it was necessary to learn the MERL generator definition language syntax, which was not always so easy, even though MERL is a relatively simple language. Even after most of the MERL syntax was correct, problems came up, e.g. comparing strings instead of integers while ordering elements. Also, the generator had to be defined for the right graph type, which was sometimes a little confusing. Luckily these problems were quickly solved by sending e-mail to MetaCase support. After getting the MERL syntax right, it did not guarantee

that the syntax of the generated Java code was correct. The bright side in the errors in the generated code was that if there were many, they were in the same place, so it was quite easy to discover them. Testing and debugging the generated plug-in was also difficult, especially because the database functionality was in different plug-ins than the core functionality. For this reason, it might have been more reasonable to implement the whole functionality in a single plug-in. On the other hand, it was easier to implement the functionality as separate plug-ins as the existing Stylebase for Eclipse tool was implemented that way. Eclipse also brought its own problems to the mix. For example, if the plugin.xml file changes, the change does not take effect when Eclipse is restarted if the name of the plug-in stays the same. It is required either to change the name of the plug-in, or remove the plug-in from the plugins directory and restart Eclipse, and then generate the plug-in and restart Eclipse once more.

Since the nature of the work was iterative, sometimes a more convenient name for an object came up or the structure of a relationship needed to be changed completely. Changing the metamodel, made it necessary to change also the code generator and forgetting the old name or structure somewhere caused errors that were not always easy to find.

Although the code generator tries to remove the old plug-in directories first before creating new ones, this does not always happen, because it is not possible to remove the directories if there is an access violation (if there is for example a command prompt window open at the same directory). Then the files are generated to the old directory and some old files may cause problems in the compiling phase. The best way to overcome this problem was to remove the old plug-ins first by hand (or close all windows that may cause an access violation).

In the DSM field, there are only few commercial tools available. Based on the tool comparison done in Section 3.2 and experiences on MetaEdit+, it can be stated that, MetaEdit+ is a good environment for the development of DSML and a code generator. There were some features that were not used in this work, such as the model animation possibilities, but during this work, a good overall understanding of the MetaEdit+ tool was gained.

Concerning experiences on DSML implementation, the following rules can be summed up:

1. Learn how to use the metamodeling tool properly before starting the implementation of DSML.
2. If Eclipse plug-ins are developed, implement rather a single plug-in than multiple plug-ins.
3. Avoid changing the names of DSML concepts after (a part of the) code generator is implemented.

8.3 Evaluation of DSML

Developing DSML is a difficult task, since both domain and language development expertise are required [50]. The author of this work had relatively little experience in developing repository-based Eclipse plug-ins, and no experience at all in developing domain-specific modelling languages. However, prototype DSML and a code generator were successfully developed, and working Eclipse plug-ins can be generated. The model-driven approach enables the binding of the variant sets in the modelling phase instead of the implementation phase. In principle, the variant sets have to be bound earlier than before, but since there is no manual implementation phase in the model-driven approach, the variability can be handled in a more flexible way than in the traditional code-centred approach.

The requirements that were set for DSML in Section 4.1 were fulfilled quite well. R3 was not fulfilled totally, since DSML is not as intuitive and easy to use as it should have been. Partly the usability of DSML is restricted by MetaEdit+, but the usability might have been improved by better design of the language and the code generator. In the definition of dialogs, there are also some weaknesses. First, the dialog box has to have the same number of widgets in every row, so sometimes empty labels need to be added to achieve a desired layout. Also, the number of columns in the dialog needs to be selected correctly from the Dialog box object. The benefit of the DSM approach is its automatic code generation, so even if the layout comes out different than was required, it is easy to change the model and generate new plug-ins. It may be necessary to perform several

iterations before the desired layout is achieved. Second, the behaviour definition in the Dialog boxes is quite complex because of the relationships have so many roles. It was difficult to define the roles to be simpler, because SQL statements need to be defined precisely.

The main problem that hinders the usability of DSML, in my opinion, is that DSML is too complicated. Although some simplifications were made along the way, the final result was still not simple enough. The n-ary relationships in the Dialog graph have too many roles and make the Dialog graphs look cluttered. This leads also to the fact that requirement R4 was not fully met, since it is possible to construct a model that either produces compilation errors, or does not perform valid actions to the database. However, the generated plug-in logs events to the log file, from which the modeller can conclude the errors in the modelling of the database action. To facilitate the adoption of DSML, a simple language with minimum functionality should be developed as the first DSML implementation.

Another problem in DSML is the references between database tables. It is possible to form the where clause of the SQL statement with the Condition column and Condition value roles, but it is not possible to use the result of a select statement as the condition for the next select statement without saving the results to a widget in a dialog box. This leads to poor usability of the dialogs since the user has to click several buttons to see the required information.

For the most part, the case example was successful, although there were some things that could not be modelled with DSML. In the Stylebase for Eclipse tool, the patterns in the MySQL database can be locked to a certain user and they can not be modified by others while some user possesses a lock. This was not possible to model with DSML. In addition, some dialog boxes were not identical to the dialog boxes of the Stylebase for Eclipse tool because of the restrictions in the database operations. In the Stylebase for Eclipse tool, the HSQL database stores only the file paths of the pattern files, which makes the database perform faster. This was not possible to do with DSML.

The iterative development approach was well suited for the development of DSML and the code generator. First, a minimum language was developed and a code generator which was able to generate only a basic Eclipse plug-in with an

empty view and some preference pages. After initial experiments, it was encouraging to notice that the code generator works and continue building the language. At a later stage, it was necessary to construct multiple models to find errors in the code generator. At first, only a single model was used for testing and all went smoothly. When a whole new model was constructed, multiple compilation errors occurred.

The domain is quite restricted since the target platform is Eclipse and of all the possible Eclipse plug-ins only repository-based ones belong to the domain. However, the whole idea behind domain-specific modelling is to restrict the domain and be able to generate full code for that restricted domain. DSML can be also extended if necessary. There are many possible directions for extensions:

1. Extending the features of the language
2. Extending to other kinds of repository-based applications
3. Extending to all possible Eclipse plug-ins.

In the first alternative, new widgets or new operations to the database could be added to the Dialog graph. It would also be useful to add the Condition column and Condition value roles to the Initialise relationship as well as add a possibility to select multiple columns with the Select relationship. In the second alternative, the Database graph and the Dialog graph would be ready to use as is, but it might be necessary to add other graphs to DSML. The third alternative might be a little easier to implement than the second alternative. For example, in the Workbench graph, the possibility to define own custom editors could be added as well as help pages for the plug-in. In the View graph, it would be easy to add new content types such as plain text. Also a pull down menu and a double click action could be easily added to the View graph. The data types in the databases are also very restricted but this is also possible to change by changing the database column definitions and implementing the changes to the code generator. Since DSML is going to be published in an open source community [11], new ideas for extension possibilities may also come from the community.

Valuable experience on defining DSML and code generator was gained during the work. This experience is summed up in the following rules of thumb:

1. If DSML is developed without earlier experience, focus on simplicity.
 - a. Try to keep the number of objects to a minimum.
 - b. Try to avoid too many roles in relationships.
2. Develop the language and the code generator iteratively.
3. Experiment with different model variants to discover errors in the code generator.

To sum up, developed DSML is able to generate repository based Eclipse plug-ins. The level of abstraction was raised well because the dialogs and the preference pages resemble the real Eclipse dialogs and preference pages. Developed DSML and the code generator reduce the time to develop a repository-based Eclipse plug-in. It is nice to see that after initial doubts, 100 percent code generation really is possible and not just a beautiful idea.

9. Conclusion

The aim of this work was to develop prototype DSML and a code generator for creating repository-based Eclipse plug-ins. A metamodelling tool comparison was performed to select a suitable tool for the development of DSML and the code generator. Four metamodelling tools were compared and the MetaEdit+ tool was selected because of the good code generation definition possibilities. DSML and a code generator for creating repository-based Eclipse plug-ins were developed with the MetaEdit+ metamodelling tool.

The requirements for DSML were fulfilled quite well. There are some deficiencies in the constraints for DSML and in the references between database tables. The usability of DSML would also have been better if the language would have been simpler. The extensibility requirement for DSML was fulfilled well and there are multiple possibilities for future extensions.

DSML was demonstrated by generating the source code for the Stylebase for Eclipse plug-in. The case example shows that DSML facilitates model-driven development of repository-based Eclipse plug-ins. Since the Stylebase for Eclipse plug-in was previously implemented manually, it was possible to compare the model-driven approach with the traditional code centric approach. A frame of reference for representing variability in domain-specific modelling was presented in this work. The domain-specific approach enabled the binding of the variant sets of the Stylebase for Eclipse product family in a more flexible way than with the traditional code-centred way, since the binding can be done in the modelling phase instead of in the manual implementation phase.

DSML developed in this work can be used for modelling a repository-based Eclipse plug-in and the code generator is able to generate a fully working Eclipse plug-in. DSML speeds up the development of repository-based Eclipse plug-ins and thus demonstrates the usefulness of the domain-specific modelling approach.

References

- [1] Greenfield J. & Short K. (2004) *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. Wiley Publishing, Inc., Indianapolis, USA. 666 p.
- [2] Tolvanen J. P. (Accessed: 18.1.2008) *Domain-Specific Modelling: How DSM Code Generation Can Go Beyond The Benefits Delivered By UML*. URL: <http://reddevnews.com/techbriefs/print.aspx?editorialid=120>.
- [3] Selic B. (2003) *The Pragmatics of Model-Driven Development*. IEEE Software, Vol. 20, Issue 5, pp. 19–25.
- [4] Luoma J., Kelly S. & Tolvanen J. P. (2004) *Defining Domain-Specific Modeling Languages: Collected Experiences*. In: *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04)*, Oct. 25, Vancouver, British Columbia, Canada. Pp. 1–10.
- [5] Gamma E. & Beck K. (2004) *Contributing to Eclipse: Principles, Patterns and Plug-Ins*. Addison-Wesley Professional, Boston. 395 p.
- [6] Anonymous. (Accessed: 16.1.2008) *About GEMS*. URL: <http://www.eclipse.org/gmt/gems/about.php>.
- [7] Anonymous. (Accessed: 31.8.2007) *EclipsePlugins: Details for the Merlin Generator Eclipse Plug-in*. URL: http://eclipse-plugins.2y.net/eclipse/plugin_details.jsp?id=916.
- [8] Anonymous. (Accessed: 17.1.2008) *Eclipse Modeling Framework Project (EMF)*. URL: <http://www.eclipse.org/modeling/emf/>.
- [9] Anonymous. (Accessed: 17.1.2008) *Eclipse Graphical Editing Framework Project (GEF)*. URL: <http://www.eclipse.org/gef/>.

- [10] Henttonen K. & Matinlassi M. (2007) Contributing to Eclipse: a Case Study. In: Proceedings of the Software Engineering 2007 Conference, March 29–30, Hamburg, Germany. Pp. 59–70.
- [11] Anonymous. (Accessed: 14.1.2008) Stylebase for Eclipse. URL: <http://stylebase.tigris.org/>.
- [12] Royce W. W. (1987) Managing the Development of Large Software Systems: Concepts and Techniques. In: Proceedings of the 9th International Conference on Software Engineering (ICSE '87), March, Monterey, California, United States. Pp. 328–338.
- [13] OMG. (Accessed: 31.8.2007) MDA Guide Version 1.0.1. URL: <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [14] Booch G., Brown A., Iyengar S., Rumbaugh J. & Selic B. (2004) An MDA Manifesto. MDA Journal: Model Driven Architecture Straight from the Masters. URL: <http://www.bptrends.com/publicationfiles/05-04%20COL%20IBM%20Manifesto%20-%20Frankel%20-3.pdf>.
- [15] OMG. (Accessed: 27.11.2007) Model Driven Architecture (MDA) FAQ. URL: http://www.omg.org/mda/faq_mda.htm.
- [16] Deursen A., Klint P. & Visser J. (2000) Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices, Vol. 35, Issue 6, pp. 26–36.
- [17] Blake D. (Accessed: 15.8.2007) Domain-Specific Languages Versus Generic Modeling Languages. URL: <http://www.ddj.com/architect/199500627>.
- [18] Langlois B., Jitka C. E. & Jouenne E. (2007) DSL Classification. In: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Oct. 21–22, Montréal, Canada. Pp. 28–38.
- [19] Feilkas M. (2006) How to Represent Models, Languages, and Transformations? In: Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), October 22, 2006, Portland, Oregon USA. Pp. 169–176.

- [20] Seifert T., Beneken G. & Baehr N. (2004) Engineering Long-Lived Applications using MDA. In: Proceedings of the IASTED Conference on Software Engineering and Applications, November 9–11, MIT, Cambridge, MA, USA. Pp. 241–246.
- [21] Clements P. & Northrop L. (2001) Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, USA. 563 p.
- [22] Svahnberg M., van Gurp J. & Bosch J. (2005) A Taxonomy of Variability Realization Techniques. Software: Practice and Experience, Vol. 35, Issue 8, pp. 705–754.
- [23] Jacobson I., Griss M. & Jonsson P. (1997) Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, New York. 497 p.
- [24] Jaring M. & Bosch J. (2002) Representing Variability in Software Product Lines: A Case Study. In: Proceedings of the Second Software Product Line Conference, August 19–22, San Diego, USA. Pp. 15–36.
- [25] Anonymous. (Accessed: 31.8.2007) SWT: The Standard Widget Toolkit. URL: <http://www.eclipse.org/swt/>.
- [26] Clayberg E. & Rubel D. (2006) Eclipse: Building Commercial-Quality Plug-Ins. Addison-Wesley, Upper Saddle River, NJ. 810 p.
- [27] Microsoft Corporation. (Accessed: 19.12.2007) Domain-Specific Language Tools documentation. URL: [http://msdn2.microsoft.com/fi-fi/library/bb126235\(en-us,VS.80\).aspx](http://msdn2.microsoft.com/fi-fi/library/bb126235(en-us,VS.80).aspx).
- [28] Kosar T., Mernik M. & Lopez P. (2007) Experiences on DSL Tools for Visual Studio. In: 29th International Conference on Information Technology Interfaces, June 25–28, Cavtat, Dubrovnik, Croatia. Pp. 753–758.
- [29] Microsoft Corporation. (Accessed: 20.12.2007) Overview of Domain-Specific Language Tools. URL: <http://msdn2.microsoft.com/en-us/library/bb126327.aspx>.

- [30] Institute for Software Integrated Systems Vanderbilt University. (Accessed: 14.1.2008) Vanderbilt University End User Licence Agreement. URL: <http://repo.isis.vanderbilt.edu/tools/tool/GME/License>.
- [31] Institute for Software Integrated Systems Vanderbilt University. (Accessed: 19.12.2007) GME: The Generic Modeling Environment. URL: <http://www.isis.vanderbilt.edu/projects/gme/>.
- [32] Ledeczki A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason C., Nordstrom G., Sprinkle J. & Volgyesi P. (2001) The Generic Modeling Environment. In: Proceeding of the IEEE International Workshop on Intelligent Signal Processing (WISP'2001), May 24–25, Budapest, Hungary.
- [33] Amyot D., Farah H. & Roy J. F. (2006) Evaluation of Development Tools for Domain-Specific Modeling Languages. In: Proceedings of the 5th International Workshop on System Analysis and Modeling: Language Profiles (SAM 2006), May 31 – June 2, Kaiserslautern, Germany. Pp. 183–197.
- [34] Institute for Software Integrated Systems Vanderbilt University. (Accessed: 10.1.2008) GME User's Manual Version 5.0. URL: <http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf>.
- [35] Anonymous. (Accessed: 17.1.2008) Eclipse Public Licence. URL: <http://www.eclipse.org/legal/epl-v10.html>.
- [36] Anonymous. (Accessed: 17.1.2008) Generative Modelling Technologies Project. URL: <http://www.eclipse.org/gmt/>.
- [37] Anonymous. (Accessed: 17.1.2008) Graphical Modelling Framework Project. URL: <http://www.eclipse.org/gmf/>.
- [38] White J., Schmidt D., Nechypurenko A. & Wuchner E. (2007) Introduction to the Generic Eclipse Modeling System. Eclipse Magazine, Issue 6, Jan. 07, pp. 11–19.

- [39] Anonymous. The Generic Eclipse Modelling System Manual. URL: <http://www.eclipse.org/downloads/download.php?file=/technology/gmt/gems/gems-3.0-rc-1-manual.zip>.
- [40] MetaCase. (Accessed: 10.10.2007) MetaCase – Domain-Specific Modelling with MetaEdit+. URL: <http://www.metacase.com/>.
- [41] Kelly S., Lyytinen K. & Rossi M. (1996) MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Proceedings of the 8th International Conference on Advanced Information System Engineering (CAiSE'96), May 20–24, Herakleion, Crete, Greece. Pp. 1–21.
- [42] Pohjonen R. & Steven K. (2007) Interactive Television Applications Using MetaEdit+. Model-Driven Development Tool Implementers Forum (MDD-TIF07). URL: <http://www.dsmforum.org/events/MDD-TIF07/MetaEdit+.2.pdf>.
- [43] Wilms H. (Accessed: 19.12.2007) Microsoft Domain-Specific Language Tools from a Developer's Perspective. URL: <http://www.infoq.com/news/2007/03/ms-dsl-developer>.
- [44] Sheldon R. & Moes G. (2005) Beginning MySQL. John Wiley & Sons, Incorporated. 866 p.
- [45] Anonymous. (Accessed: 19.12.2007) HSQLDB documentation. URL: <http://www.hsqldb.org/web/hsqldbDocsFrame.html>.
- [46] Anonymous. (Accessed: 10.10.2007) Eclipse SDK Help. URL: <http://help.eclipse.org/help31/index.jsp>.
- [47] Henttonen K. (2007) Stylebase for Eclipse. An Open Source Tool to Support the Modeling of Quality-Driven Software Architecture. VTT Tiedotteita – Research Notes 2387. VTT, Espoo, Finland. URL: <http://www.vtt.fi/inf/pdf/tiedotteet/2007/T2387.pdf>.
- [48] Buschmann F., Meunier R., Rohnert H., Sammerlad P. & Stal M. (1996) Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Chichester, England. 457 p.

- [49] van Gorp J., Bosch J. & Svahnberg M. (2001) On the Notion of Variability in Software Product Lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), August 28–31, Amsterdam, Netherlands. Pp. 45–54.
- [50] Mernik M., Heering J. & Sloane A. M. (2005) When and how to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, Issue 4, pp. 316–344.



Author(s) Sivonen, Sanna		
Title Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins		
Abstract <p>Eclipse is an open source platform for tool integration which can be extended by writing plug-ins that utilise the extension points provided by the Eclipse platform. Eclipse plug-ins are written in the Java language and the plug-in development work can be time consuming especially if multiple plug-ins are developed for the same application domain. Model-driven development is about focusing on models rather than computer programs in software development. Domain-specific modelling follows the principles of model-driven development by promoting the use of domain-specific modelling languages (instead of general-purpose modelling languages).</p> <p>The aim of this research is to develop a prototype graphical domain-specific modelling language (DSML) and a code generator for creating repository-based plug-ins for Eclipse. The purpose of DSML is to raise the level of abstraction and thus speed up the development process of several similar Eclipse plug-ins compared to hand writing the plug-ins in Java language. Also people not familiar with Java (i.e. end users) could build their own extensions with the language defined in this work.</p> <p>Developed DSML is demonstrated by generating the source code of an existing repository-based Eclipse plug-in. The plug-in that is used in the demonstration is an open source software architecture knowledge management tool called Stylebase for Eclipse, which has been developed at VTT Technical Research Centre of Finland. The Stylebase for Eclipse is a software product family, which has a number of variation points. Since the Stylebase for Eclipse tool is already developed once with the traditional code-centred approach, it is possible to compare the model-driven approach with the code-centred approach in this particular case.</p> <p>The case example shows that DSML and the code generator defined in this work can be used for generating repository-based Eclipse plug-ins. The code generator generates a fully functional Eclipse plug-in so the generated code does not have to be edited manually after its generation. Also variability in the software product family can be handled in a more flexible way in the model-driven approach.</p>		
ISBN 978-951-38-7094-2 (URL: http://www.vtt.fi/publications/index.jsp)		
Series title and ISSN VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)		Project number 7326
Date April 2008	Language English, Finnish abstr.	Pages 89 p.
Name of project MoSiS		Commissioned by
Keywords model-driven development, software product family, variability		Publisher VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4520 Fax +358 20 722 4374



Tekijä(t) Sivonen, Sanna		
Nimeke Sovellusaluekohtainen mallinnuskieli ja koodigeneraattori tietokantapohjaisten Eclipse-laajennusten kehittämiseen		
Tiivistelmä Eclipse on avoimen lähdekoodin alusta, jota käyttäjät voivat laajentaa hyödyntämällä Eclipse-alustan tarjoamia laajennuspisteitä. Eclipse-laajennukset kehitetään Java-ohjelmointikielellä ja kehitystyö voi olla haastavaa, erityisesti kehitettäessä useita hieman toisistaan poikkeavia sovelluksia samalle sovellusalueelle. Malliohjattu ohjelmistokehitys keskittyy ohjelmiston malleihin lähdekoodin sijasta. Sovellusaluekohtainen mallintaminen on eräs tapa toteuttaa malliohjattua ohjelmistokehitystä. Sovellusaluekohtaisessa mallintamisessa käytetään sovellusaluekohtaisia mallinnuskieliä yleiskäyttöisten mallinnuskielten asemesta. Tämän tutkimuksen tavoitteena on määritellä sovellusaluekohtainen mallinnuskieli ja koodigeneraattori tietokantapohjaisten Eclipse-laajennusten kehittämiseen. Mallinnuskielen ja koodigeneraattorin tarkoituksena on nostaa ohjelmistomallin abstraktiotasoa ja siten nopeuttaa tietokantapohjaisten Eclipse-laajennusten kehittämistä verrattuna laajennusten manuaaliseen ohjelmointiin. Myös käyttäjät, jotka eivät hallitse Java-ohjelmointikieltä, voivat kehittää tietokantapohjaisia laajennuksia Eclipseen tässä työssä kehitetyn mallinnuskielen avulla. Työssä kehitetyn sovellusaluekohtaisen mallinnuskielen ja koodigeneraattorin käyttöä havainnollistetaan generoimalla olemassa olevan tietokantapohjaisen Eclipse-laajennuksen lähdekoodi. Laajennus, jota käytetään havaintoesimerkissä, on arkkitehtuuritietämyksen hallintaan käytettävä Stylebase for Eclipse -työkalu, joka on VTT:n kehittämä avoimen lähdekoodin työkalu. Stylebase for Eclipse on tuoteperhe, jolla on useita varioituvuuspiisteitä. Koska Stylebase for Eclipse on aiemmin kehitetty koodikeskeisellä sovellusten kehittämistavalla, voidaan tässä työssä vertailla mallikeskeistä tietokantapohjaisten Eclipse-laajennusten kehittämistä koodikeskeiseen tapaan. Havaintoesimerkki osoittaa, että työssä kehitettyä sovellusaluekohtaista mallinnuskieltä ja koodigeneraattoria voidaan käyttää Eclipse-laajennusten kehittämiseen. Koodigeneraattori tuottaa toimivan Eclipse-laajennuksen, joten generoitua koodia ei tarvitse editoida manuaalisesti generoinnin jälkeen. Mallikeskeisessä kehitystyössä myös tuoteperheen varioituvuuden hallinta on joustavampaa.		
ISBN 978-951-38-7094-2 (URL: http://www.vtt.fi/publications/index.jsp)		
Avainnimeke ja ISSN VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	Projektinnumero 7326	
Julkaisu-aika Huhtikuu 2008	Kieli Englanti, suom. tiiv.	Sivuja 89 s.
Projektin nimi MoSiS	Toimeksiantaja(t)	
Avainsanat model-driven development, software product family, variability	Julkaisija VTT PL 1000, 02044 VTT Puh. 020 722 4520 Faksi 020 722 4374	

VTT PUBLICATIONS

- 660 Sihvonen, Markus. Adaptive personal service environment. 2007. 114 p. + app. 77 p.
- 661 Rautio, Jari. Development of rapid gene expression analysis and its application to bioprocess monitoring. 2007. 123 p. + app. 83 p.
- 662 Karjalainen, Sami. The characteristics of usable room temperature control. 2007. 133 p. + app. 71 p.
- 663 Välkkyne, Pasi. Physical Selection in Ubiquitous Computing. 2007. 97 p. + app. 96 p.
- 664 Paaso, Janne. Moisture depth profiling in paper using near-infrared spectroscopy. 2007. 193 p. + app. 6 p.
- 665 Ilmatieteen laitoksen palveluiden vaikuttavuus. Hyötyjen arviointi ja arvottaminen eri hyödyntäjätoimialoilla. Hautala, Raine & Leviäkangas, Pekka (toim.). 2007. 205 s. + liitt. 73 s.
- 666 Prunnila, Mika. Single and many-band effects in electron transport and energy relaxation in semiconductors. 2007. 68 p. + app. 49 p.
- 667 Ahlqvist, Toni, Uotila, Tuomo & Harmaakorpi, Vesa. Kohti alueellisesti juurrutettua teknologiaennakointia. Päijät-Hämeen klusteristrategiaan sovitettu ennakointiprosessi. 2007. 107 s. + liitt. 7 s.
- 668 Ranta-Maunus, Alpo. Strength of Finnish grown timber. 2007. 60 p. + app. 3 p.
- 669 Aarnisalo, Kaarina. Equipment hygiene and risk assessment measures as tools in the prevention of *Listeria monocytogenes* -contamination in food processes. 2007. 101 p. + app. 65 p.
- 670 Kolari, Kai. Fabrication of silicon and glass devices for microfluidic bioanalytical applications. 2007. 100 p. + app. 72 p.
- 671 Helaakoski, Heli. Adopting agent technology in information sharing and networking. 2007. 102 p. + app. 97 p.
- 672 Järnström, Helena. Reference values for building material emissions and indoor air quality in residential buildings. 2007. 73 p. + app. 63 p.
- 673 Alkio, Martti. Purification of pharmaceuticals and nutraceutical compounds by sub- and supercritical chromatography and extraction. 2008. 84 p. + app. 42 p.
- 674 Mäkelä, Tapio. Towards printed electronic devices. Large-scale processing methods for conducting polyaniline. 2008. 61 p. + app. 28 p.
- 675 Amundsen, Lotta K. Use of non-specific and specific interactions in the analysis of testosterone and related compounds by capillary electromigration techniques. 2008. 109 p. + app. 56 p.
- 677 Hanhijärvi, Antti & Kevarinmäki, Ari. Timber failure mechanisms in high-capacity dowelled connections of timber to steel. Experimental results and design. 2008. 53 p. + app. 37 p.
- 680 Sivonen, Sanna. Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins. 2008. 89 p.

VTT
PL 1000
02044 VTT
Puh. 020 722 4520
<http://www.vtt.fi>

VTT
PB 1000
02044 VTT
Tel. 020 722 4520
<http://www.vtt.fi>

VTT
P.O. Box 1000
FI-02044 VTT, Finland
Phone internat. + 358 20 722 4520
<http://www.vtt.fi>
