**MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING**

# Contributing To A Software Factory Framework

## An Integrated Domain-Specific Languages Projectional Editing Environment

Müllers, Bastien

*Award date:*
2021

*Awarding institution:*
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2020–2021

# Contributing To A Software Factory Framework : An Integrated Domain-Specific Languages Projectional Editing Environment

Bastien Müllers


Maître de stage :

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Vincent Englebert

Co-promoteur :

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

**Abstract**

Software factories are a promising application of Software Product Line Engineering (SPLE), enabling the mass customisation of software family members while reducing cost and effort.

Despite these promises, software factories haven't really been adopted by the industry, thanks to a difficult and costly development, and lack of tools and frameworks. Software factories also rely heavily on the use of custom-built DSL, which are still not commonly used due to high development costs.

The present thesis is a contribution to a software factory framework, introducing an editor for several generic internal DSL used to define software factories. We explore the idea of using Jetbrains MPS, a projectional language workbench, in the software factory DSL environment. The practical choices, as the technology and implementation details, will be justified through this thesis, besides to the programming tactics supported by the developed editors. We provide a qualitative evaluation of the editor behaviour according to several assessment criteria from the literature.

**Keywords**

Software Product Line, Software Product Line Engineering, Software Factory, Domain-Specific Language, Language Workbench, Projectional Editing

**Acknowledgements**

I wish to express my deepest gratitude to my supervisor, Professor Vincent ENGLEBERT, for helping me throughout this thesis and putting up with my erratic work habits.

I would also like to address my sincere thanks to Maouaheb BELARBI for the invaluable help she has been in proofreading this thesis and guiding me through the writing process.

I want to thank my friends Hugo DEVILLERS, Jérôme FINK and Guillaume MAÎTRE for providing me with great tips, guidance and remarks.

Finally, I wish to acknowledge the incredible support from my family, my mother Anne-Françoise, my sister Justine, and my aunt Sylvie.

**Résumé**

Les usines à logiciels sont une application prometteuse de l'ingénierie des lignes de produits logiciels, permettant la personalisation de masse des membres d'une famille de logiciels, tout en réduisant les coût et effort d'implémentation.

Malgré ces promesses, les usines à logiciels n'ont pas fait l'objet d'une adoption massive dans l'industrie du logiciel, à cause d'un dévelopment couteux et compliqué, et d'un mangue d'outils dédiés. Les usines à logiciels font aussi un usage approfondi de langages à domaine spécifique (DSL[1]) spécialisés, qui ne sont eux aussi que peu répandus, au vu d'un dévelopement assez cher.

Ce mémoire est une contribution à un cadre méthodologique d'usines à logiciels, présentant un éditeur pour plusieurs DSL utilisés pour définir l'usine. Nous explorons l'idée d'utiliser Jetbrains MPS, un atelier de langages projectionel, dans le contexte des DSL de l'usine à logiciel. Les choix pratiques, ainsi que la technologie et les détails d'implémentation, sont justifiés dans ce mémoire, avec les tactiques de programmation permises par l'éditeur. Nous proposons une évaluation qualitative de l'éditeur en se basant sur des critères d'évaluation repris depuis la littérature.

**Mots clefs**

Ligne de Produits Logiciels, Ingénierie des Lignes de Produits Logiciels, Usine à Logiciels, Langage à Domaine Spécifique, Atelier de Langages, Édition Projectionelle

[1] En anglais *Domain Specific Language*.

# Contents

# Chapter 1

# Introduction

## 1.1 Mass Customisation Of Software Products

The production of software products is like the production of any other goods. They can either be hand-crafted, fully customised yet expensive products designed to fit a specific customer's exact needs, or they can be mass produced, cheap products made to reasonably fit the needs of a large customer base. In the software domain, these are identified as individual and standard software, respectively. Both have drawbacks, the former being very expensive and the latter lacking diversification [1].

While customers want individualised products, the producers want to keep production costs low to increase their profit margins. To that end, producers found a compromise in the use of *common platforms*. The idea is to view each product as a combination of mass-producible standard components, and to assemble the final product according to individual customer specifications. In software, we call that production process a *software product line* (SPL). Just like in tangible goods production, SPL cover families of products [1]. Each family can be associated to a specific domain, in which different software products share common behaviours.

In SPL, variability - the way in which members of a software family vary from one another - is typically handled manually. This is reasonable on a small scale, but does not follow on a large scale. Complex variability calls for a different approach. The software factory is such an approach, making use of domain-driven engineering to specify automate and pilot the SPL [2]. Software factories, although promising for reducing production costs of many unique but related software products, are quite costly to develop. The key to the industrialisation of software is to reduce the cost-benefit trade-off, by enabling the cost effective construction and operation of software factories [3].

## 1.2 Domain-Specific Languages

One of the ways software factories make use of domain-driven engineering is by using domain-specific languages (DSL). DSL differ from *general purpose* languages (such as Java and C) by sacrificing some of the flexibility to express *any* program for productivity and conciseness of relevant programs in a particular domain [4].

Programming languages are typically implemented by creating a compiler or interpreter. While this method may perfectly apply to a DSL, it comes at a high cost and requires experienced developers to carry out. Such a cost isn't always justifiable for a typically light-weight, focused solution [5]. Another drawback to this approach is the lack of reuse from other language implementation. As an alternative, a DSL can be implemented by extending a given base language. While this approach is generally cheaper, it comes with its own drawbacks. The reuse of another language's implementation can often force compromise in the DSL's syntax. It can also impact the quality of error messages, which are sometimes limited to the base language's domain and are thus much less expressive [6]. Because of these drawbacks, DSL don't play the role we would expect in the software development life cycle [5].

However, more and more tools are being developed to ease the development of DSL. These tools, commonly known as *language workbenches*, provide integrated development environments (IDE) complete with high level tools for implementing, evaluating and maintaining DSL [5]. They reduce the cost and knowledge required to implement a DSL, yet have none of the drawbacks usually encountered when extending a pre-existing language.

## 1.3 Objectives, Motivations and Structure

The objective of this thesis is to contribute to a software factory framework by implementing an integrated editing environment for a series of DSL. By doing this we also wish to evaluate the fitness of a language workbench in the context of a software factory, and if using it can positively impact the cost of developing one.

Reducing the cost and effort of building software factories could bring about a new, effective and cheap way of producing software. While this is certainly a long way ahead, we believe it to be possible and a worthwhile investment. Although this contribution may be small, we hope the insights it brings will help pave the way for future innovation, in any measure it can.

We will first dress a state-of-the-art of SPL engineering, DSL implementation, as well as projectional editing which is the approach we've taken for this thesis. Then, we'll present the languages in detail, going through what they represent, their abstract and concrete syntaxes, as well as the way in which these languages interact with each other. We will follow by presenting the way in which we've implemented the editors, with special emphasis on our choices of technology and implementation approaches.

To evaluate our proposed editing solution, we will proceed to a qualitative assessment. This assessment is based on evaluation criteria taken from the literature. As the low adoption rate of DSL is mostly influenced by high development costs, we will also reflect on the development process. Finally, we will explain and discuss the conclusions we came to from this experiment.

# Chapter 2

# State of the art

In this chapter we will go over some of the most relevant sources available in the literature to deliver a summary of the key concepts that will be used in the rest of this thesis. These are divided in three sections, corresponding to three different domains, yet overlapping in the context of this thesis : Software Product Line Engineering, Approaches for Domain-Specific Language Implementation, and Projectional Editing.

## 2.1 Software Product Line Engineering

In [1], Pohl defines software product line engineering (SPLE) as : *«a paradigm to develop software [...] using platforms and mass customization.»* A *software platform* is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced. In summary, a software product line (SPL) is a family of software sharing a common software platform. *Mass customization* is the large-scale production of goods tailored to individual customers' needs.

Software derived from a common SPL belong to the same domain, and differ from each other according to variable aspects.

### 2.1.1 Variability Management

Instead of understanding each individual system by itself, which is the classical approach in software development, SPLE views it differently. The pillar principle behind the SPL paradigm consists in considering the SPL as a whole *and* the variations between the systems. Software within a SPL may support many different individual customers, and therefore managing *variability* efficiently is a key aspect of SPLE. This variability must be defined, represented, exploited, implemented, evolved, etc. This process is called *variability management* [7].

To fully understand variability, we need to start by stating a few key concepts [1] :

- **Variability subject** : a variable item of the real world or a variable property of such an item. A variability subject is, essentially, what varies. A *variability object* is a particular instance of a variability subject[1]. For instance, let's say a company produces coffee cups. The cups can be red or blue. The colour of the cup is a variability subject, while blue and red are variability subjects.

- **Variation point** : a representation of a *variability subject* within domain artefacts enriched by contextual information. This applies to all kinds of development artefacts. The contextual information encompasses the details about the embedding of the variability subject into the SPL, such as the reason why the variation point was introduced. An example of that could be the way in which an application stores personal data. The reason for that variability subject is that different countries have different laws about personal data protection to which the application must comply, if the customer chooses to distribute the application in these countries.

- **Variant** : a representation of a *variability object* within domain artefacts. A variant is a variability object whose subject is a variation point. Continuing on with the previous example, we can imagine a variant as a GDPR-compliant personal data storage strategy.

Furthermore, a variability can be described as either internal or external [1]. The former, *external variability*, is variability that is visible to the customer. The customer can choose the variants they need. The latter, *internal variability*, is variability that is hidden from the customer. Resolving the internal variability is up to the stakeholders representing the providers of the SPL. External variability can be caused by differences between stakeholder needs, as those would imply having a variant for each specific need, left to the customer to choose. In addition, external variability can be introduced because of differences in laws and standards that apply to the domain of the SPL. Internal variability often emerges when refining external variability. The realisation of the options proposed to the customer, generally at a high abstraction level, often demands fine tuning of several options at a lower abstraction level. Since the customer is only interested in high-level decisions, these options need not be communicated to the client. Similarly, internal variability realisation can lead to other lower-level internal variability. Finally, technical issues generally don't have to be considered by the client, and therefore can cause internal variability [1].

Since managing variability is crucial to ensure efficient SPL processing, modelling variability is paramount. Pohl introduces an *orthogonal variability model* to accurately model the variabilities in a SPL. An orthogonal variability model is a model that defines the variability of a software product line. It provides a cross-sectional view of the variability across all software development artefacts. An example of such a model is shown on Figure 2.1.

To complete the variability model, we need a way to describe the relations between variants that belong to different variation points. For that purpose, Pohl introduces two types of dependencies : *requires* dependency and *excludes*

[1] This definition describes variability in the broader sense, not restricted to the context of the software product line.

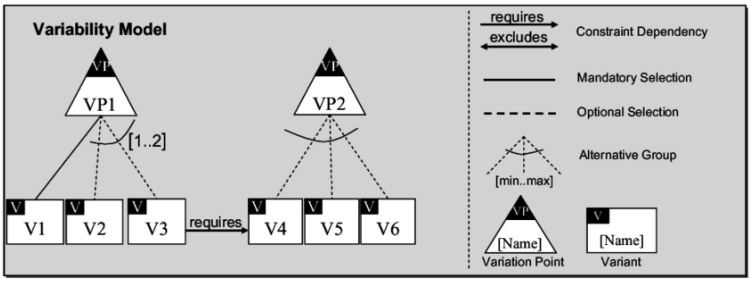


Figure 2.1: An example of an orthogonal variability model, taken from [8]

dependency. These dependencies are called variability constraints, and can be expressed between two variants, two variations points, or between a variant and a variation point.

The variability model on its own is not able to represent the full meaning of variability in SPLE. In addition, we need the traditional views on design and requirements, and the relation between the variability and these views so that we know how variability will impact these views [7].

### 2.1.2 Software Factories

The key objective of SPL is to optimise the reusability of software artefacts across different, yet similar in domain, software products. While achieving this is an obvious benefit in the long term, software product lines allow for the realisation of something even more interesting : *economies of scope*. Economies of scope differ from economies of scale in that instead of enabling mass production, they enable *mass customisation*. In software production, economies of scale happen quite naturally, in the distribution of copies of a same software. Economies of scope arise when the same styles and patterns are used to develop multiple related designs, and the same artefacts are reused for their implementations [3].

To achieve economies of scope in a software product line, we need a *factory* to automatically produce these unique software products. A software factory systematically captures knowledge of how to produce the members of a specific SPL, makes it available in the form of assets, and then systematically applies those assets to automate the development of the family members [3]. A software factory reduces the cost, time, and effort for each individual software it produces, yet designing and implementing a software factory is also very costly and difficult.

In [3], the software factory is defined as a *model driven product line*, a SPL automated by metadata captured by models using DSL. To achieve that, a software factory developer has to develop DSL tools to edit the models and translate those models into executables or lower-level specifications.

## 2.2 Approaches for DSL Implementation

Implementing a DSL from scratch using common general-purpose language (GPL) approaches and technologies is not always viable due to the high cost of development not being worth investing in a specialised solution. For that reason, there exist several language development approaches most commonly associated with DSL.

We can classify those approaches under two main categories : internal and external DSL. Internal DSL are *embedded* within another language, known as the base language [5]. External DSL are independent from any base language.

Another way of implementing a DSL consists of using a *language workbench*, a term coined by Martin Fowler [9]. Language workbenches are IDE made specifically for the development of languages which provide tools that support the definition and edition of both the syntactic and semantic aspects of the language in development.

### 2.2.1 External DSL

Domain-specific languages are first and foremost *programming languages*. Any and all GPL implementation approaches also work for DSL. Therefore, the first approaches to detail when talking about their implementation should be those commonly associated to GPL.

In the classical GPL implementation process, there are two main approaches : compilation and interpretation. In compiled languages, the source program written in a high-level language is fed into a program called a compiler. The compiler reads and translates the input program from a high-level language into a low-level language which can then be executed. In interpreted languages, a program called the interpreter reads the input program and directly executes it.

Both of these approaches require first a parser. The parser reads the input stream and generates an *intermediate representation* (IR), a series of operations and operands generally in the form of a parse tree. Then, for the compiler or interpreter to make use of that data, we need to turn that IR into a tree data structure, called an Abstract Syntax Tree (AST). By visiting it, the compiler generates the corresponding low-level code, and the interpreter executes the corresponding actions [10]. There exist a variety of parser generators, tools designed to offer the language developer a fully-functioning parser, simply based on a grammar definition. One of the most commonly used parser generators is ANTLR[2].

This approach, when applied to DSL, gives what is called an *external DSL*, a DSL that is not dependant on any base language. The main advantages in building an external DSL are, firstly, the total absence of constraints regarding syntax, primitives and other aspects of the language that could be restricted if it was embedded within a base language instead. Secondly, this approach offers the ability to do error detection and static analysis at *domain level* [6].

Both compilers and interpreters are difficult programs to build and maintain,

[2] https://www.antlr.org/

requiring a great deal of technical knowledge, skill, time and effort to achieve. This drives the cost of these approaches quite high, which, when applied to DSL, often makes the potential benefits of having a specialised solution too small to justify the cost, even with the advantages it presents [5].

### 2.2.2 Internal DSL

Implementing a DSL from scratch is costly and requires a lot effort. An alternative to that option is to implement a DSL by extending a pre-existing language. DSL implemented this way are known as *internal languages*, and the language they extend is known as the *base language*. Several approaches to internal DSL exist, differing from each other in the way the base language's implementation is reused. The main advantage of the following approaches is their ability to make use of the features provided by the base language [6] :

**Preprocessing** : This approach relies on a *preprocessor* to translate the DSL constructs into base language constructs. This means that the static analysis is entirely performed at the level of the base language, which severely decreases the quality of the error messages who can only be expressed with base language concepts. However, this approach is very easy [11].

**Embedding** : Existing mechanisms in the base language are used to build a library of domain-specific operations [6]. The implementation of the base language's compiler or interpreter is reused as is. This approach can severely limit the DSL notation in order to fit the base language.

**Extensible compiler/interpreter** : The base language's compiler/interpreter is extended to support DSL constructs. This approach supports better type checking and optimization than the previous two, while requiring less effort than building a compiler or interpreter from scratch, but it is also very hard to carry out.

**Commercial Off-The-Shelf** : An existing tool or solution is applied directly to a specific domain, for example XML-based DSL. This approach can be feasible for particular domain problems, like, in the case of XML, processing and querying documents.

### 2.2.3 Language Workbenches

Language workbenches are IDE made specifically for DSL developers. In practice, they facilitate the implementation and maintenance of DSL with high-level tools. Some even apply domain engineering to language development, like Jetbrains MPS[3], which provides high-level DSL designed for language development.

A language workbench must support notation, semantics and an editor for the language in development. In [12], the authors sum up language workbenches components in a complete feature model. Each main feature is subdivided in a variety of sub-features, and the authors provide a complete description of each, as well as a graph comparing language workbenches and their supported features.

[3] https://www.jetbrains.com/mps/

Existing language workbenches are described around the 4 following aspects, as mentioned in [5]. Table 2.1 provides a comparison of several language workbenches based on the first three of these aspects.

**Language structure aspect.**

The structure of the language in development serves to represent the domain concepts and their relations. There are two ways in which a language workbench lets a developer define the language structure : the *grammar-driven approach* and the *model-driven approach*.

The grammar-driven approach is based on a *grammar definition*, in the form of a Context Free Grammar (CFG). It describes the *concrete syntax* of the language and can be defined using meta-languages like BNF[4], SDF[5], or a custom-built DSL.

The model-driven approach utilizes *meta-modelling* for defining the language structure. In this scenario, a program is a model conforming to the meta-model designed by the developer, which defines the *abstract syntax* of the language. The concrete syntax is determined by a projection, done by the editor. This approach is simpler than the grammar-driven approach because the language developer only defines a semantic model.

**Language editor aspect.**

The language workbench allows the developer to define how the user sees and edits the programs. The editor should also support other standard features, like highlighting, error checking and code completion. There are two main approaches to program editing : parser-based and projectional editing.

The parser-based approach relies on the use of a parser to generate the AST from a text stream. Users interact with the concrete syntax. This approach is the main approach used by grammar-driven workbenches. The parser is generated from the grammar definition, in the same way a parser generator would. In most cases, parser-based workbenches only support textual notation.

Projectional editing uses the abstract representation and allows the user to edit the AST directly, without parsing. The projection generates a visual representation directly from the AST, which allows any type of notation to be used, as needed. This approach is used by model-driven workbenches.

**Language semantics aspect.**

This aspect relates to how the language developer defines how the programs are executed. There are two ways to approach this : translational semantics and interpretive semantics.

In translational semantics, the language is translated into another language, usually a GPL. This can be performed either via model transformation (model-to-model) or code generation (model-to-text). The former consists in building the target AST while traversing the source AST, and the latter in translating the source AST into the corresponding target language code directly. The main

[4]Backus-Naur Form
[5]Syntax Definition Formalism

advantage of model transformation is to support *bidirectional mapping* between the source and target AST.

Workbenches using interpretive semantics let the developer define *semantic actions*, to be executed while traversing the AST.

**Language composability aspect.**

DSL are built to express concepts of a single domain, while the problem to be solved by the language developer might be expressed over several domains. Language composability is the way several DSL can be composed with each other. Language composition is described in detail in [13]. The authors identify four types of language composition : language extension, language unification, self-extension and extension composition.

A *language extension* is essentially a language fragment added over a base language. The extension is dependent on the base language, and makes no sense by itself. This often implies modifying the base language implementation. A language-development system is said to support language extension if the implementation of the base language can be reused unchanged to implement a language extension.

*Language unification* of two independent languages means a bidirectional composition, where elements of each language are free to interact with elements of the other. It is hard to do in practice, requiring that parts of the languages be equalized. Often, the unification of the language implementations has to be done by hand. A language-development system is said to support language unification of two language if the implementation of both languages can be reused unchanged by adding glue code only.

*Self-extension* reflects the cases where a language is extended by *embedding* a DSL into it with a base language program. A language-development system is said to support self-extension if the language can be extended by programs of the language itself while reusing the language implementation unchanged.

*Extension composition* reflects the ability of a language-development system to support the composition of language extensions of a base language, that is, if the extensions can work together. Systems that support language unification also support unification of their extensions. In systems that support only language extension, or self-extension, there are three cases : no support for extension composition, support for incremental extension (where an extended language can in turn be extended, and so on), and support for extension unification (where two extensions can be unified by adding glue code). In addition, systems that support self-extension support another form of extension composition : self-application, where a host language extension can be used in the implementation of another extension.

In parser-based workbenches, composability is supported by merging the grammars of the languages. In projectional workbenches, language composition is supported by integration of the abstract representations.

| | | Model-driven structure Projectional editor | | | | Grammar driven structure Parser-based editor | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | MPS | Màs | MetaEdit+ | WholePlatform | Xtext | SugarJ | Ensō | Spoofax | Rascal |
| Supported notations | textual | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | tabular | ✓ | ✓ | ✓ | ✓ | | | | | |
| | symbolic | ✓ | | ✓ | ✓ | | | | | |
| | graphical | | | ✓ | ✓ | | | ✓ | | |
| Supported semantics | model-to-model | ✓ | | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| | model-to-text | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| | interpretive | ✓ | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ |

Table 2.1: *Various language workbenches supported notations and semantics*

## 2.3 Projectional Editing

Projectional editing allows users to directly edit the AST of the program via a *projection* - a rendering of the AST using projection rules. This approach to program editing allows the use of arbitrary notations, as well as graphical and textual notations, sometimes all at once.

The suitability of a language for its target audience is guided by many criteria, but among them proper notation is especially important. Another one of those criteria is composability. Global-purpose languages generally use either textual or graphical notation, yet many real-world languages require a mix of graphical, textual, tabular and symbolic notations. Projectional editing supports this [14]. Moreover, projection-based language systems support composability more easily since it is based on the integration between the abstract representations instead of the grammar rules [5].

### 2.3.1 Usability of Projectional Editors

Despite their advantages, projectional editors haven't seen much adoption in practice. For projectional editors that use mostly textual notation, users tend to expect the editor to behave in the same way a parser-based editor would. This is rarely the case, and users often find themselves compelled to know and understand the structure of the AST they're editing. Projectional editors also make version control system (VCS) integration more complicated, since the program isn't stored as text, but rather as a serialized AST [14]. These problems cause poor editor usability for experienced developers and complicates the classical VCS-centred workflow.

In [14], the authors hypothesize that these drawbacks are the main reason behind the low adoption rate of projectional editors. They conduct a study in which they evaluate how well MPS performs against traditional parser-based editors in 3 aspects : efficiently entering code, selecting and modifying code, and infrastructure integration.

For efficiently entering code, their results indicate that, after a learning phase, MPS lets developers work efficiently and productively. They also concluded that, except for supporting comments, MPS addresses the issue of selecting and modifying code quite well. Finally, they use the mbeddr[6] project as a professional experience to evaluate the effectiveness of MPS infrastructure-

[6] http://mbeddr.com/

integration support. They conclude that MPS supports infrastructure integration well enough to be usable in practice.

### 2.3.2 Efficiency of Projectional Editors

The efficiency of projectional editing has been compared to that of parser-based editing in [15]. The authors also compared the efficiency of experienced users of projectional editing to beginners, and tried to identify the commonalities and differences between projectional and parser-based editors.

They first conducted a controlled experiment with students who had no experience with projectional editing and compared their efficiencies with the control group. They found that editing efficiency was quickly achievable with a projectional editor. With a short 45-minute training, students were able to achieve efficiency comparable to the ones using the parser-based editor.

The second experiment compared the efficiency of the inexperienced students with that of professionals with experience using projectional editors. Their results show that, for basic editing operations, more experience with projectional editing did not lead to significantly better efficiency. However, in advanced editing, more experience with projectional editors lead to significantly better efficiency than the inexperienced users, and even the parser-based editor users.

For the last question, they conducted a fine-grained analysis of editing operations and errors. They found that writing code was not negatively impacted by projectional editing, but relied on increased use of code completion. Selecting code required more experience and attention. Projectional editing showed a trade-off between fewer mistakes and an increased complexity of editing operations, yet experience reduced such problems, leading to fewer errors. Projectional editing fostered a shift from *copy-paste* strategies towards operations that work well on the AST, which requires an increased understanding of the underlying AST.

## Conclusion

In this chapter we presented a state-of-the-art of the relevant concepts used for the rest of this thesis. We gave a presentation of the general context in which this thesis takes place - the software factory - and explained the relevant concepts we will be using in the rest of our thesis. We also explained the available approaches for implementing a DSL, and compared their advantages and drawbacks. Finally, we discussed the question of projectional editing when compared to the traditional parser-based approach. In the next chapter, we will be presenting the languages we implemented editors for.

# Chapter 3

# Case Study : The «Methodology» Collection Of Languages

The objective of the present thesis is to implement an editing environment for several given DSL for a software factory. In this chapter, we will present the considered software factory DSL, their abstract and concrete syntaxes and the ways in which these languages are composed together. The figures and listings presented in this chapter are all taken from [2].

## 3.1 Overview

The DSL editor we have implemented in this thesis supports a collection of languages defined in [2]. This collection of languages is a proposal for a methodological framework used to describe the behaviour of a software factory. Programs written in these languages are to be used to orchestrate the assembly of software product assets into the finished product. The domain expressed by these DSL is software engineering, specifically within the context of a feature-oriented SPL.

The methodology will be applied to a feature model, using a feature modelling language called *FeatSimple*. This language is a simplified version of the *FeatAll* language described in [2]. The central aim of the methodology is to describe a strategy with which to implement the features in the model. To that end, it uses the *Strategy* language, which describes software product line implementation strategies, and the *Tactic* language, which describes specific implementation tactics. To describe the orchestration of tactics within a strategy, the *Recipe* language is used. In the context of this thesis, the *Recipe* language is completely enclosed within the *Strategy* language. As such, both are considered together, as one.

The meta-model on Figure 3.1 features the concepts of the 3 languages of the methodology framework, and their relations. The concepts of each language

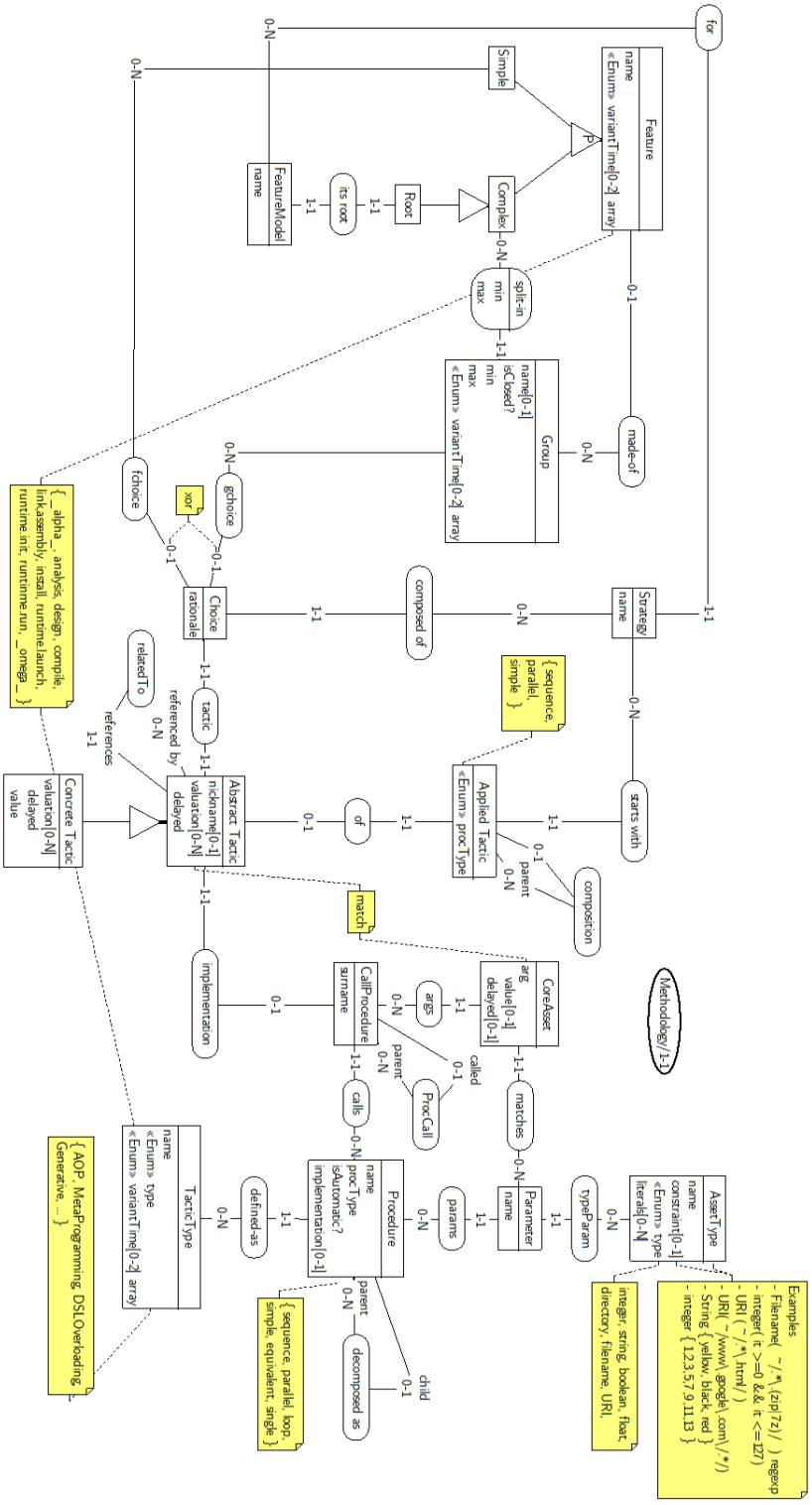


Figure 3.1: The Complete Methodology Meta-Model

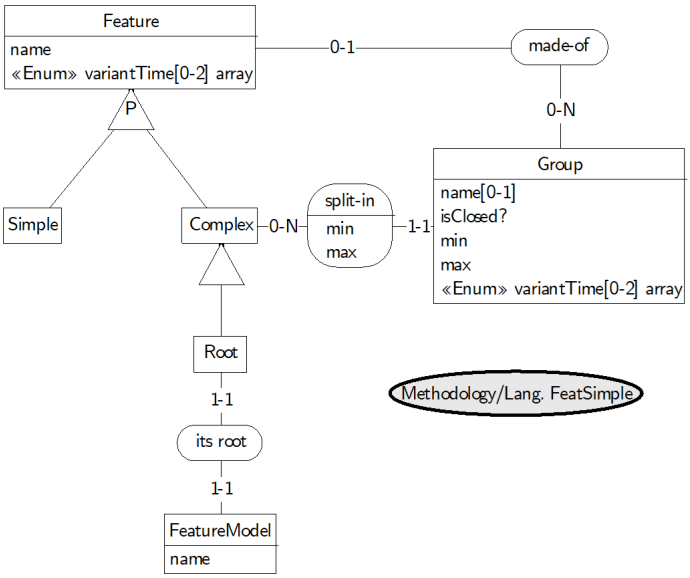


Figure 3.2: FeatSimple Language Meta-Model

and their relations will be explained in the corresponding sections, while the relations between the languages will be explained in the composition section.

These languages use a domain-specific primitive type in the form of an enumeration, called *variantTime*. A variant time is the time at which the variant (i.e. the feature) is to be introduced in the system. The values of this enumeration are therefore directly taken from the software life cycle lexicon. These values are specified on Figure 3.1.

## 3.2 The FeatSimple Language

The FeatSimple language is a simplified version of a feature modelling language called FeatAll[1]. A program written in this language aims to represent a feature model and follows the meta-model described in Figure 3.1.

A feature model is composed of features organised in a tree structure, originating from a single feature called the root. The FeatSimple language distinguishes between simple and complex, the former being restricted to the leaves of the tree. These are the elementary, indivisible features. Complex features act as the nodes of the tree. They can be made up of several sub-features organised in groups. Groups correspond to a variation point, and the features in these groups to variants. The feature model expressed by this language is interpreted by the methodology as the common thread to the production of any particular system.

[1] [2] page 143

### 3.2.1 Abstract Syntax

The root node of the FeatSimple abstract syntax tree is the *FeatureModel* node, which is an instance of a feature model. A feature designates a simple configurable unit of the system or a composed functionality. All the features are reified from the most abstract feature known as the *Root* and situated at the highest level of the tree. A feature model must always have exactly one root. *Simple Features*, as opposed to *Complex Features*, are the leaves of the AST, depicting the low-level features that cannot be further subdivided. Complex features can be divided in *Groups* of features. Note however that complex features don't have to be subdivided.

A feature model has a name attribute, and so do all features and groups of features. Features and groups also have a collection of between 0 and 2 variant times. Groups also have an *isClosed* boolean attribute. If set to false, this attributes allows the group to be extended with more features in the future. On top of that, groups have multiplicity attributes, corresponding to how many features have to be selected from the group at minimum and maximum. Finally, multiplicity attributes also exist for the parent-children relation going from a complex feature to groups. These attributes express how many times the group can be replicated within the parent feature, at minimum and maximum.

### 3.2.2 Concrete Syntax

A FeatSimple program starts by declaring a feature model (**FeatureModel**), followed by a single feature denoting the root of the model. Those are declared using the **Feature** keyword, while **bindingTime** is used to define their variant times. Complex features also list a series of feature groups (**Group**). When declaring a group, a multiplicity is written in brackets before **Group** and another in parentheses after the group's name. The former corresponds to the feature-to-group *split-in* relation multiplicity attribute and the latter the the group's own multiplicity. In addition to variant times, groups must also declare whether they are open or closed (**isClosed**), and declare the features that make up the group.

```
FeatureModel "Tourist"
{
  Feature "Application"
    bindingTime = Compile .. Runtime.launch
  {
    one Group "functionality" any
    {
      Feature "PointsOfInterest"
      Feature "Positionning"
      Feature "Quizz"
      Feature "ePayment"
        bindingTime = Compile
    }
  }
}
```

Listing 3.1: FeatSimple Program Example

As shown in Listing 3.1, specific multiplicities can also be declared using certain keywords. For the split relation, those are : *one* for [1-1], *many* for [0-n] and *optional* for [0-1]. For the feature group, those are : *xor* for (1-1), *or* for (1-n), *any* for (0-n) and *all* for (n-n).

## 3.3 The Tactic Language

The Tactic language is used to represent a generic tactic for implementing a feature at given variant times. Those generic tactics are called *tactic types*. A tactic type describes a way to implement a feature. It is described by a procedure, with typed parameters. The types of these parameters are called *asset types* and are described within the tactic program. Asset types are of a basic type and can be complemented by literals or a constraint. A procedure can be implemented by an executable script or code asset referenced in the program, in which case it is considered automatic. Otherwise, a procedure can also be implemented manually. In that case, it will reference a human-readable implementation guide that can be expressed as a text file, pdf, schema, or sometimes even nothing at all.

### 3.3.1 Abstract Syntax

The abstract syntax of the Tactic language is given by the meta-model shown on Figure 3.3. The main component of the AST is the *TacticType* node which is defined as a collection of *Procedures*. The *Parameters* of the procedure are all linked to an *AssetType*. Procedures can be further decomposed as child procedures.

Tactic types are of a certain type, which corresponds to a variability implementation technique (e.g. Aspect Oriented Programming). Just as the features in FeatSimple, they too correspond to 0, 1 or 2 variant types. Tactic types are named, and so are procedures, parameters and asset types. These latter have a type too, which is expressed as an enumeration whose values can be found on Figure 3.1. Additionally, asset types can be complemented by a list of literals, or a constraint. Procedures can point to an implementation, although that doesn't necessarily mean executable code (e.g. a written procedure, list of instructions, a pdf, etc...). The *isAutomatic* boolean attribute tells whether or not the procedure can be executed automatically, by a computer. Procedures also have types, which influence the way their decomposition into child procedures works. The types are as follows :

- **sequence** : the child procedures must be executed in sequence.
- **parallel** : the child procedures can be executed in parallel.
- **equivalent** : the child procedures are all equivalent and represent different alternatives. Only one is executed.

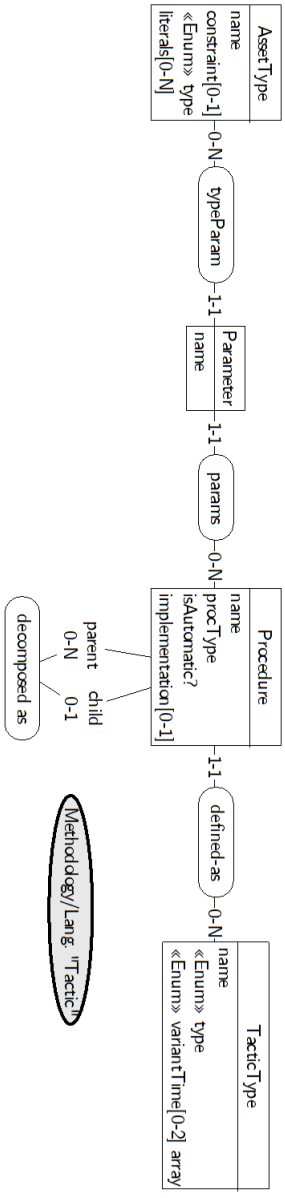


Figure 3.3: Tactic Language Meta-Model

- **optional** : the procedure does not have to be executed.
- **loop** : the procedure can be repeated, but in sequence.
- **parloop** : the procedure can be repeated, in parallel if necessary.
- **simple** : the procedure is executed exactly once.

Only the first three types of procedures can be reified into child procedures.

### 3.3.2 Concrete Syntax

A Tactic program first begins by declaring the asset types :

```
declare new AssetType FileText Filename;
declare new AssetType How String { "before", "after" };
declare new AssetType FileZip Filename(*.zip ; *.7zip);
```

Listing 3.2: AssetType Declarations Example

The declaration begins with the keywords **declare new AssetType**, followed by the name and the type of the asset type and ending with a semicolon. Optionally, one can declare a constraint between parentheses or add literals between braces. The developer can declare any number of new asset types at the beginning of a program. That is followed by declaring a single tactic type :

```
TacticType ByInsertion
  type = "code modification" ;
  bindingTime = compileTime ;
  { procedure parallel Insert
    {
      loop procedure AlterFile(in : FileText;
                               how : How;
                               where : String;
                               what : FileText);
      procedure AppendProject(where:Path ; what:FileZip);
    }
  }
```

Listing 3.3: Complete Tactic Program Example

The tactic type is declared with the **TacticType** keyword followed by its name. The keywords **type** and **bindingTime** are used to set the type and variant times, respectively. Then, a list of procedures is declared with the keyword **procedure**, preceded by the procedure type and followed by the name and a list of parameters. These are declared as a name, a colon, and the name of an asset type previously declared. If the procedure type allows it, child procedures can be declared too, in brackets and separated by semicolons.

## 3.4 The Strategy Language

The Strategy language is used to model a strategy : the way in which to use tactics to implement the features on the feature model. A strategy is therefore composed of a set of choices concerning the tactic (concrete or abstract) used to implement a variant or variation point. The difference between concrete and abstract tactics lies in the valuation of the arguments of the procedure they call. A concrete tactic must declare a value for each argument, where an abstract one can set arguments to a special *delayed* value. This means that other tactics can reference an abstract tactic and pass values to delayed arguments.

A Strategy program also defines a *recipe* to be followed in order to carry out the strategy. The recipe utilises the tactics defined by the strategy to create an organised tree-structure.

### 3.4.1 Abstract Syntax

The abstract syntax of the Strategy language is described by the meta-model shown in Figure 3.4. The main component of the Strategy AST is the *Strategy* concept which will be defined by one or several *Tactics* given by the *Choices*. A concrete tactic can pass value to its parameters, while an abstract one cannot and must therefore delay their valuation. Besides, it can reference other tactics, and be implemented by a *Procedure Call*. The latter can have a list of *Core Assets*, which correspond to the called procedure's parameters. A procedure call can also call child procedures calls. The strategy is also defined by what it starts with : an *Applied Tactic*. Those are the application of a previously defined tactic, to which they are linked, and can be composed of other applied tactics.

The strategy, tactics, procedure calls and core assets have a name attribute. The choices all have a rationale attribute, to justify that choice. Tactics also have a list of values for their procedure call arguments, which can be either delayed or set to a value (only delayed in the case of an abstract tactic). The core assets also have a value which can be delayed. The list of values in a tactic must match the values of its procedure call core assets. Applied tactics are of a certain procedure type. The possible procedure types are the same as Tactic's procedure types, only limited to *sequence*, *parallel* and *simple*.

### 3.4.2 Concrete Syntax

A strategy is declared using the **Strategy** keyword, followed by its name and the name of the feature model the strategy is being written for, using the **for FeatureModel** keywords, as shown on Listing 3.4.

The developer must then declare the list of tactic choices that make up the strategy, between braces. Each choice construct specifies first its name and then which simple feature or feature group it implements using the keyword **implements**. Thereafter, the developer specifies the tactic, which can be either abstract or concrete. This is carried out using the keywords **with (abstract)**

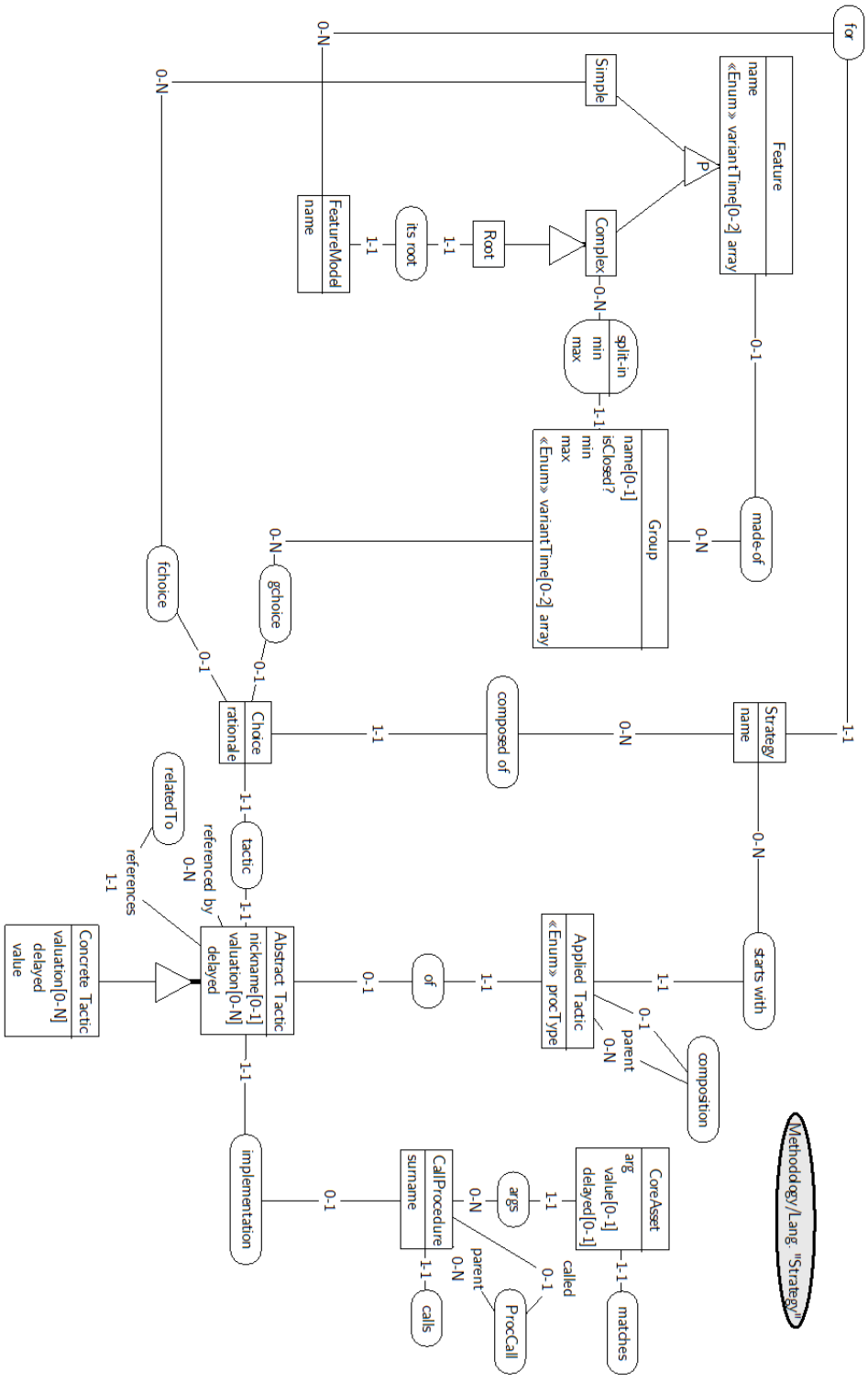


Figure 3.4: Strategy Language Meta-Model

**TacticType**[2] followed by its name, and finally the rationale behind that choice (**because**). In braces, the developer will then declare the procedure calls using the keyword **call**, followed by the name of the procedure, previously declared in a Tactic program. Following that come the arguments and their values. This process is then repeated for the child procedures, if needed.

```
Strategy "default" for FeatureModel "Tourist"
{
  funct : implements Group Application.functionality
  with abstract TacticType "ByInsertion"
  because "Resource files and Main.java must be patched
           to adapt the interface and the behaviour.
           The ad-hoc code must be added to the project."
    {
      call AlterFile( in= "src/Main.java",
                      how ="after",
                      where = "// insert feature here",
                      what = $whatJava );
      call AppendProject( where= "src/",
                          what= $whatProject) ;
    }
}
Recipe {
  simple{
    funct
  }
}
```

Listing 3.4: Strategy Program Example

The program ends on a "recipe" section (**Recipe**) where the applied tactic is declared by writing its type and the name of the tactic it corresponds to (declared in the Strategy program), as well as its child applied tactics if necessary.

## 3.5 Composition

At the centre of the Methodology is the Strategy language, whose programs are written *for* a feature model, and *using* tactic types. Thus, a strategy *imports* a single feature model, as well as several tactic types and is, as such, dependent on both languages to be able to use their concepts. The other two languages however don't need to be able to manipulate Strategy constructs. Since the dependencies only go one way, language unification is not necessary.

Two relations between strategy and feature model concepts exist : one where the strategy node references a feature model, and one where a choice node references either a group of features or a simple feature. With tactic types, there are three : one for the tactic to reference a tactic type, one for the procedure call to reference a procedure, and one where a core asset references a parameter.

[2]The abstract keyword here is optional, and by its presence defines the tactic as abstract.

Several meaningful constraints have to be applied to certain strategy node attributes to guarantee the model makes sense.

- The features and feature groups referenced in a choice must be declared in the feature model referenced in the strategy.
- The parameters referenced by a core asset must correspond to one of the parameters that belong to the procedure called by the core asset's parent procedure call.
- The procedure referenced in a procedure call must be declared in the tactic type referenced by the parent tactic node.

## Conclusion

In this chapter we have presented the syntactic rules of the three software factory languages we will be developing an editor for. We have given an overview of their abstract syntaxes by explaining their meta-models. Several examples were given for the concrete syntaxes, supported by explanations. Finally, we discussed the way in which these languages interact, and listed the constraints it implies on the syntax. In the next chapter, we will detail the implementation of this editor, as well as the various technology-related choices we made.

# Chapter 4

# Methods

In this chapter we will go over the implementation of the editing environment for the languages we introduced in the previous chapter. First, we will explain what approach and technology were chosen to carry it out, along with the reasons behind this choice. Then, we will detail the changes and adjustments that were made to the original syntactic rules to fit the technology limitations. We will conclude this chapter by explaining the projection of the AST.

## 4.1 Technology Used

We have made the choice to implement this editing environment with Jetbrains MPS. This choice was motivated by a series of factors which will all be detailed in the following subsections. The first three arguments are made from the language-developer point of view, while the remaining two are set from the end-user point of view. From the start, we made the choice to use a language workbench. This was an easy choice to make, as it offered the best option for minimizing both the development cost and effort, while not limiting the expressibility of the syntax. MPS presented itself as the state-of-the-art language workbench in that regard.

A capture of the entire MPS window is shown on Figure 4.1. It was taken from the MPS project editor, but the generated Methodology DSL editor is almost completely identical. The MPS project files of the implementation can be found here : `https://github.com/bmullers/MethodologyMPS`.

### 4.1.1 Documentation and Support

One of the deciding factors behind this choice was centred around the ease of learning the technology. Realistically, this is still one of the main reasons behind the slow adoption of DSL in software development. Besides, since developers are considered as important stakeholders in the software development life cycle, they will generally oppose the use of complicated and poorly documented technologies. To this regard, MPS performs very well, providing extensive online

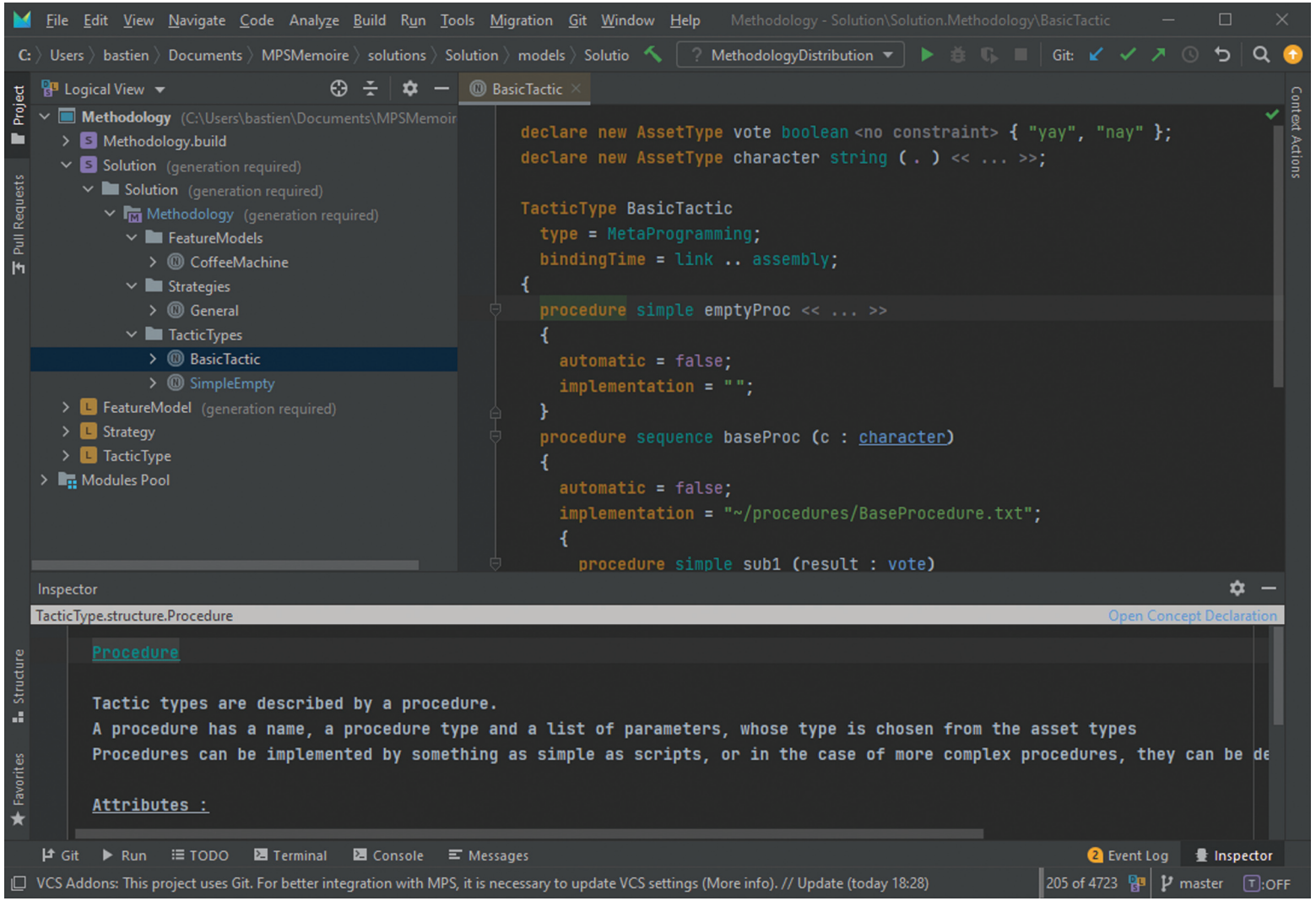


Figure 4.1: Screen Capture of the MPS Editor

documentation in the form of user guides, tutorials, example projects, videos and an active technical support forum. MPS is also an active open source project, with its own bug tracking and reporting tool, as well as all the community features proposed by Github[1].

### 4.1.2 Maintenance and Version Control

While the objective of this thesis is to implement only the editor aspect of the DSL presented in the last chapter, we thought it wise to take into account the potential future incorporation of the semantics aspect as well. Therefore, it is important to use a technology that facilitates maintenance and version control. We have already discussed how MPS supports version control well enough to be usable in practice in section 2.3.1.

MPS utilises a series of DSL, whose respective domains all correspond to some aspect of DSL engineering. All these DSL depend on the Structure language, with which the developer defines the AST. Any change in the editor aspect will not impact the semantic aspect, and vice versa. Changes in the AST specification however, will affect both, but since all the other aspects follow directly the structure of the AST, changes to be applied to other aspects can be directly derived from the changes made to the AST. Although this does in no way provide immunity to breaking changes, we believe this is effective in limiting their frequency and impact.

### 4.1.3 Composability

The composition of the DSL we implemented is of great importance, as the Strategy language cannot work independently and neither Tactic nor FeatSimple provide meaningful information by themselves, in the context of the software factory methodology. As such, choosing a technology that would facilitate the proper composition of these languages was worth the investment.

The composability aspect is a special case in MPS. MPS is built around the idea of *modular languages* : languages that use a relatively small general-purpose core and can be extended with smaller domain-specific extensions as need be. These smaller extensions, called *language modules*, can be imported and used in a program as needed. Modules have clearly defined dependencies and can reference elements from other modules. Code written in one language module can be embedded into code written in another. This provides MPS with language composition very similar to object-oriented programming, supporting concepts such as inheritance, interfaces, base language generator overriding and independent language embedding [16]. Such a composability vastly facilitates the composition of the languages.

It is notable that having such a composability also affects the two previous arguments. Indeed, the language composition is done so intuitively that no extensive documentation or training on that subject has to be followed, which in turn facilitates maintenance over long periods of time with different developers and teams of varying skills and expertise levels.

[1] https://github.com/

### 4.1.4 Auto-completion

The first and most important end-user-side feature that we identified was auto-completion. This feature is a staple of IDE and has a significant impact on productivity. We've also discussed earlier in section 2.3.2 that projectional edition particularly relied on code completion. Hence, this makes it an essential feature for a projection-based language workbench.

MPS does provide code completion, albeit not in the same way most parser-based editors do. The auto-completion menu does not pop up by itself as the user writes a word, but rather requires the user to use a combination of keys to open the menu. That menu, however, does not require any text to be entered beforehand on which to base the suggestions. In fact, the projectional approach to edition allows the completion menu to provide context-specific suggestions, which can be further reduced by ways of constraints such as scope limitations. This essentially means a user will be using code completion any time it can be used, and as such the key combination is not as negatively impacting as it seems.

### 4.1.5 IDE Generation

Over the years, IDE have become the gold standard of software development tools. There are IDE for every GPL there is, and Jetbrains provides some of the most popular of them, like IntelliJ[2], PyCharm[3] or CLion[4]. All of the Jetbrains IDE are built on the same engine, and MPS is no different. MPS does, however, provide the option of generating your own IDE, built specifically for the chosen language modules. Although having a completely dedicated IDE is already a strong argument in itself, it comes with the added benefit of being made with an engine that most developers are already familiar with, and for which they can import their preferences.

In our use-case, the end-user will be asked to solve internal variability, using DSL tied to software engineering concepts. Hence, we can convene that the end-user will be trained in software engineering and development. This makes providing a dedicated IDE a very strong argument for MPS, since the end user will most likely expect to use such a tool.

## 4.2 Changes to the AST Meta-Models

Several changes had to be applied to the meta-models presented in the previous chapter. These changes stem from the fact that they make use of constructs that aren't present in the MPS Structure language. The use of projectional edition, however, allowed us to limit the effects of these changes in the concrete syntax.

[2] https://www.jetbrains.com/idea/
[3] https://www.jetbrains.com/pycharm/
[4] https://www.jetbrains.com/clion/

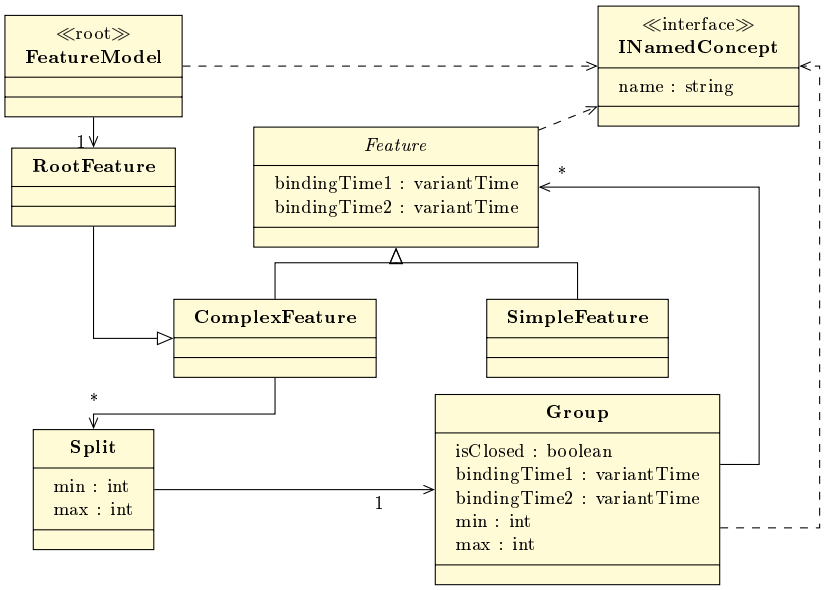


Figure 4.2: The FeatSimple Language Structure

### 4.2.1 Changes in FeatSimple

There are three significant changes performed in FeatSimple. First, all the attributes expressing a name have been removed, and the concepts of which they were an attribute implement the MPS core language *INamedConcept* interface instead. This change has also been carried out on the other two languages. The intent was to allow the MPS editor to display the value given to the name attribute instead of the concept's name in the auto-completion menu.

The second change is the addition of the *Split* concept, which didn't exist in the original meta-model, but was rather expressed as the *split-in* relation. We had to add it because MPS does not support adding attributes to relations. The original multiplicity of the relation has been applied to the *ComplexFeature-to-Split* relation, and the *Split-to-Group* relation has been set to a 1-1 multiplicity, therefore keeping the original meaning.

Finally, the array of variant times has been replaced by two attributes instead. This was changed primarily because MPS does not support arrays of attributes, but also because the parent-child relation could not have expressed that either, due to the relation multiplicities not permitting 0-2. The same reasoning has been applied to the other occurrences of variant time array attributes in the other languages. The updated meta-model, following the rules of the MPS Structure language, can be seen in its entirety on Figure 4.2.

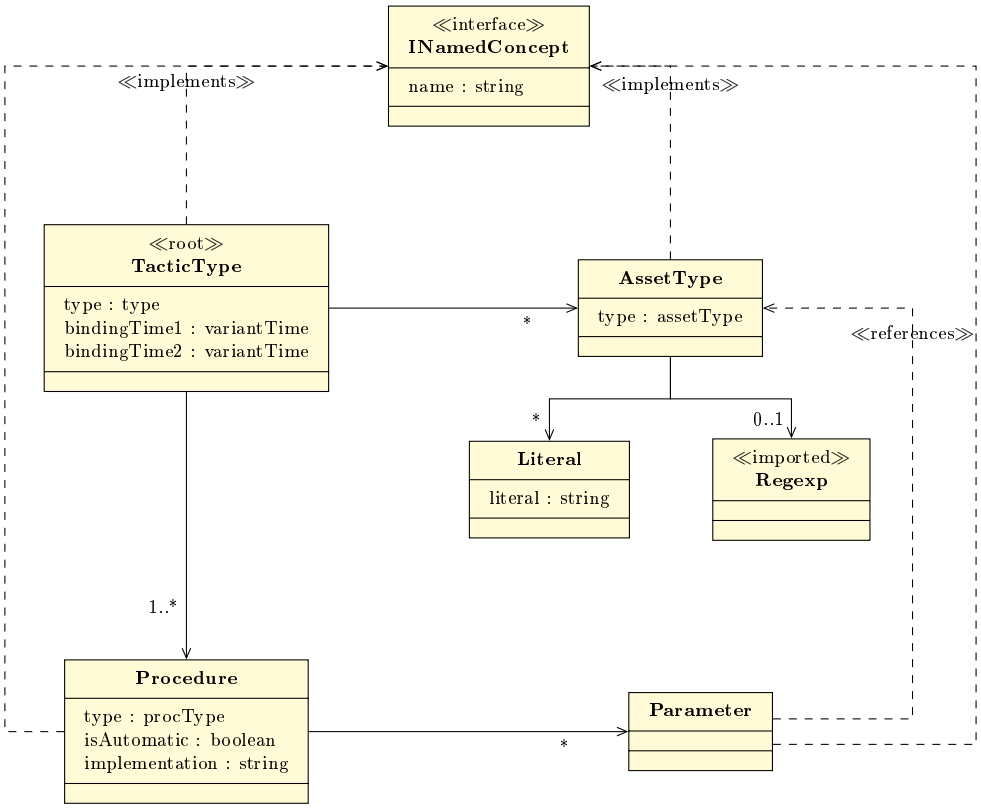


Figure 4.3: The Tactic Language Structure

### 4.2.2 Changes in Tactic

The only changes applied to the Tactic language, apart from the changes made to names and variant times, concern the *AssetType* concept. We've replaced the array of literals with a one-to-many parent-child relation with a new concept, called *Literal*, only comprised of a literal argument. For the sake of brevity, we also limited the possible constraints to regular expressions only, which we added by importing a concept from the MPS *Regex* language and attaching it to the *AssetType* concept with an optional parent-child relation. The relation between the *Parameter* concept and *AssetType* has been set to a reference relation, which is the only way to express an horizontal relation in the AST in the Structure language. The modifications applied to the meta-model are illustrated in Figure 4.3.

### 4.2.3 Changes in Strategy

More substantial changes were carried out in Strategy. In order to simplify the meta-model, we removed the original *Choice* concept and instead placed its attributes and relations directly in *AbstractTactic*. This isn't problematic, since both concepts were related through a one-one correspondence, and as such

made no semantic difference to have two distinct concepts.

As mentioned for the previous languages, array attributes are systematically replaced by a new concept related to the original concept with a one-to-many parent-child relation. This applies in this case of *AbstractTactic*'s valuation attribute, replaced by the new *Valuation* concept.

In several instances, a single reference or relation could apply to either one of two different concepts. That is the case in *AbstractTactic*, which can either reference a FeatSimple feature or group[5], whose valuation can be either delayed or a value, and who can either correspond to a Tactic language TacticType or an AbstractTactic. MPS does not support *xor* relations, but by making use of the MPS modular language architecture, we replaced those with interfaces implemented by two different concepts, corresponding to each option.

These changes are visible on the final MPS Structure meta-model of the Strategy language, on Figure 4.4.

## 4.3 Projection

Although several refactors were applied to the meta-models of the three considered languages, the projections were made to accurately respect the concrete syntaxes presented in the previous chapter. Actually, the concrete syntax was originally designed as a part of a free text, parser-based editing environment. However, the MPS Editor language supports textual notation and its features allowed us to translate that concrete syntax into projection rules without encountering major obstacles.

The projection is made to resemble conventional programming parser-based editors. Hence, special attention has been paid to syntactic highlighting, which in the context of the projectional editor takes the form of text colouration. This has been achieved using a style sheet, a particular feature of the Editor language. Since each cell of the editor either has to declare its style individually, or inherit its style from its parent cell, using a style sheet allowed us to easily apply colouration, and sometimes underlining as well. These styles were allocated to keywords, strings, references, boolean values, enumeration values, integers and other more specific aspects of the editor. This gave us the added benefit of facilitating changes in the color palette since everything is declared in one place.

The MPS editor is built in the Jetbrains IDE engine, which like any other IDE engine uses sub-windows to display the important information. One of those is the object inspector window, which in the case of MPS is used to edit hidden values that aren't displayed on the code window. The interesting part is that the generated editor also supports the use of that window, which the Editor language allows us to utilize as we please. In our use-case, there aren't any hidden parameters to fine tune using the inspector. However, this allowed us to provide a solution to one of the common problems of editing a program with a projectional editor : the programmer has to know and understand the structure of the AST. In our editor, the inspector window has been used to display documentation about the current AST node. On it is featured a brief

[5] This was originally supposed to be in the *Choice* concept.

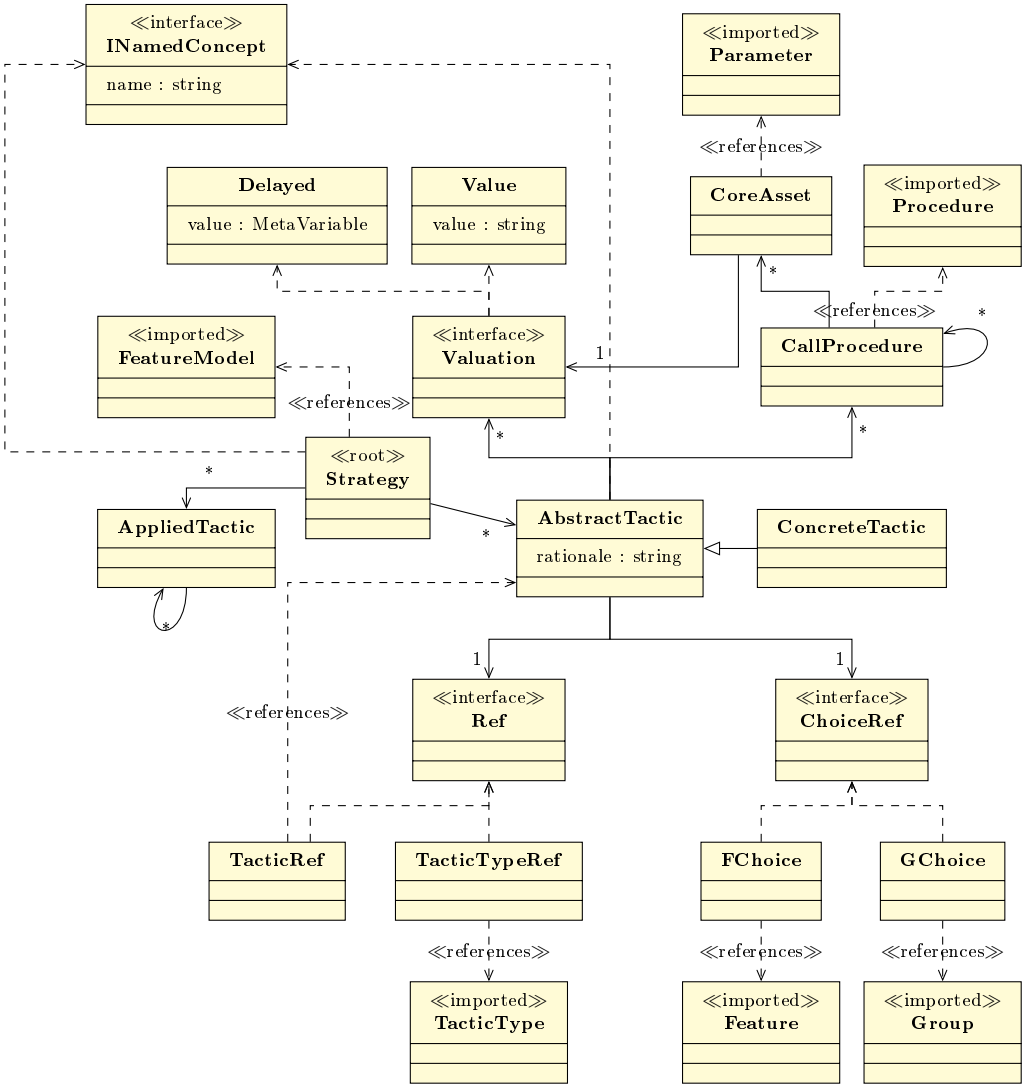


Figure 4.4: The Strategy Language Structure

description of the concept, its attributes and children. This window can be opened at any time without needing to close the editor window. The concept documentation it displays depends on the position of the cursor in the editor window. This was added as a test feature, and is currently only available for the Tactic language. We believe this will help inexperienced users learn our languages faster, while not negatively impacting the more experienced users.

### 4.3.1 The FeatSimple Projection

The following listing shows the editor on an empty FeatSimple program. On this example, we can see the first bit of syntax highlighting we added : the keywords are in orange. By default, any non-editable text in MPS is bold, and empty text cells display either a red or grey message depending on if they're required or can be left empty, respectively. Empty sections with one pair of angle brackets correspond to a single item, while those with two pairs represent an empty list. On this example, we can observe the structure of an invalid FeatSimple AST where both the Feature Model and root Feature have no name, the root feature has no binding time and isn't split in groups of sub-features. If we were to name the model and feature, the program would be valid.

```
Feature Model <no name>
{
  Feature <no name>
    bindingTime = <binding time>
  {
    « ... »
  }
}
```

Listing 4.1: FeatSimple Empty Program

An example of a valid, more complex FeatSimple program can be found in Listing 4.2. We can observe more colors used for highlighting in this example, denoting between integers, booleans and values picked from an enumeration. This example also shows how the projection has been designed with clear indentation in mind. The feature model and root feature have been named[6] and the root feature has been split into two groups. Values have been passed to the *binding time* attributes, which appear in the projection as a single variable, in respect to the original concrete syntax. The features declared in the groups shown in this example are *simple features*, which we can determine by the absence of an empty list of groups in brackets. The latter three, declared in the second group, have been collapsed. The editors support the collapsing of many of the language elements. When collapsed, the projection is reduced so they fit in a single line, which allows the user to navigate the bigger programs more easily.

[6] Note that the MPS *INamedConcept* interface allows any format of name. We chose to use it for simplicity's sake but this can be easily adapted to fit the Java naming format by making our concepts implement another interface, if the semantics implementation were to require it.

```
Feature Model CoffeeMachine
{
  Feature Coffee Machine
    bindingTime = compile <binding time>
  {
    [1–1] Group Power (1–2)
            bindingTime = assembly .. compile
            isClosed = true
          {
            Feature Socket
              bindingTime = assembly <binding time>
            Feature Battery
              bindingTime = <binding time>
          }
    [1–2] Group Brew (2–6)
            bindingTime = runtime.init .. runtime.run
            isClosed = false
          {
            Feature Espresso
            Feature Soup
            Feature Long
          }
  }
}
```

Listing 4.2: FeatSimple Example Program

### 4.3.2 The Tactic Projection

Listing 4.3 shows an empty Tactic program. We can see the empty list of asset types on top. The first empty procedure of the list of procedures is already instantiated, since that list cannot be empty. The example also shows that procedures are by default of the *simple* type, and set as not automatic. The list of procedure parameters, located after the name of the procedure, is empty by default. For the sake of respecting the original syntax, we have wrapped all the string cells with quotation marks. The strings and quotation marks are coloured green, and the grey empty text has been removed since the marks were enough to signify an empty string cell to the user.

```
« ... »

TacticType <no name>
  type = <no type>;
  bindingTime = <binding time>;
{
  procedure simple <no name> « ... »
  {
    automatic = false;
```

```
    implementation = "";
  }
}
```

Listing 4.3: Empty Tactic Program

We can find an example of a complete Tactic program on Listing 4.4. In this example we see two asset types being declared, one with literals and the other with a regular expression constraint. These are further referenced in the procedures arguments. The references are coloured blue and underlined. Being imported from another language, the colouration of the regular expressions is as defined in the language's own editor.

```
declare new AssetType vote boolean <no constraint> { "yay",
                                                     "nay"};
declare new AssetType character string ( . ) « ... »;

TacticType BasicTactic
  type = MetaProgramming;
  bindingTime = link .. assembly;
{
  procedure simple emptyProc « ... »
  {
    automatic = false;
    implementation = "";
  }
  procedure sequence baseProc (c : character)
  {
    automatic = false;
    implementation = "~/procedures/BaseProcedure.txt";
    {
      procedure simple sub1 (result : vote)
      {
        automatic = true;
        implementation = "./subP1.py";
      }
      procedure loop sub2 « ... »
      {
        automatic = true;
        implementation = "";
      }
    }
  }
}
```

Listing 4.4: Example Of A Tactic Program

### 4.3.3 The Strategy Projection

An empty Strategy program can be found in Listing 4.5. We can clearly see the division between Strategy and Recipe. What we notice immediately is also how a strategy references a feature model immediately. Both strategy and recipe clauses can be left empty while keeping the program valid.

```
Strategy <no name> for FeatureModel <no featureModel>
{
  « ... »
}

Recipe {
  « ... »
}
```

Listing 4.5: Empty Strategy Program

Listing 4.6 shows a valid Strategy program. In it, we can see the tactics referencing tactic types, and later procedures from that type and parameters from that procedure. All these references are constrained by logical rules expressed in the Constraint module. For instance, a strategy for a certain feature model can only declare tactics that implement a feature or group from that specific model. We could not, however, find a way to invalidate a program in which a procedure call did not reference *all* the parameters of the procedure, or a way to remove already referenced parameters from the auto-completion menu. In fact, the constraints aspect of the Strategy language could be improved in many aspects, but the editor is still usable in practice.

```
Strategy General for FeatureModel CoffeeMachine
{
  Powerline : implements Socket
  with abstract TacticType BasicTactic
  because ""
  {
    call baseProc ( c = "A" )
  }
  Coffee : implements Brew
  with abstract TacticType BasicTactic
  because "this is a rationale"
  {
    call sub2 ( « ... » )
  }
}

Recipe {
  simple Powerline {
    sequence Coffee {
      « ... »
    }
  }
```

```
}
```

Listing 4.6: Empty Strategy Program

## Conclusion

In this chapter we described the way in which we undertook the implementation of the editor for the languages described in the previous chapter. We first explained our reasoning behind the choice of MPS as our technology of choice. Then we explained and justified the few refactors we applied to the original AST meta-models. Finally, we took a look at the projections of some example AST for each language, and commented those. In the next chapter we will evaluate the editor, as well as its development process.

# Chapter 5

# Evaluation

This chapter is dedicated to show how well our implementation strategy performed. Our analysis tackles, at first, the implementation process. Then, it assesses the end product using some relevant evaluation criteria taken from the literature.

## 5.1 The Implementation Process

As expected from using a language workbench, the suite of tools and DSL offered by MPS reduced greatly the effort required to build the editors, compared to implementing them from scratch. In fact, the editors were implemented in little over a week by one developer. Learning the technology needs some more time, since we have to get familiar with every MPS language, which also required training in the projectional editing of those languages. MPS, however, does provide enough documentation, tutorials and sample projects to effectively mitigate that learning period. Overall, in the entire implementation process, we encountered only one major error, caused by a bug when trying to generate the IDE, which we resolved relatively quickly and without necessitating technical assistance. Still, we did spend some time on the implementation of the reference scopes, due to somewhat poor documentation of the Scope language.

All in all, we consider the implementation process to be relatively easy, especially once reasonably experienced with the MPS environment. We did however note the absence of support for comments in the code, which is likely to be expected by any developer and can also negatively impact maintainability by not allowing the developers to leave documentation and important messages in the code. With that said, we consider it a minor inconvenience, especially since, MPS being an active open-source project, this is still subject to a potential future addition.

## 5.2 Criteria-Based Assessment

In [17], the authors describe a framework for assessing DSL qualitatively. This framework is organised around a list of quality characteristics that concern every aspect of a DSL. Although this framework is mostly angled towards DSL design, we can identify the relevant ones for our use case (i.e. criteria impacted by the implementation) : usability, reliability, maintainability, reusability and integrability. Each can be further divided into more specific sub-characteristics, not all of which necessarily relevant to our evaluation.

By assessing each of these characteristics and ranking them according to their importance, we realise a qualitative assessment of our DSL implementation, stating its strong and weak points. Considering the similarity of our languages in their implementation, we assess these characteristics for all 3 DSL at once.

### 5.2.1 Usability

Usability is defined as the degree to which a DSL can be used by specified users to achieve specified goals. Usability in [17] is mostly approached from a DSL design and documentation point of view, but we recognise attractiveness as a relevant sub-characteristic in the context of the implementation.

Attractiveness is defined as the DSL having symbols that are good-looking. Although still mostly dependant on design, this is obviously impacted by the notations supported by the approach used to implement the DSL. Considering our typical end-user, we set the minimal level to be similar to common code syntax and basic editor highlighting, which is what a programmer would expect.

In that regard, our DSL performs as expected, providing basic syntax highlighting, and utilizing common code edition concepts such as collapsing of elements and indentation. Hence, we mark it as satisfactory. Due to the limited effect of implementation on overall usability, we judge it to be of relatively low importance.

### 5.2.2 Reliability

The reliability of a DSL is the property of that DSL to aid producing reliable programs. This characteristic is observed through two sub-characteristics : model checking and correctness. Model checking reduces error rates, while correctness prevents unexpected relations between language elements. We consider this aspect to be very important, since this is one of the primary functions of program editors, and depends on the implementation alone.

We assume our DSL to be performing well in model checking, thanks to our use of projectional editing, which allows the user to directly edit the AST and therefore not only detects but prevents any inconsistency in the AST. For correctness, MPS let us declare constraints over the definition of the language structure. These reduce the scope of the potential nodes to be referenced in some places. For example, a strategy written for a specific feature model cannot declare a tactic for a feature that is declared on another feature model. Still, as

mentioned earlier in section 4.3.3, these constraints are still incomplete. Thus we consider the correctness to be good.

### 5.2.3 Maintainability

Maintainability, the ease of maintaining a language, is declined into modifiability, low coupling and reusability. Modifiability is the ability of the DSL to be modified to add new functionality without degrading existing functionality. Low coupling is the extent to which the DSL is composed of discrete components such that changing one component has minimal impact on other components. We consider maintainability to be of medium importance, as maintenance is inevitable in the software life cycle.

Low coupling is especially good in MPS thanks to its modular language approach. The MPS modular language approach separates each aspect of the language into distinct modules, whom all share a dependency with the language structure module. A change in the structure might cause errors in the other modules, but a change in any other module will not. Following that logic, we can determine that it is also reasonably easy to add functionality to a MPS language without causing breaking changes.

Reusability is the degree to which language constructs can be used in more than one language, where symbols and other elements of the DSL can be used in more than one DSL. This is clearly the case in MPS, as our DSL shows : we used a concept from the MPS *Regex* language, called *Regexp*, directly into our structure, by simply importing it to the program. This can also be done with any and all concepts of our DSL, but not only concepts. For instance, the editor style sheet declared in the Tactic language was imported and used in all 3 languages. Based on this observation, we conclude that the reusability is well supported in our DSL implementation, as long as it is conducted within MPS. The authors of [17] specified reusability as a distinct characteristic separate from maintainability, while explaining that it is in fact directly linked to it. They justify it not being a sub-characteristic of maintainability because it needs special attention in the assessment process. However, since we did not follow their assessment process, we put it back in the maintainability characteristic.

### 5.2.4 Integrability

Integrability is the degree to which the language can be integrated with other languages. Let's say in our case, a vaster array of languages is developed to automate the entire variability management, external and internal. We could easily imagine the need to integrate some of our DSL concepts into other languages, or even integrate other languages into these ones. Thanks to its modular language approach, MPS makes language composition quite easy, and this applies to integrability too, as long as the other languages are also implemented using MPS. For the integration with the rest of the software factory framework however, this would depend on the model transformations in the semantics aspect, which isn't part of this thesis. Since we did not consider composing this language with more languages in the future, we rank this characteristic as of low importance.

| Characteristics | Importance | Sub-Characteristics | Assessment |
|---|---|---|---|
| Usability | + | Attraction | satisfactory |
| Reliability | +++ | Model Checking | good |
| | | Correctness | good |
| Maintainability | ++ | Modifiability | good |
| | | Low Coupling | very good |
| | | Reusability | good |
| Integrability | + | Integrability | good |

Table 5.1: *Criteria-Based Qualitative Assessment Of DSL Implementation*

### 5.2.5 Interpretation

These assessments seem to indicate that the implementation provides a suitable solution to the problem, but it is necessary to take these results with a grain of salt. These characteristics were picked from a much larger set, which is originally designed for assessing every aspect of a DSL. From the framework presented in [17], we only used some of the characteristics, and did not follow the proposed assessment methodology, preferring a simpler approach. Therefore, this assessment is quite limited, and fails to bring up the topics of user-experience and productivity.

## 5.3 Outlook

The work presented in this thesis is still incomplete. Much still needs to be done in the way of program validation. Some behaviours, like the asset type constraints, were left incomplete too for the sake of simplicity. However, we've already explored these aspects, even if only briefly, in the current state of the implementation. We use this experience to assert that MPS is fully capable of supporting, as well as providing tools for, these aspects. As we've seen, little work is required to build a functional editor. We believe that, with a reasonable amount of additional work put in the editor, we could effectively transform it from a simple to a very capable and efficient editor.

The other, most important missing part of the work is the semantics aspect of the languages. Although this wasn't the objective of this thesis, we would like to discuss on this aspect. It seems to us that, after limited experience with the semantics aspect, the most interesting approach supported by MPS is the model-to-model transformation. We believe that approach to provide the best maintenance and testing capabilities since it allows the full code generation of all languages to be concentrated in a single language module. The MPS *Generator* language supports having several generation approaches. This is interesting in our context, since we could easily imagine the methodology being translated into a chosen output language. We could also imagine choosing between different configurations of the same target language according to the hardware platform that will run the generated code.

The one drawback of model-to-model transformation is that it requires to completely define the target language's structure, projection rules and model-

to-text transformation rules. This can however be mitigated by the specific use we make of that language. For example, we may need to output C++ code, yet only a limited subset of the C++ syntax is required. In that case, we wouldn't need to add support for the entire C++ syntax and code generation, but only the necessary one. There also is the option to import other languages that aren't defined within the project. MPS, being used by Jetbrains internally, offers up-to-date, maintained, Java and XML languages that can be easily imported and used in any project, and feature complete model-to-text translation rules. Other languages have also been implemented by MPS users, and some can be found on Github. These can prove useful, yet unreliable.

Overall, we believe MPS to be capable of providing the required environment to handle the semantic aspects of the software factory methodology.

# Conclusion

In this chapter we presented qualitative evaluations of the implementation process and finished product. Our overview of the implementation process shows that MPS succeeds in reducing both the cost and effort required to implement a DSL, when compared to classical approaches. Our assessment of the editor seems to support our choices of implementation tactics and we consider our implementation to be successful. In the next chapter, we will present the conclusions we came to.

# Chapter 6

# Conclusions

Implementing the aforementioned language editors in MPS allowed us to conduct several observations. The first remark is that the composition of the languages was achieved without requiring extensive understanding of the four composability approaches. In fact, for any language developer familiar with the object-oriented paradigm, the composability offered by the MPS modular languages approach comes quite naturally. This is interesting in the case the DSL suite has to be extended later on, possibly by another developer, and software factories, which are heavily DSL-dependant and designed to be maintained over long periods of time, fit that description. Judging from this and our evaluations presented in the previous chapter, we can deduce that our implementation strategy supports maintainability quite well, while providing a generally reliable editor.

The editor we've implemented gives us no reason to doubt the potential of projectional editors in terms of efficiency. Although it does require some getting used to, as navigating the AST directly is quite a different process than free-text editing, we believe it should pose no problem in our particular use-case. The intended user of our language editors being an engineer of the software factory, and therefore a trained developer, we assume learning the AST of the languages will not be problematic. Besides, the transition period from parser-based to projectional editing would not present a concern in the long run. This is obviously aided by the availability of a custom generated IDE, with VCS support.

At the end, these observations allow us to deduce that language workbenches using the modular language approach such as MPS can be an acceptable option to integrate DSL into software factory frameworks.

# Bibliography


[1] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*, vol. 1. Springer, 2005.

[2] V. Englebert, *Ingénierie d'usines à logiciels — version préliminaire 0.51.* University of Namur, Aug. 2021.

[3] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 16–27, 2003.

[4] M. Voelter, *DSL Engineering.* 2013.

[5] E. Negm, S. Makady, and A. Salah, "Survey on domain specific languages implementation aspects," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 11, pp. 624–633, 2019.

[6] A. Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, pp. 26–36, 01 2000.

[7] F. J. Van der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.

[8] K. Petersen, J. Zaha, and A. Metzger, "Variability-driven selection of services for service compositions," pp. 388–400, 09 2007.

[9] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005.

[10] T. Parr, *Language Implementation Patterns.* The Pragmatic Bookshelf, 2010.

[11] T. Kosar, P. E. Martı´nez López, P. A. Barrientos, and M. Mernik, "A preliminary study on various implementation approaches of domain-specific language," *Information and Software Technology*, vol. 50, no. 5, pp. 390–405, 2008.

[12] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der

Woning, "The state of the art in language workbenches," in *Software Language Engineering* (M. Erwig, R. F. Paige, and E. Van Wyk, eds.), (Cham), pp. 197–217, Springer International Publishing, 2013.

[13] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition untangled," in *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[14] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *Software Language Engineering* (B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, eds.), (Cham), pp. 41–61, Springer International Publishing, 2014.

[15] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), p. 763–774, Association for Computing Machinery, 2016.

[16] M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with mps," *Software Language Engineering, SLE*, vol. 16, no. 3, 2010.

[17] G. Kahraman and S. Bilgen, "A framework for qualitative assessment of domain-specific languages," *Software & Systems Modeling*, vol. 14, no. 4, pp. 1505–1526, 2015.