Test-based Feature Management

for Agile Product Lines

Felix Riegger

Diplomarbeit

Fakultät Informatik
Studiengang Informatik
Hochschule Mannheim

26. Februar 2010

Matrikelnummer:	xxx
Ausgeführt bei:	Prof. Dr. Frank Maurer Agile Software Engineering Group Department of Computer Science University of Calgary

Betreuer:	Prof. Dr. Astrid Schmücker-Schend
Zweitkorrektor:	Prof. Dr. Ivo Wolf

Abstract

Software product lines (SPL) and agile methods are both widespread practices in software development. A SPL is a family of software which is based on common features, but at the same time allows variations to fit the customer needs. In order to create a SPL, domain engineering is required to identify commonalities and variations of the family. This means a lot of upfront work to produce requirement and design artifacts before actual implementation.

In agile methods lightweight artifacts such as executable specifications are used to enable a bottom up development process. In executable acceptance test-driven development these artifacts are recorded as acceptance tests before the actual code is written.

Agile product line engineering tries to integrate SPLs and agile methods to benefit from the advantages of both. One approach is to map features of the product line to acceptance tests instead of using specification and design documents.

Agile Product Liner DSL offers a domain-specific language (DSL) that enables users to create a graphical feature model and link features to acceptance tests. These tests can be executed directly from the feature model.

German Abstract

Software Product Lines (SPL) und agile Methoden sind zwei weit verbreitete Verfahren in der Softwareentwicklung. Eine SPL ist eine Softwarefamilie, welche auf gemeinsamen Features basiert, aber gleichzeitig Anpassungen erlaubt, um den Wünschen des Kunden zu entsprechen. Um eine SPL zu erstellen wird das sogenannte Domain Engineering angewendet, wodurch Gemeinsamkeiten und Veränderbarkeit der Softwarefamilie identifiziert werden. Dazu müssen Anforderungsanalysen und Entwürfe erstellt werden, was einen sehr hohen Aufwand im Vorfeld bedeutet.

In Agilen Methoden wird auf leichtgewichtige Elemente wertgelegt, wie zum Beispiel ausführbare Spezifikationen, um einen Botton-Up-Entwicklungsprozess zu ermöglichen. In Executable Acceptance Test-driven Development werden solche Artefakte in Form von Akzeptanztests geschrieben, bevor der eigentliche Quelltext geschrieben wird.

Agile Product Line Engineering versucht SPLs und agile Methoden zu vereinen, um von den Vorteilen beider Verfahren zu profitieren. Ein Ansatz ist es den den Featuren einer SPL Akzpetanttests zu zuordnen, anstatt Spezifikationen und Entwurfsdokumente zu verwenden.

Agile Product Liner DSL bietet eine domänenspezifische Sprache (DSL), welche es einem Anwender ermöglicht ein graphisches Featuremodell zu erzeugen und Features Akzeptanztests zu zuordnen. Diese Tests können direkt aus APLD ausgeführt werden.

Acknowledgements

I would like to take this opportunity to thank everyone who has helped and supported me.

To Prof. Dr. Schmücker-Schend and Prof. Dr. Maurer, who supervised this thesis and made my two stays in Canada possible and to Prof. Dr. Wolf for being my co-supervisor.

To Uta Fehlinger for her help and endless morale support which got me through.

To Denis Elbert for the great time we spent together in Calgary and the good collaboration while developing the fundament of GreenPepe 2010.

To Theodore Hellmann, Amanda Mickley and Yaser Ghanam for proof-reading my work.

To the members of the ASE Group as well as Heiko Ordelt for their help, guidance and support.

Table of Contents

Table of Figures

List of abbreviations

APL	*Agile Product Line*
APLD	*Agile Product Liner DSL*
APLE	*Agile Product Line Engineering*
ASE Group	*Agile Software Engineering Group*
CIL	*Common Intermediate Language*
CLI	*Common Language Infrastructure*
CLR	*Common Language Runtime*
CTS	*Common Type System*
DSL	*Domain-specific language*
DTE	*Development Tools Extensibility*
EATDD	*Executable acceptance test-driven development*
EMD	*Element Merge Directive*
FMD	*Feature Model DSL*
GP2010	*GreenPepe 2010*
IDE	*Integrated Development Environment*
MEF	*Managed Extensibility Framework*
SPL	*Software Product Line*
SPLE	*Software Product Line Engineering*
STDD	*Story test-driven development*
TDD	*Test-driven development*
UI	*User Interface*
VS	*Visual Studio IDE*
XP	*Extreme Programming*

1 Introduction

Software has become a crucial part of nearly every industry. The importance of software requires increasingly fast creation of high-quality software and thus increased productivity and efficiency. Most software systems are not unique solutions, but part of a family of similar systems that differ in specific aspects. Because of this, the process of developing new software can be made more efficient by reusing parts of similar, pre-existing systems.

Reuse strategies are not a new thing in software development. Traditional approaches to software reuse are small-scale, technology-driven and the results often do not meet business goals. The use of software product lines offers a systematic approach and strategic reuse that yields predictable results. It is an innovative, growing concept in software engineering. SPLs are inspired by product lines in manufacturing where they have been shown to provide measurable benefits [1]. A SPL describes the commonalities and variations of different systems that can be instantiated from it. A single software system that is derived from the product line consists of a selection of configurable reusable artifacts.

An example for a SPL is mobile phones from Nokia. Mobile phones share a common set of features like telephony, text messaging, display output, input keys but at the same time these features vary from phone to phone. Instead of re-implementing the same feature for every phone that supports it, artifacts can be reused. By establishing a product line Nokia was able to increase the production of phones per year from 4 to 25-30 [1].

Software product line engineering (SPLE) is used to create and manage such a product line. The first step in SPLE is domain engineering, in which commonalities and variations of the similar systems are identified and core assets are created that include all of these elements. This requires a lot of upfront analysis and design work, before actual products can be derived from the product line. One common way to describe the commonalities and variations are feature models that present the product line. Based on the feature models, a feature management includes configuration mechanisms.

Another increasingly popular approach to create software is agile software development (ASD). ASD is based on practices referred to as agile methods. Agile methods are lightweight software development processes that emphasize that responding to changing requirements is more important than sticking to a plan. Intense upfront design is avoided since ASD views change as a natural part of software development. Instead, after basic architectural decisions have been made, short development iterations are used to produce running software as fast as possible. Intense customer collaboration makes sure that the evolving software product actually does what the customer wants and needs.

One common practice in agile methods is executable acceptance test-driven development. Acceptance tests are usually created by the customer and describe what the future system needs to do from the customer's perspective. Frameworks like GreenPepper allow the automated execution of these acceptance tests against the system under development.

Agile product line engineering (APLE) is a young field that tries to integrate ASD and SPLE. One big difference between the two practices is the amount of upfront design work. While domain engineering in SPLE requires a huge effort before actual products can be created, agile methods try to minimize upfront design and produce running versions as fast as possible. Although the integration attempt is challenging, it has a huge potential of increasing quality, cuts in cost and reductions in time-to-market [2].

Different approaches can be used to integrate ASD and SPLE. First, agile methods can be used to tailor product instances to a specific situation. This method targets SPL-based organizations that want to become more agile [3; 4]. Second, it is possible to create product lines using agile practices. As mentioned above, in agile methods acceptance tests are commonly used to describe what the system under development has to do, and thus are also design artifacts. The Agile Software Engineering (ASE) Group at the University of Calgary uses these test artifacts to describe and define features of a product line [5; 2; 6]. In order to get further insights and to evaluate and evolve this approach, tool support is needed. The creation of such a tool is the goal of this work.

1.1 Goal of this Work

The goal of this work is to create a test-based feature management tool as extension for Visual Studio 2010. The name of this tool is Agile Product Liner DSL (APLD). The basis of feature management is a feature model, which can be represented either textually or graphically. APLD should allow a user to create, edit and save such a model which means according tool windows, wizards and/or editors are needed. Additionally, APLD should allow a user to create configurations. A configuration means a selected subset of the features from the feature model. A user needs to save and load configurations and needs to validate them. To make the feature management test-based the modeling also has to include tests. Features and tests have to be set into relation with each other. The tests of the model have to be mapped to acceptance tests in the file system. Furthermore, tests should be executable from APLD.

1.2 Structure of this Thesis

The second chapter gives a short overview on which research work and on which tools this work is based. The third chapter describes ASD and SPLE in more detail and how APLE tries to combine them. Moreover, insights into the technical fundamentals are delivered, which comprise the .NET Framework, C#, the integrated development environment (IDE) Visual Studio (VS) and how it can be extended. It also introduces domain-specific languages (DSL) and the DSL Tools. The fourth chapter describes the DSL Tools in more detail, as they are an important basis for this work. In chapter five a concept containing the characteristics of a system which meets all the requirements of the goal of this thesis is developed and chapter six illustrates how APLD is actually implemented. The final chapter presents the conclusion of this work and gives an overview of possible future work.

Class, method and special names are written in *italic*, whereas code fragments are written in the font `Consolas` with a gray background and a black outline surrounding the paragraph.

2 Related Work

2.1 Agile Product Line Engineering

This work is based on research on APLE conducted by Ghanam and Maurer in [5] and [2] as well as Ghanam, Maurer and Park in [6]. In [5] a detailed theoretical description of how variability and traceability can be managed via executable specifications is given. This work looks into how this can be realized as tool support in Visual Studio 2010. Chapter 3.3 describes APLE and the above mentioned approaches in more detail.

APLD is based on two extensions for Visual Studio 2008 and 2010 respectively, Feature Model DSL and GreenPepe 2010. Both tools are shortly introduced in the following two sections.

2.2 Feature Model DSL

Feature Model DSL (FMD) [7] is based on DSL Tools [8], a Microsoft product, and was developed by André Furtado and published under the Microsoft Public License. Its original design was proposed by Gunther Lenz and Christoph Wienands in the book *Practical Software Factories in .NET* [9]. The current version of FMD is an extension for Visual Studio 2008 and offers, among other things, a visual designer, a toolbox, and property windows for creating feature models. It also includes a configuration tool window, which allows creating configurations, that means features are selected. This configuration can be validated and saved. HTML reports can be generated reflecting the whole model and the current configuration. Additionally, custom build actions can be added and triggered from this tool window. They must be provided in separate assemblies and must implement a certain interface. FMD is described in more Detail in 5.3 and DSL Tools in chapter 3.8 and chapter 4.

2.3 GreenPepe 2010

GreenPepe 2010 (GP2010) is an extension for Visual Studio 2010 developed by the ASE Group. The purpose of GP2010 is to execute acceptance tests based on the executable specifications of GreenPepper, a tool developed by Pyxis Technologies [10] (see 3.4.1 Acceptance Testing Frameworks). GP2010 is integrated into the Solution Explorer of Visual Studio and offers test execution from the context menu. The test execution and its results are visualized in a separate tool window, the GreenPepe View. As of the time of writing, GP2010 stores the result HTML files in the same directory in which the test itself is located. To view and edit tests and test results, Visual Studio's built-in HTML editor is used.

In comparison to the Visual Studio extension offered by Pyxis, GP2010 allows the execution of HTML files in the Solution from the VS Solution Explorer. This enables developers to store the acceptance tests together with the source code of the system under test. The extension from GreenPepper needs a connection to a Confluence or XWiki server to access, store and run acceptance tests. GP2010 was developed with the beta versions of the VS 2010 SDK and is based on .NET 4. Consequently, it requires VS 2010 to run.

3 Fundamentals

This chapter gives insights into the theoretical and technical foundation of this work. First, ASD and SPLE are described. This allows for the deeper discussion of APLE which follows. Acceptance tests, the key element of an approach to APLE this work is based on, are also described. Then the .NET Framework, C#, and the Visual Studio IDE are introduced and it is described how Visual Studio can be extended. Finally, domain-specific languages and DSL Tools to create DSLs for Visual Studio are presented.

3.1 Agile Software Development

The term was introduced through the Manifesto for Agile Software Development [11] in 2001 by a group of seventeen people called the Agile Alliance. The members of this group are advocates of lightweight development processes. Agreeing on the importance of being able to respond to changing requirements within the project time frame, they chose the term *agile* to refer to the values and principles comprised by the manifesto [12]. The manifesto encompasses the following values:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more." [11]

It is important to note, that this does not mean that the items on the right have to be abolished or do not have any value, but that the items on the left provide more value. Traditionally organizations have put a huge emphasis on processes and tools, documentation, contract and planning and have neglected the items on the left. An agile process emphasizes values like interaction between individuals and acceptance of changes and at the same time use the items of the right when necessary or if they add indispensible value [13].

The Agile Manifesto introduced twelve principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
- Business people and developers must work together daily throughout the project
- Build projects around motivated individuals
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is the primary measure of progress
- Agile processes promote sustainable development
- Continuous attention to technical excellence and good design enhances agility
- Simplicity – the art of maximizing the amount of work not done – is essential
- The best architectures, requirements, and designs emerge from self-organizing teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

Agile methods are lightweight software development methodologies that adhere to those principles and introduce among other things short development iterations, disciplined project management and quick adaptation to changing requirements. Some specific types of ASD are Extreme Programming (XP) [14], Scrum [15], Crystal Methodologies [12], Adaptive Software Development [16] and Agile Unified Process (AUP). These types use, among others, practices like test-driven development (TDD), executable acceptance test-driven development (EATDD), continuous integration (CI), pair programming and daily scrum meetings.

3.2 Software Product Lines

Clements and Northop state in *Software Product Lines: Practice and Patterns*: "A software product line 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." [17]

Software Product Line Engineering (SPLE) is a paradigm to develop software applications using common platforms and mass customization [18]. 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 [19]. Besides the code, a software platform includes requirements, architecture, documentation and all other artifacts that are indicated by the development process.

Typically SPLE comprises two parts, domain engineering and application engineering. Domain engineering is conducted upfront to create a platform consisting of product line artifacts. The scope of the domain is analyzed and commonalities and variations are identified.

In application engineering, concrete instances are derived from the platform, in which the reuse of platform artifacts saves time and resources.

3.2.1 Feature Modeling for SPL

Feature models are an important kind of requirement artifacts used in SPLs [20]. It is even considered as prerequisite for SPLE [21]. Features describe the characteristics of a system in terms of functionality as well as quality. Feature modeling allows a hierarchical decomposition of features organized in a tree. Features can consist of sub-features and can be mandatory, optional or alternative.

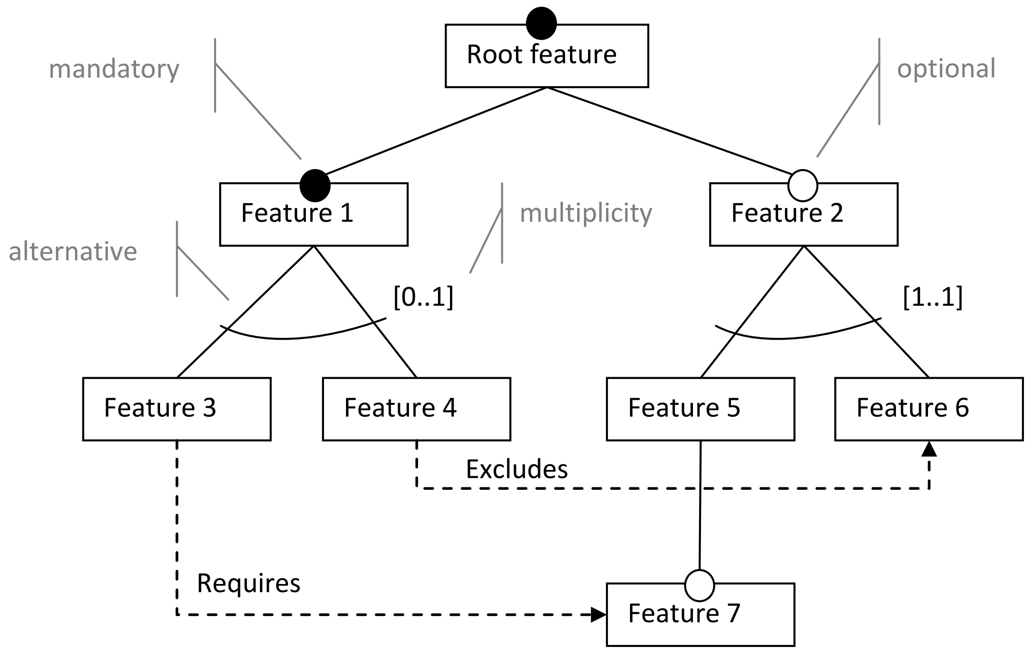


Figure 3-1: A feature model including the elements of a possible notation

Different notations for feature models exist. One of the notations is illustrated in Figure 3-1. It consists of features that are either mandatory or optional. Features can have sub-features that are alternatives, in which the multiplicity describes the allowed composition. The multiplicities set in the example model describe in fact an exclusive or ([1..1]) as well as a normal or ([0..1]. Additionally, cross-cutting constraints are possible. The two most common are *Exclude* and *Requires*. In Figure 3-1 Feature 4 excludes Feature 6, which means Feature 4 and 6 cannot be part of the same product. Feature 3 requires Feature 7, which means, a product including Feature 3 must also contain Feature 7. The notation of the given example is also the notation used in this work.

Lee states "that the feature model can provide a basis of developing, parameterizing, and configuring various reusable assets […]. In other words, the model plays a central role, not only in the development of the reusable assets, but also in the management and configuration of multiple products in a domain." [21] A feature management allows the creation and manipulation of feature models and based on that, the selection of features adhering given constraints in order to create configurations. Based on such a configuration a product can be instantiated or other actions can be triggered, like for example reports.

3.3 Agile Product Line Engineering

For each, ASD and SPLE, there is a lot of literature and research addressing the two topics. When we look at APLE, however, literature and research are rare.

As mentioned before there are two different approaches to integrating ASD and SPLE. The first tries to use agile methods as development model in SPLE, the second emphasizes agile as key player within which SPLE techniques will be used. This particularly targets agile organizations which want to establish a SPL [2]. An approach to the latter sees test artifacts as possible bridge between ASD and SPLE, especially acceptance tests, as first proposed by Ghanam, Park and Maurer in [6].

Ghanam and Maurer also proposed an *Iterative Model for Agile Product Line Engineering* [2]. Instead of the platform requirement and architectural design prerequisites of instantiating product instances, a bottom-up approach is advocated that builds the product line iteratively from existing product instances. In this approach, acceptance tests are seen as the corner stone of the bridge between the two practices.

In 2009 this idea was refined in *Extreme Product Line Engineering: Managing Variability & Traceability via Executable Specifications* [5]. It presents how SPLE and ASD stand in conflict and presents a strategy for overcoming them in an XP environment. This is based on test-driven development that utilizes executable specifications in the form of acceptance tests. Like mentioned in 3.2.1, a very common way to express variability and commonality in software product lines is a feature model. The term *feature* refers to a chunk of functionality that delivers business value [22]. The production of test artifacts is driven by features requested by the customer. It is, however, unclear how features and tests are related to each other. For example, a feature can be tested by several test artifacts and a single test might test parts of several features. Ghanam proposes the use of acceptance tests to model variability in product families.

Each feature of a software product line can be linked to acceptance tests which test it. In XP it is assumed that these tests are up-to-date, and they build a sufficient representation of the functionalities and features the system under test offers. By bringing the features of a software product line together with features that are represented by acceptance tests, the variability of the software family needs only be introduced via these tests. The proposed model also shows an approach to instantiate concrete products from the family using the test artifacts. Based on the assumption that acceptance tests are an accurate and up-to-date reference of features in the system, the following steps are needed to instantiate a product:

1. Select acceptance tests that are linked to the needed features
2. Execute acceptance tests with code coverage
3. Extract code covered by the executed tests
4. Verify and build

In order to introduce new features to the product line, new acceptance tests have to be written. If changes have to be made to the system, acceptance tests must also be adjusted. After acceptance tests have been changed, all variants of the family have to be re-instantiated and tested to verify that nothing has been broken and the change has propagated to all relevant versions.

3.4 Acceptance Tests

In the previous chapter it was proposed that test artifacts, in form of acceptance tests, can be seen as possible bridge between ASD and SPLE. As mentioned in 3.1 EATDD is a common practice in agile methods, mainly in XP. There are many analogous terms for acceptance tests: executable specifications, story tests, functional tests, scenario tests, system tests and many more [23].

Acceptance tests are high-level tests which test if the program is meeting the initial requirements. It is usually performed by the customer. In XP, these tests are created by the customer together with the developer during the design/planning phase. The customer creates the tests from user stories. Acceptance tests may or may not be automated [24].

3.4.1 Acceptance Testing Frameworks

Two popular frameworks for acceptance testing are Fit, which is described in the book *Fit for developing Software: Framework for Integrated Tests* [25] and GreenPepper from Pyxis Technologies [10]. Both offer executable specifications in form of HTML tables or bullet lists.

Pyxis states that executable specifications is an approach to automatically execute human written specifications against the system under test to verify if it is doing what the customer wants. The goal of this approach is to minimize the risk of the developed system not meeting the requirements [26].

GreenPepper defines a syntax for executable specifications and provides a Java and a .NET runner to execute these specifications. The specification has to be written in a HTML table or HTML bullet list. A thin layer of code, called a *fixture*, maps the human written specification to the actual code.

GP2010, which was described in 2.3 and is extended in this work, is based on GreenPepper.

3.5 C#, .NET and Visual Studio IDE

Microsoft’s .NET Framework is a large set of class libraries that can be used with several programming languages (for example F#, Visual Basic or Managed C++). Moreover, the framework provides a common language runtime (CLR). It is responsible for the execution of .NET applications written in a .NET programming language. When an application written in a .NET language gets compiled, it is translated into the common intermediate language (CIL), formerly known as the Microsoft Intermediate Language (MSIL), and assembled into byte code. This byte code gets executed in a virtual machine at runtime and a Just-In-Time compiler translates it into native code which can be executed by the CPU. The benefit of this is that different parts of an application can be developed in different languages, but all will still compile to the CIL and thus are compatible. The Common Type System (CTS) specifies how types are represented in memory and .NET languages have to follow that specification. Not all .NET languages support the whole set of possible types. To ensure compatibility between different programming languages Microsoft created the Common Language Specification (CLS). The CLS consists of a subset of data types and rules. Types defined in the CLS are available in all .NET programming language and include types like Boolean, integer and double. If types are used that are specific for a single .NET language, but are not part of the CLS, the assembly cannot be used by other .NET languages. Together, all the parts described above build up the common language infrastructure (CLI).

C# was developed by Microsoft in 2001 and is one of the languages designed for the CLI. C# 4.0 is the latest version of the language. The specification was finalized in May 2009 and is currently in beta testing and will be released together with the .NET Framework 4.0 and Visual Studio 2010.

Visual Studio is an IDE from Microsoft that offers development for all platforms supported by Microsoft Windows, Windows Mobile, Windows CS, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

3.6 Visual Studio Extensibility

Visual Studio can be extended in several ways [27]. Macros and add-ins allow the customization of the Visual Studio IDE and are based on the Visual Studio Automation Model. The Automation Model can be accessed through the Development Tools Extensibility (DTE) object, which is the highest level object in the automation model hierarchy. Macros and add-ins, however, are limited. To create custom editors or custom project types, the Visual Studio SDK is needed, which allows the creation of VSPackages and Managed Extensibility Framework (MEF) extensions. Many components of Visual Studio, like the code editor, itself are VSPackages and MEF extensions.

The possible ways to extend VS do not exclude each other. An extension can be a VSPackage that includes MEF components and at the same time uses the Automation Model to accomplish certain tasks. Any combination is possible.

An extension developed with the DSL Tools (see 3.8) has to be a VSPackage, because the created extension highly integrates into the Visual Studio environment, including new designers. Using the DSL Tools project template, the necessary project structure for a VSPackage is automatically generated. VSPackages created with DSL Tools for the Visual Studio 2010 SDK are also MEF components, because the new version introduces a model bus that allows DSL extensions via MEF.

APLD is based on the DSL Tools, therefore, it is a VSPackage that exports MEF components. It also uses the DTE objects to access the VS environment.

3.7 Domain-Specific Languages

In contrary to general-purpose languages, a domain-specific language (DSL) is a special-purpose language that is dedicated to a given problem domain and is not intended to solve problems outside this domain. A generic or general-purpose approach provides a solution for many problems, but the solution may be suboptimal or harder to achieve, compared to a solution created using a DSL for a smaller set of problems [28]. The boundaries between the two terms are blurry and one domain-specific language might be more specific than another one [29]. The development of a DSL is useful when a particular problem set can be solved more efficiently and the problem reappears often enough. A DSL may be textual, graphical, or both. Graphical DSLs often include code generation from the graphical model. Because DSLs are very problem-specific they are often more accessible to experts of the given problem domain [30]. A well known DSL for example is Microsoft Excel. Although spreadsheets are inadequate for creating three-dimensional animations or programming real-time systems, they are very powerful in dealing with certain forms of calculations. Furthermore, they are comprehensible for people who are familiar with calculations, but not necessarily with programming languages. The semantic distance between the language and the problem is smaller. Additional examples of DSLs are regular expressions, SQL, HTML and UNIX shell scripts, GraphViz and expectations in JMock [31].

3.8 Microsoft DSL Tools for Visual Studio

Microsoft's DSL Tools extend the Visual Studio SDK. They can be used to implement graphical DSLs and deploy them as extensions to the Visual Studio IDE. DSL Tools intend to reduce the cost of designing a new DSL. Diagrammatic languages can be created quickly and tools for generating artifacts from them can be implemented.

Graphical DSLs have important aspects, the most important being notation, domain model, generation, serialization and tool integration [32]. The DSL Tools address these aspects. They offer a diagrammatic language to create new DSLs and are integrated into VS.

With the DSL designer, domain models and notations are defined. Created diagrams are serialized to an XML file. All necessary files are generated from the model. The generated artifacts consist of two parts. First, the code that describes the new DSL itself, which includes notation, domain models and serialization. Second, a package for Visual Studio which integrates the DSL into the environment, including tool windows, DSL designer, and toolbox items. Chapter 4 explains DSL Tools in more detail.

3.9 Summary

This chapter described the theoretical fundamentals, which include ASD, SPLE and APLE, as well as technical fundamentals, which include the .NET Framework, C#, and Visual Studio and its extension, this work is based on. Since this work also involves the extension of an existing tool, which offers a DSL for feature models DSLs in general and Microsoft's DSL Tools were explained. The DSL Tools are a complex and powerful framework to create DSLs for Visual Studio. Their key elements are explained in more detail in the next chapter.

4 DSL Tools in Detail

This chapter explains the fundamentals of the DSL Tools. Key elements of DSLs created with the DSL Tools are clarified and an in-depth look into the structure of a DSL Tools project is delivered. It also shows how a DSL can be customized to fit a given situation. Many of the aspects explained in this chapter can be found in greater detail in the book *Domain-Specific Development with Visual Studio DSL Tools* [32].

4.1 Terminology

The DSL Tools are used to create a DSL as well as code to integrate the DSL into Visual Studio as VSPackage. The DSL and the integration code generated is C# code. Both, C# and the DSL Tools use classes and properties, so to distinguish between them, classes and properties related to the DSL Tools are referred to as *domain classes* and *domain properties*, whereas *classes* and *properties* refer to classes and properties in C#.

4.2 Elements of a DSL

In the DSL Tools, a DSL consists of several components.

- **Domain model**
  The domain model is the core of every DSL. It describes the concepts, properties, and relationships of the language.
- **Presentation layer (including graphical notation, explorer and properties window)**
  This component defines how the model gets presented in the UI of the designer. There are three types of windows: the designer; the model explorer; and property windows. The graphical notation visible in the designer window consists of a diagram which acts as a container for shape and connector maps.

- **Creation, deletion and update behavior**
  Newly created elements have to be integrated into the model and it has to be defined what happens, when an element is deleted, and how changes are propagated.
- **Validation**
  The DSL Tools offer hard and soft constraints as well as rules to allow validation of the created model. Hard constraints restrain the user from making changes to the model that would violate a constraint, whereas soft constraints can be violated by the user at some points in time but not in others. Rules allow the introduction of certain behavior depending on model changes. They can be used to restrict certain changes or to propagate other changes through the model.
- **Serialization**
  When a DSL is defined, a domain-specific serializer is automatically generated, which will save and load models in a XML format. There are several customization options available to modify that format.

4.2.1 Domain Model

The core element of a DSL designed with DSL Tools is the domain model. A domain model consists of domain classes and domain relationships.

A domain class is a node in the domain model and represents an entity of the domain. There is one domain class which is the root of the model: the root domain class.

Domain relationships connect domain classes and describe their relation. Each domain relationship has two ends, a source and a target, thus they are directed. Both ends are called a domain role and the connected class on that end is called the role player. A domain role has a property name and a multiplicity. The multiplicity defines how many links can have the model element as role player.

There are four possible values for multiplicity:

Multiplicity	**Description**
1 / One	Every model element of this class has to play this role exactly once.
0..1 / ZeroOne	A model element of this class may play this role no more than once.
0..* / ZeroMany	A model element of this class may play this role any number of times.
1..* / OneMany	Every model element of this class has to play this role at least once.

There are two kinds of relationships, embeddings, and references. An embedded relationship has some constraints. The multiplicity of its target role has to be either *One* or *ZeroOne*. If more than one embedding relationship targets a domain class, the multiplicity of each of them has to be *ZeroOne*, as only one model element of that class can be embedded. In a complete domain model every domain class must be target of at least one embedding relationship (expect for the root domain class, which is not target of any relationship, but only source). This ensures that the domain model can be described as a tree. The model can easily be serialized into a XML structure. It also implies delete propagation, like in a tree: when an embedded parent model element is deleted its children are deleted as well.

A reference does not have these constraints. Its domain roles can have any multiplicity. If a role player is deleted, only the link is deleted by default, but not the referenced model element.

Both, domain classes and domain relationships, know the concept of inheritance.

New domain classes and domain relationships are introduced via the DSL designer. They appear in the classes and relationships section of the DSL definition diagram.

4.2.2 Presentation of the Domain Model

The domain model is an abstract description of the model that describes the DSL. In order to work with the domain model and to create a model based on this domain model, it has to be visualized. The DSL Tools offer some classes which are used to present the elements of the domain model. The DSL Designer implements a graphical representation of the model, whereas the DSL Explorer tool window offers a textual representation in form of a tree, which implies that reference relationships cannot be visualized (see 4.2.1). The domain property window offers access to the domain properties of selected domain classes and domain relationships.

The DSL Designer is the main component and is realized as editor in Visual Studio. Usually this editor is a graphical designer holding a diagram. The root domain class is mapped to the diagram. Every other element of the model can have a graphical representation in the form of shapes and connectors, where shapes map to domain classes and connectors to domain relationships. A shape can be a geometry shape, a compartment shape, an image shape, a port or a swimlane. Shapes and connectors appear in the diagram elements section of the DSL definition diagram. In the following section, the shapes and connectors are described in more detail.

Geometry Shapes

A geometry shape is a shape that is based on a circle, an ellipse, a rectangle or a rounded rectangle. It can have any kind of decorator. Decorators are described later in this section.

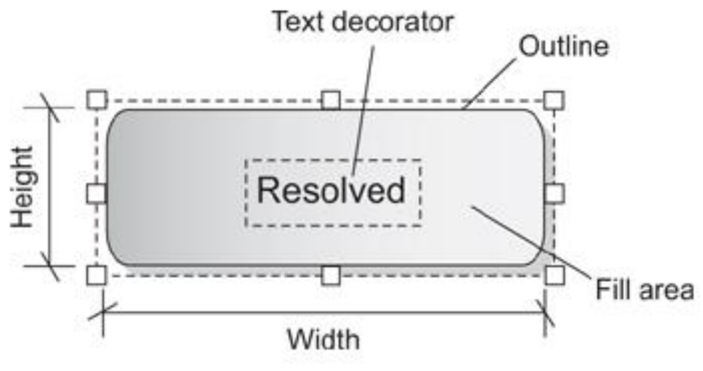


Figure 4-1: A geometry shape [32]

Image Shapes

An image shape allows any image to be used as shape and the shape has no outline. It typically has a text decorator positioned outside of the image and thus the shape itself, but it can also have any type of decorator. For instance the shape to describe an alternative in FMD is an image shape and can be seen in Figure 4-2 on the next page.

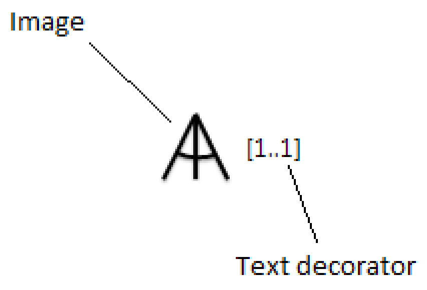


Figure 4-2: The Alternative shape from FMD is an image shape

Compartment Shapes

A compartment shape is a geometry shape that has compartments. Each compartment can have a list of elements. The compartment shape is restricted to be a rectangle or a rounded rectangle. The collapse/expand decorator allows it to hide the compartments.

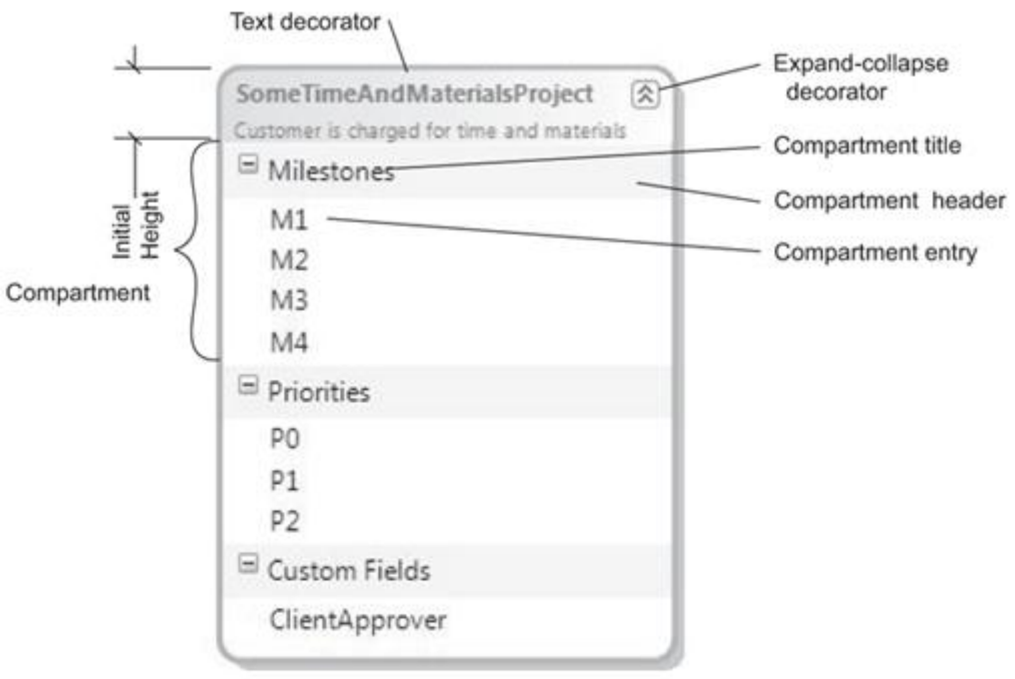


Figure 4-3: A compartment shape [32]

Ports

Ports are shapes that are used on the outline of a parent shape and usually act as connection points. They can only be moved on the border of the parent shape. Except for that peculiarity, a port shape is in fact a geometry shape and supports various kinds of decorators. Figure 4-4 illustrates a port.

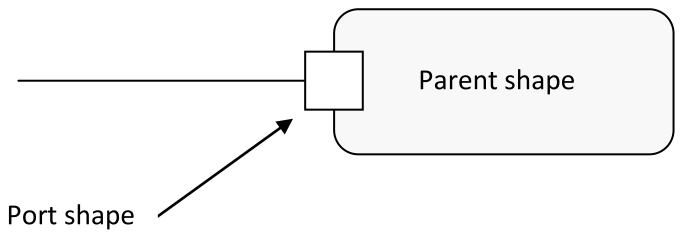


Figure 4-4: A port shape on a parent shape with a link

Swimlanes

The last shape is a swimlane. They are used to partition the diagram into rows or columns. The DSL definition diagram, for example, is separated into two columns, one for classes and relationships, and one for diagram elements.

Connectors

While shapes describe the appearance of nodes in the diagram and are mapped to domain classes, connectors describe the appearance of links, and thus domain relationships. A connector, like domain relationships, is directed and has a source and a target end. Source end, target end and dash style can all be defined. A connector takes all kinds of decorators, which will appear next to the dash. A connector can have different routing styles such as straight or rectilinear. Figure 4-5 shows a connector with a rectilinear routing style and an arrow as target end style.

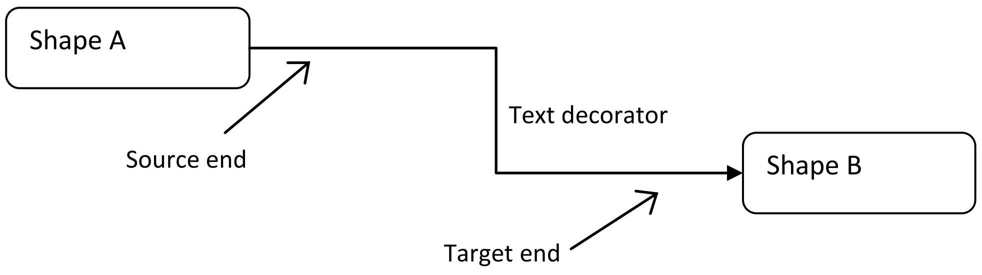


Figure 4-5: A connector with rectilinear routing style and text decorator

Decorators

Decorators are used to add further information to connectors and shapes. There are three decorators available. Text decorators, image decorators and an expand/collapse decorator. The lattermost has a predefined icon showing two small arrows. Without customization, it only has an implemented functionality when it has been added to a compartment shape, where it is able to expand or collapse the compartments. Text decorators are used to put text on or next to shapes respectively, depending on the shape. Image decorators do the same for images.

Connector and Shape Maps

The DSL Designer needs to know which shape or connector has to be used to visualize a certain model element in the diagram. Shape and connector maps are used to provide this information. In a shape map, the shape and the domain class that have to be mapped are defined. Additionally, the path to the parent element must be defined to specify, which element is the logical parent of a mapped domain class. This determines which shape the parent shape of the shape mapped will be. With a port, for example, the parent shape would be the shape to which outline the port is attached to. This relation is described through the underlying domain model, and the path in the model determines the path in the graphical notation.

4.2.3 Domain Properties

All model elements – that means domain classes, domain relationships, connectors and shapes – can have so called domain properties. Domain properties describe states, which are similar to properties in C#. They can be of any CLR value type and are one of three possible kinds: *Normal*, *Calculated*, or *CustomStorage*. *Normal* domain properties follow the standard behavior in the DSL Tools, which means that when a value is set it is automatically serialized to the XML file that keeps the model and diagram information. *Calculated* domain properties only have a getter, which has to be implemented manually. Usually the value returned by this getter depends on other domain properties, but it can be based on any kind of algorithm. The Important thing is that the value is not set by the user, but is calculated at runtime. *CustomStorage* means that the getting and setting of values has to be implemented manually and thus can be completely customized.

4.2.4 The In-Memory Store

When the DSL is deployed as a tool into Visual Studio the central element will be the DSL editor. The editor consists of two parts: the *DocData* class and the *DocView* class. *DocData* holds the model, whereas *DocView* represents the editor window and visualizes the model. As a result, model and view are separated. *DocData* is responsible for loading and saving models and has an instance of the in-memory store. When the code describing the DSL is generated, a corresponding C# class is created for each element of the domain model and their presentation elements. Each of these classes is a direct or indirect sub-class of the *ModelElement* class. Domain classes are a direct sub-class of *ModelElement*, while domain relationships are a sub-class of *ElementLink*, which itself is a sub-class of *ModelElement*. Shapes and connectors are indirect sub-classes of *PresentationElement* which itself is also a sub-class of *ModelElement*. The in-memory store knows all these classes and manages all instances of them. This means that presentation elements are part of the model and only their visualization is handled by the view. The store is responsible for the creation, updating and deletion of any model element. Any of these have to happen in a transaction. The store therefore has a transaction manager which also provides undo and redo of transactions. Every model element has a reference to the store it belongs to. All manipulations can also be achieved programmatically by using the store's API.

4.2.5 Creation, Deletion and Updating

When a new element is created and added to the store, it must be set into relation with the other elements. As mentioned in 4.2.1, every element needs to be connected to a parent through an embedded relationship and there must be a path back to the root element.

Elements can be created either programmatically or by the user. To enable the user to add elements to the model and the diagram, tools are added to the Toolbox of Visual Studio. There are two kinds of tools, namely element tools, which add elements and connection tools to add links between elements. Tools are created through the DSL Explorer. Element tools are linked to the domain class they create, while connection tools are linked to a connection builder.

The element merge directive (EMD) decides how a newly created element has to be embedded into the model. The EMD is defined for the parent element. That means, if the user uses an element tool and drags it over the diagram in order to create a new element, the parent would be the class which is mapped to the diagram. The EMD of this class decides what links have to be built between the new element and the diagram class. By default an embedded relationship is created, indicating that the parent element owns the new element, but this behavior can be changed. EMDs are also responsible, if the parent has to be changed, for example when an element is moved from one swimlane to another.

When a connection tool is used to create a new connection, it uses a connection builder. The connection builder knows which elements may be connected and what links have to be created in order to connect these elements. To describe this, link directives are used or, alternatively, the connection builder can be completely customized.

Element and connection tools handle only the creation of domain classes and relationships. The connectors and shapes are created automatically by the connector and shape mappings.

To describe, what happens when an element is deleted, delete propagation rules are used. The default rules are:

- If an embedding link or an element are deleted, the complete embedded sub-tree is deleted as well
- Deleting an embedding child will not delete the parent
- Deleting of reference links will keep both role players

This behavior can be changed or additional rules can be added. Delete propagation can be set for domain classes and specifies to which links the deletion is propagated. Delete behavior is set for the domain model only. It is only intended to reflect the model, so it does not make sense for the presentation. Further customization is possible by overriding parts of the *DeleteClosure* class.

4.2.6 Serialization

When a DSL is defined, a serializer is generated automatically. It serializes the domain model and the diagram to XML and introduces a domain-specific schema. The serialization can be customized in several ways, but since the default behavior is not changed in the scope of this work, serialization will not be explained in further detail.

4.2.7 Validation

There are two categories of constraints: hard and soft. Hard constraints cannot be violated, as the user cannot set such values. Soft constraints can be violated at some points in time, but not in others.

The DSL Tools offer validation methods to test soft constraints. The validation methods can be applied to any class that is based on *ModelElement*. In case a class should participate in validation, a partial class has to be created and attributed with the *ValidationState* attribute setting it to *ValidationState.Enabled*. Every validation method in that class has to be attributed with the *ValidationMethod* attribute which takes a *ValidationCategory* as parameter. *ValidationCategory* describes when the validation should happen. Standard values are *Load*, *Menu*, *Open* and *Save*, but custom categories can be introduced. *Load*, *Open* and *Save* happen whenever the corresponding operation is performed on the diagram, whereas *Menu* happens when the context menu is opened. Each category has to be activated in the DSL Explorer in the validation properties of the editor. Every validation method has a *ValidationContext* as parameter. The context allows logging of errors, warnings and messages, which are added to the error list of Visual Studio.

Hard constraints can be introduced by customizing the DSL, for example introducing a special connection builder that allows only the connection of certain elements. Some are already enforced as default behavior of a DSL, which are the maximum multiplicity of roles, types of role players, and types of property values.

4.3 The Structure of a DSL Tools Project

A DSL Tools solution in Visual Studio contains two projects, *Dsl* and *DslPackage*. The *Dsl* project provides code defining the DSL, its behavior in the designer, how it is serialized, and how transformations work. It therefore contains a serializer/de-serializer for reading and writing instances of the DSL from or to files, class definitions for processing the DSL and its diagrams in an application, a directive processor enabling the user to write text templates that will process the DSL, and essential components of the designer that edits the DSL in Visual Studio. The *DslPackage* project contains code that couples the DSL with the Visual Studio environment, which means it contains everything that is needed to extend the Visual Studio environment with a VS Package providing the DSL. This includes document handling code that recognizes the DSL's file extension and opens the appropriate designer, menu commands associated with the DSL's designer and item template files from which new instances of the DSL can be created. In both projects a folder named *GeneratedCode* can be found, in which most of the files reside. These files are generated from the *DslDefinition.dsl* in the *Dsl* project. As mentioned before the DSL Tools themselves are a domain-specific language and offer a graphical designer to create new DSLs. This information is stored in the mentioned *DslDefinition.dsl* file. The *GeneratedCode* folders contain *.tt* files (text template) and the generated files. These *.tt* files contain *include* directives which reference the DSL definition file as well as a template for the file and a processor which are shipped with the DSL Tools. When the templates are transformed, the processor generates an output file from the information in the definition file and the referenced template. The following snippet shows the content of the *Shapes.tt* which is responsible for the generation of the class *Shapes.cs*.

```
<#@ Dsl processor="DslDirectiveProcessor"
       requires="fileName='..\DslDefinition.dsl'" #>
<#@ include file="Dsl\Shapes.tt" #>
```

The architecture of the DSL Tools has three layers. First common features of all DSLs that can be created with the DSL Tools are contained in the Compiled Framework layer. Next the generated code, which puts everything defined in the DSL definition file into code. Finally, a layer of hand crafted code, which allows further customization of the generated code. See Figure 4-6.

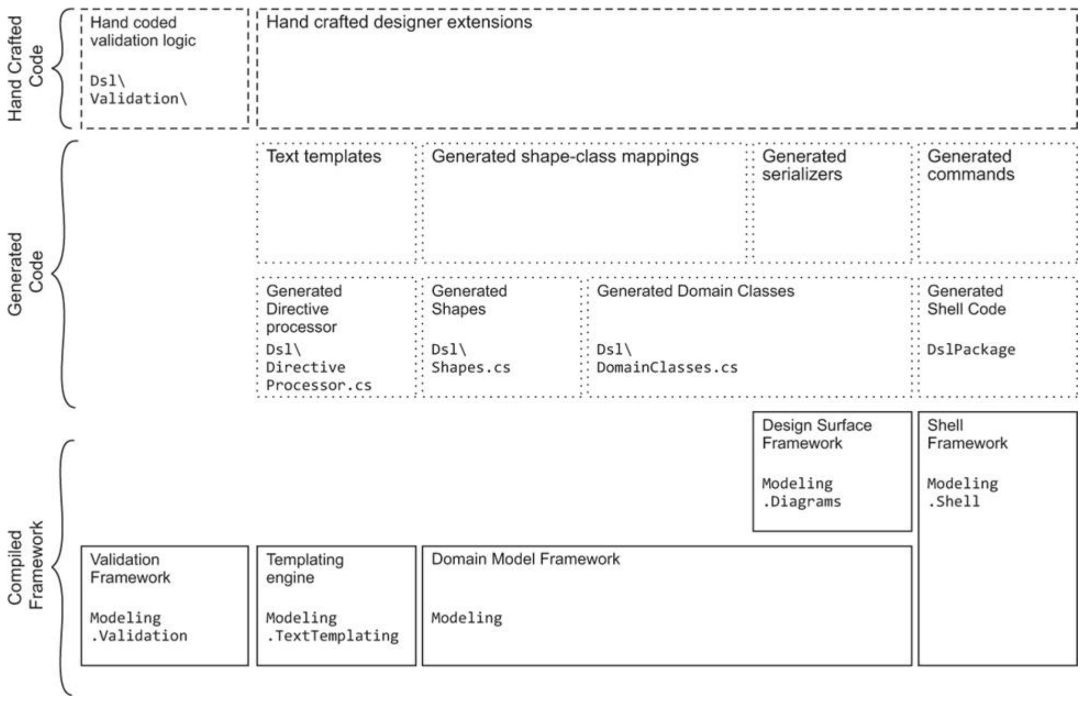


Figure 4-6: Architecture of the DSL Tools [32]

4.4 Customization

It makes no sense to edit the generated class files to introduce further customization. After changes have been made to the DSL definition file, the text template files have to be transformed to propagate the changes. The generated files would be overwritten, so the customizations would be lost. .NET languages offer the concept of partial classes, which means a class can be distributed over multiple files by using the *partial* keyword. The DSL Tools use this concept to offer customization. All the generated files are partial, so methods can be added and methods from the framework can be overridden. Additionally, the DSL Tools support the design pattern known as the *generation gap* [33]. It is a common pattern that allows simple integration of generated and hand-crafted code and is based on double-derived classes.

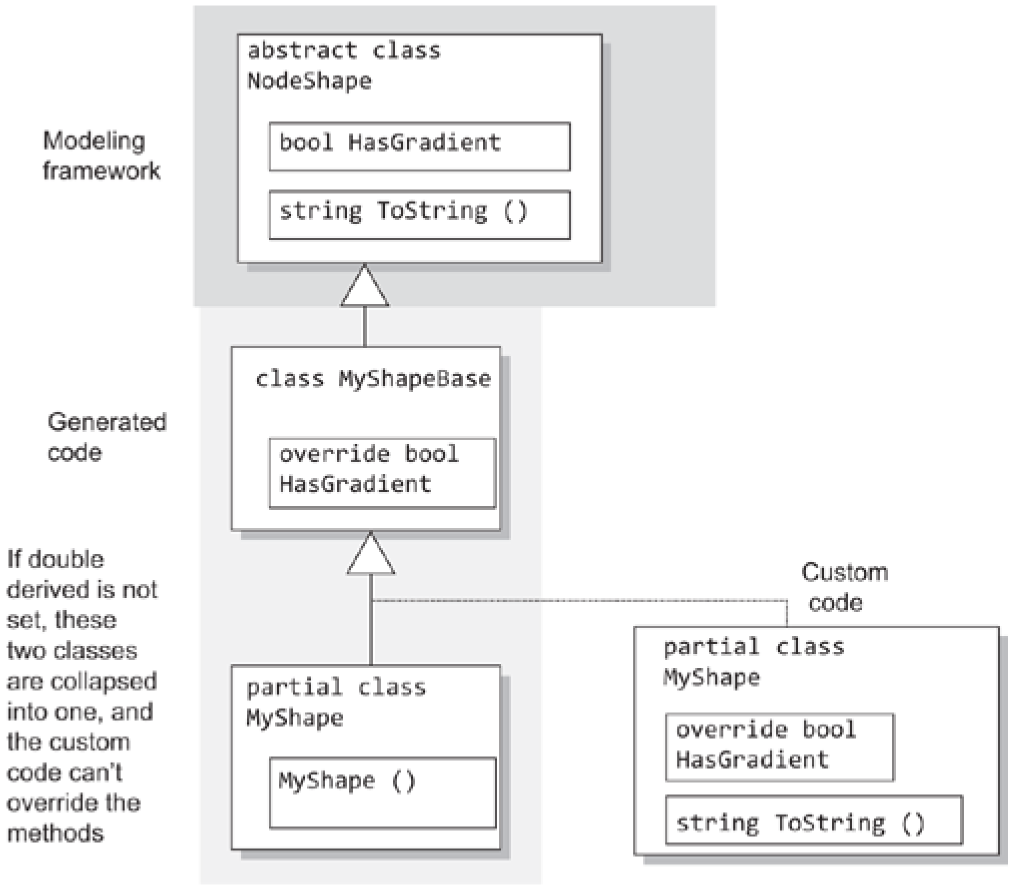


Figure 4-7: Use of the generation gap pattern to introduce customizations [32]

The DSL Tools offer double derived classes to use this pattern. It is used by APLD to introduce customization. It is recommended that hand-crafted classes are put in the folder *CustomCode*.

4.5 Summary

This chapter explained the basics of the DSL Tools in more detail. The key elements of a DSL defined using the DSL Tools were identified and described. Additionally, the structure of a DSL Tools project was discussed as well as how customization can be introduced.

5 Test-based Feature Management

This chapter looks into how the goal of this work can be achieved by meeting the given prerequisites and developing a corresponding concept.

5.1 Prerequisites

As of the time of writing current projects of the ASE Group at the University of Calgary are based on Java or C#. More and more projects are related to digital tables, where the .NET framework and its graphical subsystem Windows Presentation Foundation (WPF), have shown to have advantages over Java technology. C# is the .NET language of choice and Visual Studio the IDE used by the ASE's developers. As more and more projects are based on C#, the group is especially interested in projects that intend to offer tool support for agile development practices in the form of extensions which are available for Visual Studio.

In the very near future Microsoft will release version 4.0 of its .Net framework as well as Visual Studio 2010 which will support the new version to the general public. Beta 1 of both products was released on May 18th 2009, Beta 2 on October 23rd 2009 and the release candidate on Feb 10th 2010. VS2010 introduces a new method of deployment: VSIX container files.

This work introduces a test-based feature management and links feature modeling to acceptance tests. GreenPepper is the acceptance testing framework mainly used by the ASE Group. GreenPepe 2010 is an extension for the current version of Visual Studio 2010 that allows recognition and execution of GreenPepper based acceptance tests in the Visual Studio Environment. It is reasonable to extend GreenPepe 2010 in such a way that it offers its acceptance test handling capabilities to other extensions in the Visual Studio environment.

It was required that new extensions work in Visual Studio 2010. There are two ways to achieve this: develop the new extension with Visual Studio 2008 and to make sure it runs in Visual Studio 2010 afterwards, or develop in Visual Studio 2010 from the beginning. While developing extensions the developer can debug them in an experimental instance of Visual Studio. This instance is the same version as the version in which the extension is developed. For that reason, if the extension has to run in VS2010, it makes sense to develop with VS2010 in order to use this debugging functionality. Also, new features like deployment with VSIX files or extensibility via MEF are only available in the new version. The new extension has to communicate with GreenPepe 2010, which was developed with VS2010. It is easier to implement that inter-extension communication if both are developed in the same version of VS. It was decided to develop the extension with the beta versions and to take the risk of stability problems, because of all these aspects. It also implies a migration to the most current testing version, if released.

The technical prerequisites summarized:

- C# as implementation language
- The resulting tool has to be a Visual Studio 2010 Plug-in
- Development with the Beta versions of VS2010

5.2 Necessary Functionality

In order to realize a test-based feature management, the following key characteristics can be identified as necessary functionality. A feature model is needed that can describe a product line as system with components, features, sub-features, and constraints for features and sub-features. Additionally, the feature model has to include tests and relations between features/sub-features and those tests. A user interface is needed that allows the creation and manipulation of a feature model and the creation of configurations. The feature model has to be persisted. The tests of the model have to be mapped to real acceptance tests in the project.

To reach these goals in the given timeframe the following solutions are a possible approach. First, the persistence should be realized with XML. Serialization to XML has some benefits. The persisted model exists in a human-readable form. Moreover, it is wide spread and there are several libraries that offer XML serialization. Second, a dialog based user interface in the form of wizards to create and edit the feature model and create configurations is needed. Third, an object model is required that represents the feature model in memory. A first draft for this model can be seen in Figure 5-1. There is one system which can contain components. Each component can consist of several features. These may have sub-features. A sub-feature can be default or optional and sub-features can exclude each other mutually. This information is stored in a constraint class. Each sub-feature can be related to an arbitrary number of tests and each test can be related to any number of sub-features.

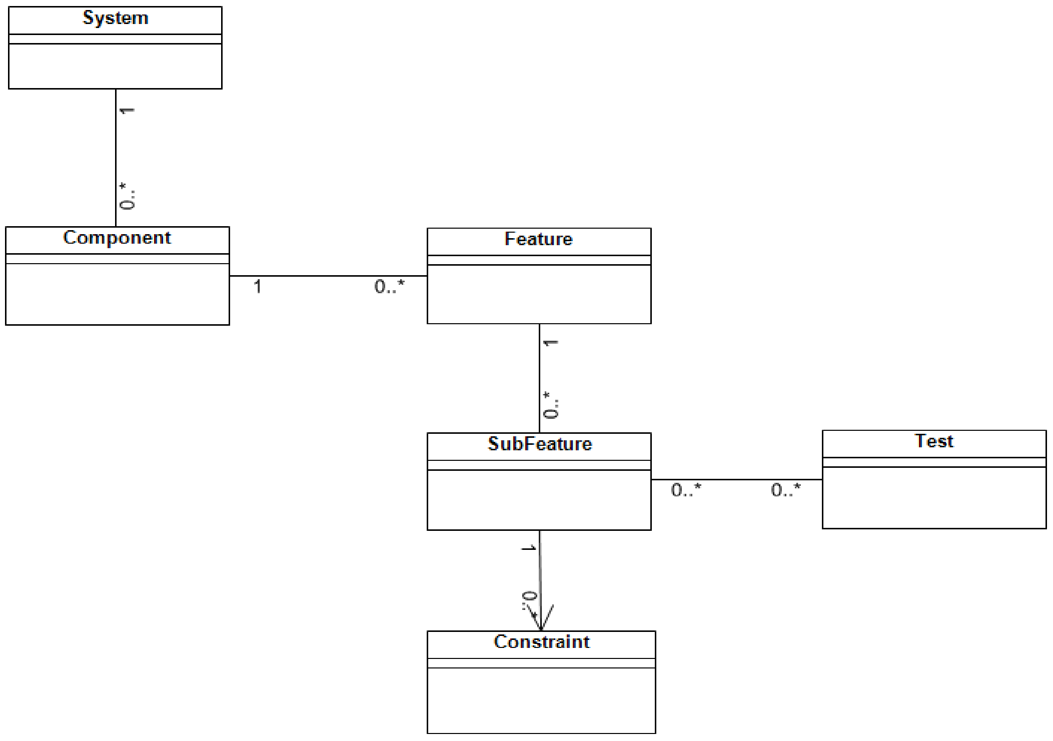


Figure 5-1: A first draft of an object model for the test-based feature model

Fourth, mapping the model elements that represent acceptance tests to real tests in the project requires identification of and access to test files in the solution. Additionally, the execution of test files and showing the result can give feedback, if the features based on those tests are ready to be built. With GreenPepe 2010 the group already has a tool at hand that implements that functionality. It is therefore reasonable to extend it in that way, so it provides access to acceptance tests in the solution and the execution of those to other extensions in the Visual Studio Environment.

5.3 Investigating existing Tools

Feature modeling is not a new thing. Many tools exist that support the creation, editing and/or analysis of feature models. What is new, though, is the approach to combine feature models and acceptance tests. To implement this approach it should be considered to extend an existing feature modeling tool instead of building everything from scratch. There are a handful of criteria a potential tool has to meet. First, the extension might imply the manipulation of the original source code, therefore the tool has to be published under an open source license and the source code must be available. Second, as mentioned before, it is required by the ASE Group that new projects that offer tool support have to be implemented with C#. Hence, if another tool is supposed to be extended it needs to be written in C# as well. Third, while speaking of tool support, tools that are intended to support developers in their work have to integrate well into the workflow. The IDE used by the development team is the central element in that workflow, therefore, it was also required that the new tool has to integrate with Visual Studio as the IDE of choice. After some research it was clear that there are many open source projects that realize feature modeling in all variations of complexity, but most of them are written in Java and are plug-ins for the Java IDE Eclipse, which eliminates them from consideration.

Table 5-1 shows an overview of some feature modeling tools. Regarding C# and Visual Studio, there are just two tools, which come into consideration: Feature Model Tool and Feature Model DSL.

Table 5-1: Overview of existing feature modeling tools

Tool name	Open source	Language	IDE	Homepage
EMF Eclipse	✓	Java	Eclipse	[34]
Feature IDE	✓	Java	Eclipse	[35]
Feature Model DSL	✓	C#	VS 2008	[7]
Feature Model Plugin	✓	Java	Eclipse	[36]
Feature Model Tool	?	C#	VS 2008	[37]
Hydra	✓	Java	Eclipse	[38]
Pure::variants	✗	Java	Eclipse	[39]
xFeature	✓	Java	Eclipse	[40]

Both, Feature Model Tool and Feature Model DSL, were created with the DSL Tools from Microsoft and the Visual Studio SDK 2008. They work with Visual Studio 2008. The homepage of Feature Model Tool does not state a license and the source code is not available. Therefore only Feature Model DSL is left as potential candidate.

Feature Model DSL was published under the Microsoft Public License, which is an open source license. It offers feature modeling integrated in the Visual Studio environment including visual designer to create and modify models. Feature models are serialized and persisted to XML. It also offers a configuration tool window that allows the creation of configurations based on the feature model and offers the implementation and launching of custom actions based on the configuration.

Since the source code is available, the tool can be modified and extended. Some functionality identified as necessary in chapter 5.2 is already provided by FMD. Most importantly, it includes a complete domain-specific language for feature modeling based on the DSL Tools from Microsoft and thus provides a graphical designer and notation to create and modify a feature model. Figure 5-2 shows the underlying domain model of FMD.

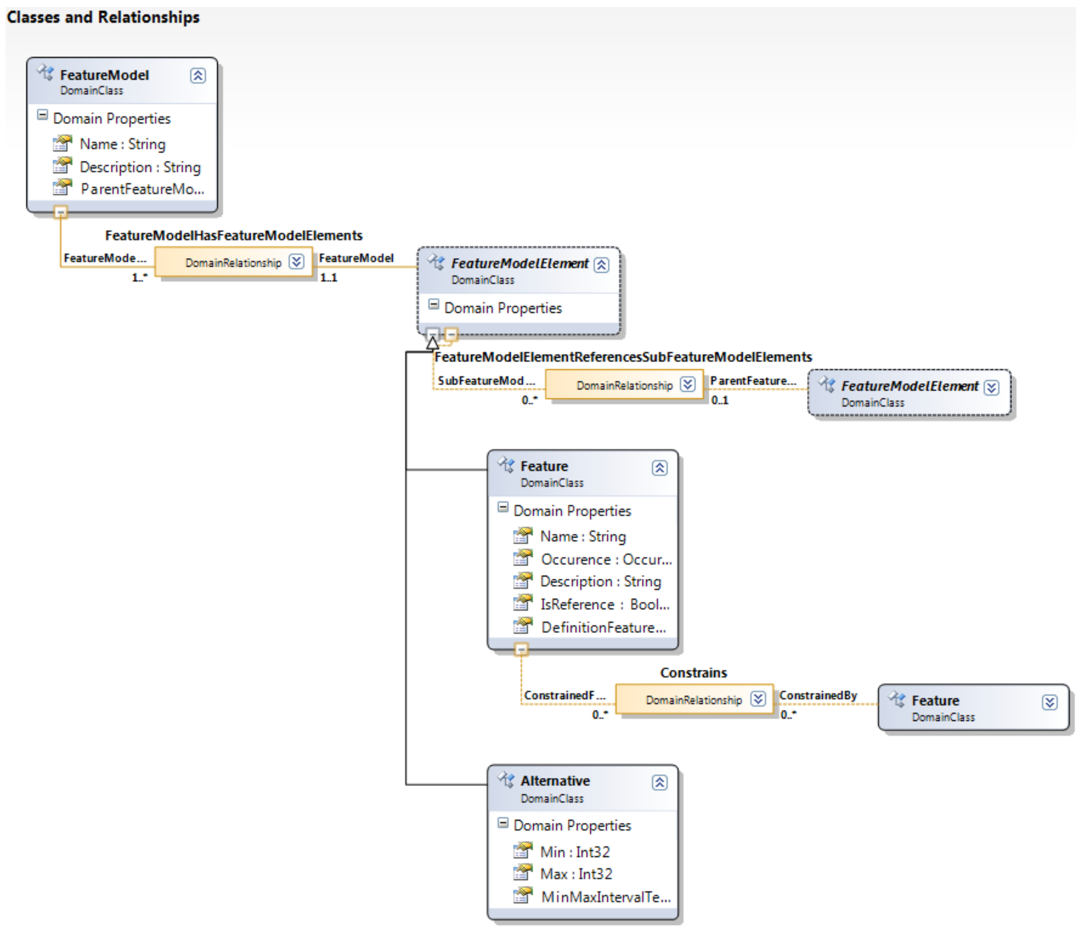


Figure 5-2: The domain model of FMD

Although this domain model varies from the model suggested in chapter 5.2, it is a suitable model to express features for software product lines. The system and components proposed in that model can be seen as features themselves, whereupon the system would be a mandatory feature and the components can be mandatory or optional. The model in FMD allows a more flexible design.

Of course, the provided domain model does not comprise tests. Introducing tests affects all parts of the extension, including the underlying domain model, the graphical notation, tool windows and the configuration window. The benefit of extending Feature Model DSL is a completely integrated graphical feature modeling experience including test representation and mapping. This is achievable in the given timeframe, so it was decided to proceed with this approach.

Another required step is the migration of FMDSL into a Visual Studio 2010 DSL Tools project, before it can be extended.

The decision to base the modeling on FMD does not affect the extension of GreenPepe 2010 to provide the functionality described in chapter 5.2.

Extending FMD leads to the following goals. FMD has to be migrated to be a Visual Studio 2010 DSL project. It has to be extended to introduce acceptance tests into the feature model and into the other program parts accordingly. GreenPepe2010 has to be extended to provide access to the query of tests in the solution as well as execution of tests and result presentation to other extension in the Visual Studio environment. A mapping between tests in the model and tests in the solution has to be realized. This can be achieved by consuming the extension points introduced to GreenPepe 2010.

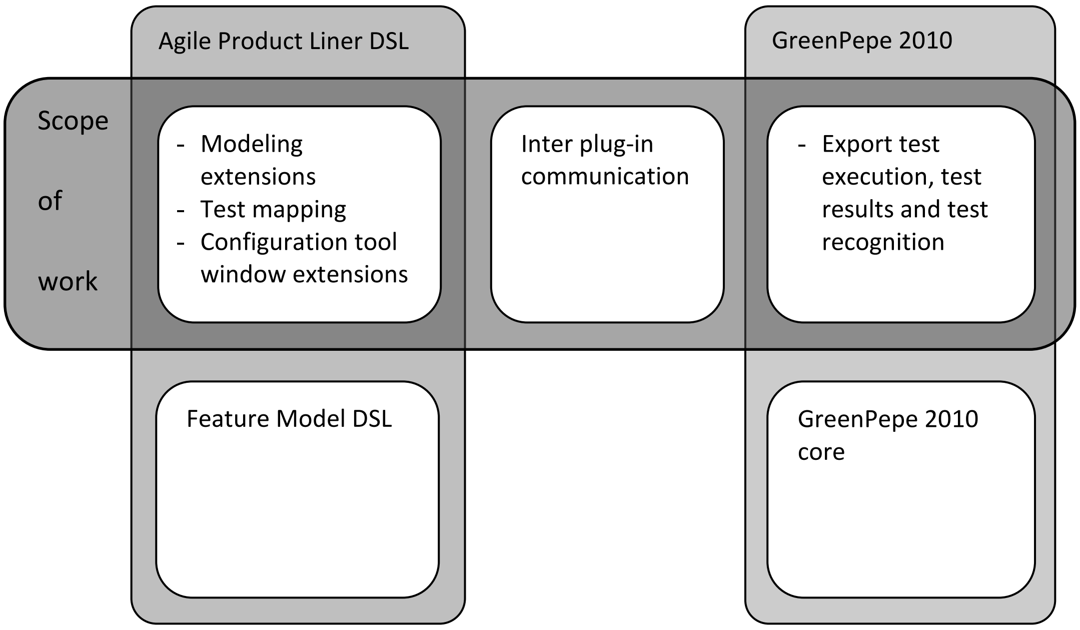

Agile Product Liner DSL
GreenPepe 2010
Scope of work
- Modeling extensions
- Test mapping
- Configuration tool window extensions
Inter plug-in communication
- Export test execution, test results and test recognition
Feature Model DSL
GreenPepe 2010 core

Figure 5-3: Scope of this work

Figure 5-3 illustrates the scope of this work and how Feature Model DSL and GreenPepe 2010 are extended and integrated.

5.4 Modeling Extensions

The DSL of FMD has to be extended with the following elements:

- **Tests**
  A test in the feature model represents an acceptance test.

- **Relationship between features and tests**
  Features can be linked to tests.

- **Exclude relationship between features**
  FMD offers a *Constrains* relationship between features. So far the only constraint that can be chosen is *require*. This relationship also has to offer an *exclude* constraint.

How these extensions are implemented is described in more detail in 6.3.

5.5 Mapping between Tests and Features

A mapping between acceptance tests and features can be realized with different levels of granularity. When we look at how test artifacts can be mapped to features it is important how acceptance tests are organized. This of course depends on the used testing framework. Executable specifications based on GreenPepper are stored in HTML pages. Each page can consist of several tests (which can be defined in tables or lists). The test pages are usually organized in a test project. This is illustrated in Figure 5-4. Three levels of granularity can be identified, which are described in the following.

Usually there are not many test projects. Most of the time there is just one per system under test. Although it might happen in theory that it would make sense to map a feature to a complete test project, this is very improbable.

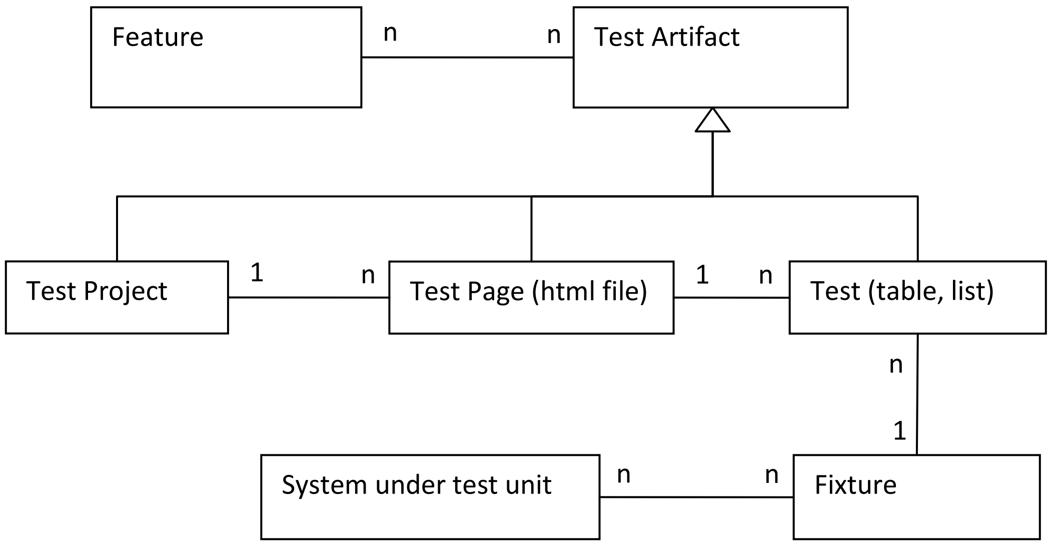


Figure 5-4: An object model for GreenPepper based test artifacts

Ghanam and Maurer propose to map features to tests on the test page level, which means tests are mapped to test tables inside a test page [5]. Although their ideas are based on the FIT framework and FitNesse, his approach is relevant as FIT and GreenPepper have many similarities and the table structure is almost identical. This approach has some advantages. All tests for a single feature can be organized in a single test page. In this case, variability moves to the test page level. Test tables can be specified as default, thus the feature would make no sense without the functionality tested at this point. Other test tables can be specified as optional. The tested functionality can be seen as an add-on. However, this approach has two disadvantages. To introduce variability on the test page level, the acceptance testing frameworks (in this case FIT or GreenPepper) have to realize the concepts of variability on the test page level by supporting keywords like *default* and *optional*. In order to map features to single test tables, test page files have to be parsed, and an object model representing all artifacts of a test page has to be created. This has to include test tables, test lists and text and make them accessible at runtime. That means a parser is needed but implementing such a parser takes too much time considering the limited timeframe of this work.

The third option would be to map features to test files, thus test pages. Instead of having tables that describe a certain part of functionality a whole test page is used for the same thing. This would mean a whole test page can be optional or default. A test page can, of course, include just a single test table or list. Consequently, if we consider having just one table per test page, the mapping to test pages and the mapping to single tables can mean the same thing. See Figure 5-5.

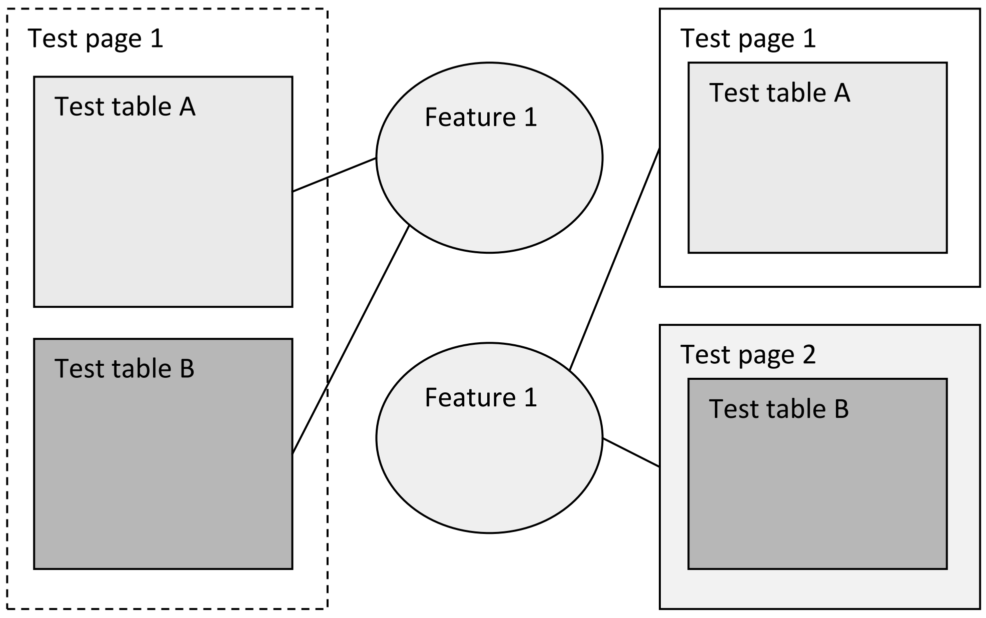


Figure 5-5: The two mappings express the same thing

To avoid the need of changes in the acceptance test framework itself the variability can be moved to the feature model. That means that, instead of introducing new keywords to the acceptance testing framework, the needed information is specified in the features that are mapped to the tests, thus in the feature model.

Of course this approach has limitations. The user has to spread related tables over different test pages, although it might be appropriate to group them in the same test page in terms of semantics. On the other hand, using this approach, no information about the content of a test page is needed. Hence, there is no need for a parser. Features can be mapped to files. Additionally, the acceptance testing framework can stay as-is and no customizations are needed, which is an advantage especially when new versions of the testing framework are released.

Considering the limited timeframe of this work, it was decided to take the third approach. A feature can be mapped to several test pages which are in fact several HTML files. Like described in Figure 5-4 the mapping between tests and features is n-to-n.

To implement this mapping in the graphical feature model, a 2-layered mapping is needed. One layer maps features to tests as part of the diagram. This mapping would be n-to-n. The second layer maps tests in the feature model to real acceptance test files in the file system. A test in the model can either be mapped to a file or not, thus the relation is 1-to-0..1. This is illustrated in Figure 5-6.

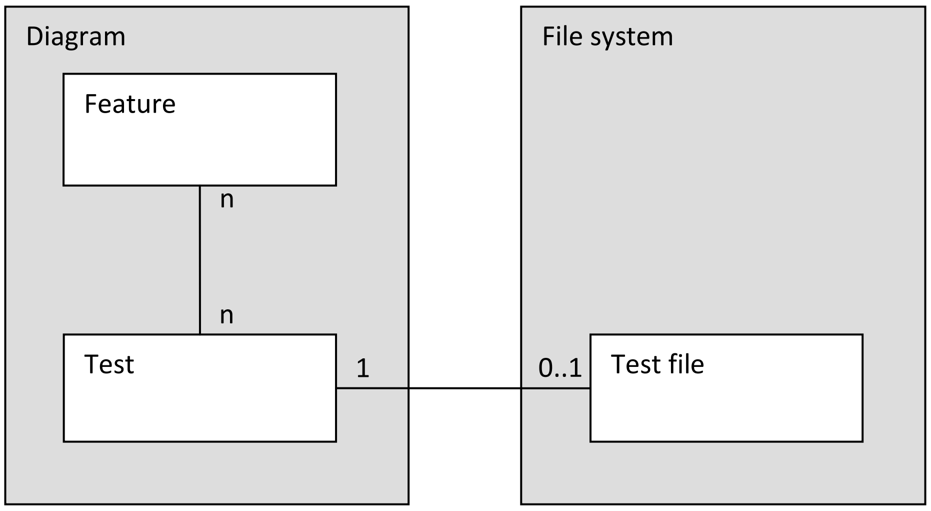


Figure 5-6: two-layered mapping between features and tests

5.6 Collapsing and Expanding of Test Nodes

A DSL developed with the DSL Tools offer graphical modeling in the form of diagrams. A system can consist of several features. Each of these can be mapped to several tests. The space a diagram can take on a screen is very limited, so if there are too many items in the diagram one can lose track quite easily. In order to keep the user from being overwhelmed, hiding of tests as well as collapsing and expanding of certain tests would be helpful. First of all, to offer collapsing and expanding, the user needs something to perform such actions. As the user interacts with a diagram by clicking on a specific area , this would be an intuitive approach. A good solution for that would be to use ports, which are described in 4.2.2. Ports are special shapes that are used on the outline of other shapes as end point of incoming or outgoing connections. In order to trigger collapsing and expanding, these ports can be made clickable. As the mapping between tests is n-to-n, some questions emerge on how collapsing and expanding can be realized.

One question is how many ports a test and a feature shape should have. A possible approach can be seen in Figure 5-7. A feature has a port for each connected test. Collapsing or expanding would be for each connection. It would be very unhandy to collapse and expand every single connection. Besides there would still be the question what happens when one of the connections to Test 2 has to be collapsed.

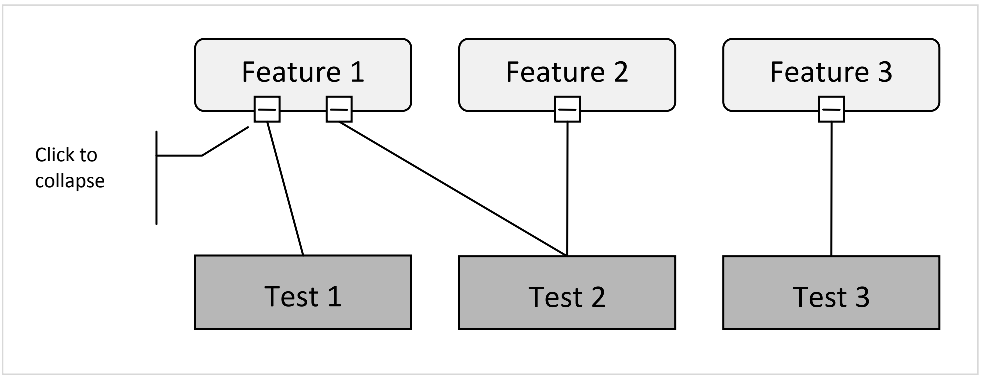


Figure 5-7: Collapsing with n ports per feature

As the modeling process is feature-centered another idea would be, that all test connections a feature has must end in a single port on the feature shape. See Figure 5-8. The user can show and hide all tests that are connected to and hence relevant for this feature by clicking the one existing port. But there is still the question of what happens with Test 2, which is connected to two features, if the port of one of these features is clicked. If only the connection is hidden, there is no graphical hint how many connections this test actually has.

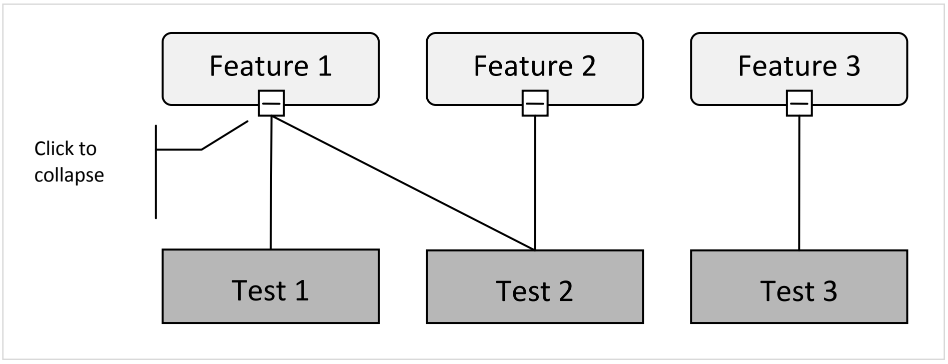


Figure 5-8: Collapsing with one single port per feature

In order to solve this issue, one solution would be to introduce an extra test shape for each connection a test has. That means that for every connection between a feature and a test a separate test shape is introduced. The feature has its own test shape. To indicate that different shapes are actually representing the same test, a new relation is introduced (Figure 5-9). If the port of a feature is clicked, all connected tests are hidden, while all tests connected to other features are still visible.

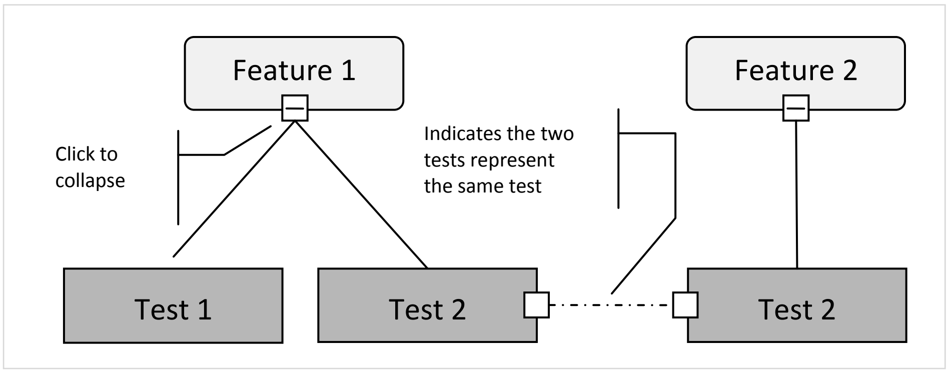


Figure 5-9: More than one shape representing the same acceptance test

To indicate that a test is actually connected to more than one feature, ports are also introduced for tests. This can be seen in Figure 5-10. This approach has two crucial disadvantages. The connections between test shapes presenting the same test grow exponentially. The original goal, to improve the overview, would not be reached and even worse, clarity would even decrease with every other connection a test has. Another, even more important issue is that the meaning of the diagram would be destroyed. A user would expect that there is a one-to-one relation between an actual test and the shape by which it is presented. Thus, two test shapes stand for two different tests.

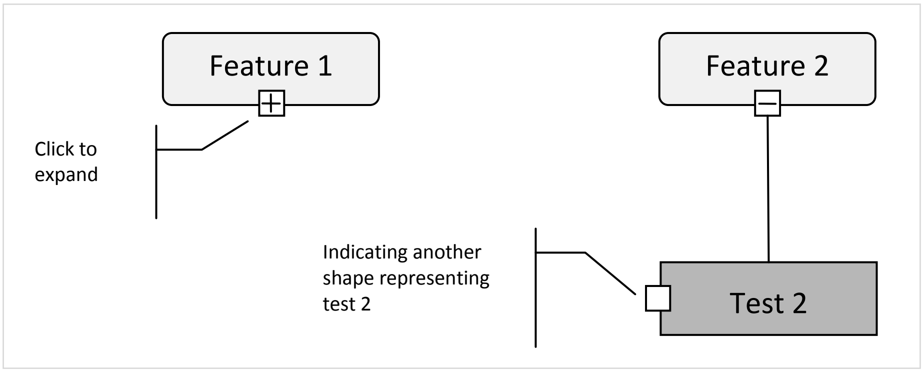


Figure 5-10: After collapsing

To avoid this, another approach can be considered. This time ports are used on both ends as connection end points for connections between features and test shapes. If a feature is connected to one or more tests it has a port that enables collapsing and expanding. A test shape has a port for each connection to a feature. Collapsing hides all connections. Test shapes only get hidden if the last visible connection has to be hidden. An example can be seen in Figure 5-11. After collapsing the children of *Feature 1*, *Test 1* is hidden, because it is only connected to *Feature 1*, whereas *Test 2* is still visible because of its connection to *Feature 2*. The port on the test shape of *Test 2*, that has no outgoing connection shows, that the test is connected to another feature. This last approach is used for APLD.

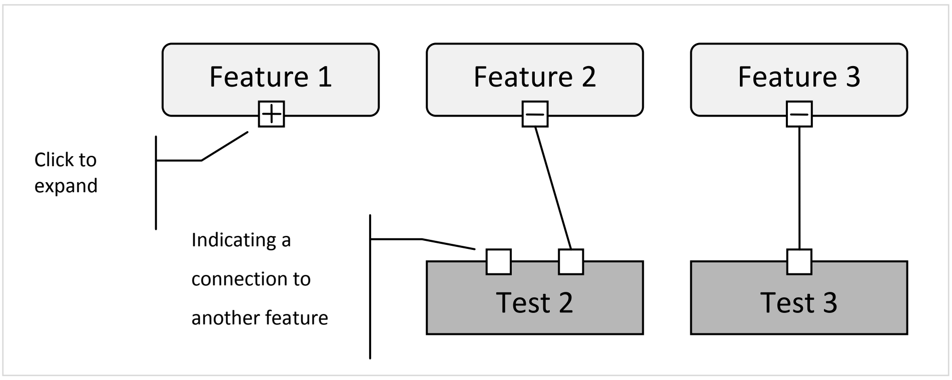


Figure 5-11: Possible Solution for Collapsing

Instead of collapsing tests individually it should also be possible to hide all at once. To realize this, *Hide all tests* and *Show all tests* commands can be added to the context menu of the diagram.

5.7 Summary

In this chapter a plan to realize a test-based feature management was developed, following the given prerequisites. A set of required functionalities was identified whose main elements are a feature model including tests, mapping to existing test files and the execution of those as well as a configuration tool that allows the selection of features to create a configuration. Existing feature modeling tools were investigated, the possible extension of those was evaluated and as a result it was decided to extend FMD. It was investigated how the test mapping can be realized and it was decided to map to test files. A collapsing and expanding strategy was developed to keep the overview in the model. It was elaborated what elements have to be added to the DSL provided by FMD. In order to query and execute tests it was planned to utilize GP2010.

6 Implementation

After a concept has been developed this chapter shows the key development steps to realize this concept. As mentioned in the previous chapter APLD is based on FMD. First, FMD has to be migrated to be a VS2010 project. Its DSL is then extended to provide test-based feature modeling. The resulting DSL Tool including the extended DSL is called APLD. GreenPepe2010 is extended to offer the query of tests in the solution and the execution of tests via MEF. APLD uses the functionality offered by GreenPepe 2010 to realize test mapping, test execution from the model diagram as well as the configuration tool window and representing test results in both, the diagram and the tool window.

6.1 The Structure of APLD

APLD is a DSL Tools project and therefore adheres to the structure described in chapter 4.3. In APLD many customizations are introduced and are located in the *CustomCode* folders of the two projects *Dsl* and *DslPackage*. All generated code is in the same namespace, which is predefined by the DSL Tools and cannot be changed. Therefore, the namespace of FMD was renamed to *UofFCASE.AgileProductLinerDSL*. As described in chapter 4.4 the concept of partial classes is used to customize the generated code, which implies that all classes are in the same namespace. To structure the project, partial classes are further spread over different folders, which describe the included functionality by name. All code that describes functionality that does not fit in a special description is located in the folder *CustomCode* directly.

Dsl project

- **Bounding**
  Custom code that ensures the connection points on ports
- **CollapsingExpanding**
  Code that is related to the collapsing and expanding of tests

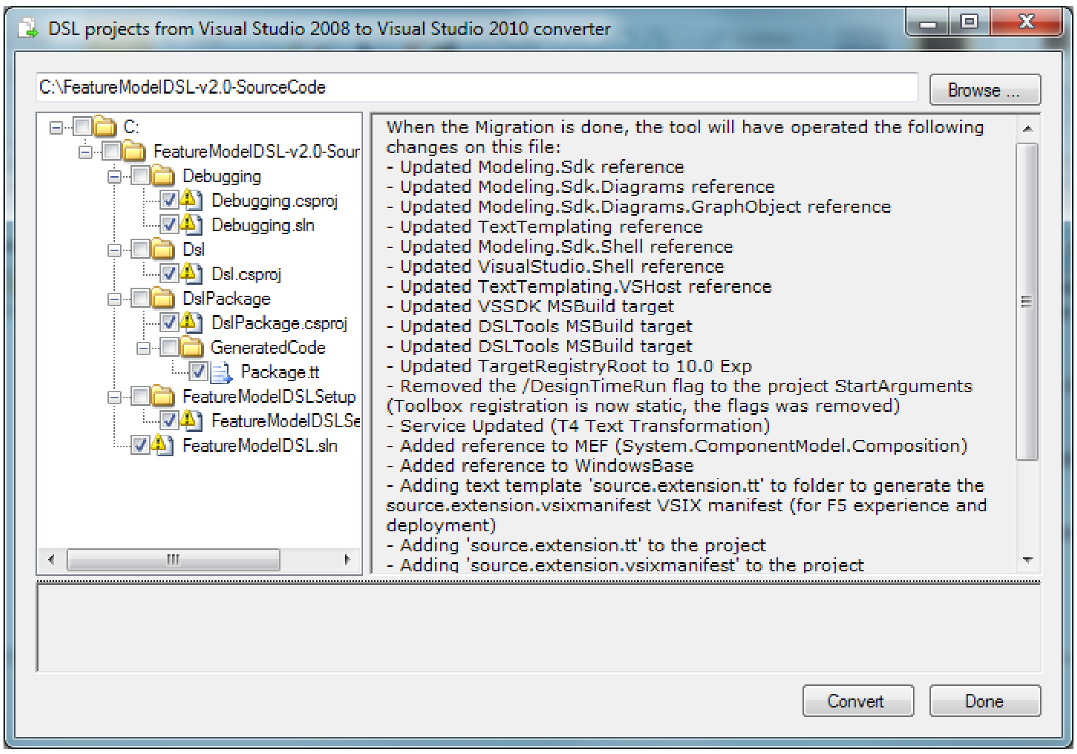
- **ConnectionBuilders**
  Custom connection builders that describe how connections are created in the diagram
- **Decorators**
  Custom decorators
- **Deletion**
  Custom code which is related to deletion
- **DiagramColoring**
  Code related to the custom coloring of model elements
- **Util**
  Helper classes related to the DSL itself
- **Validation**
  Custom validation code

DslPackage project

- **Commands**
  Custom commands in the context menu of the diagram are defined here
- **Confeaturator**
  All code that is related to the configuration tool window
- **Handler**
  Classes that forward actions to the DSL or to GreenPepe 2010
- **Mapping**
  Code that is related to the mapping of test files
- **Util**
  Helper classes related to the Visual Studio environment

6.2 Migration to Visual Studio 2010

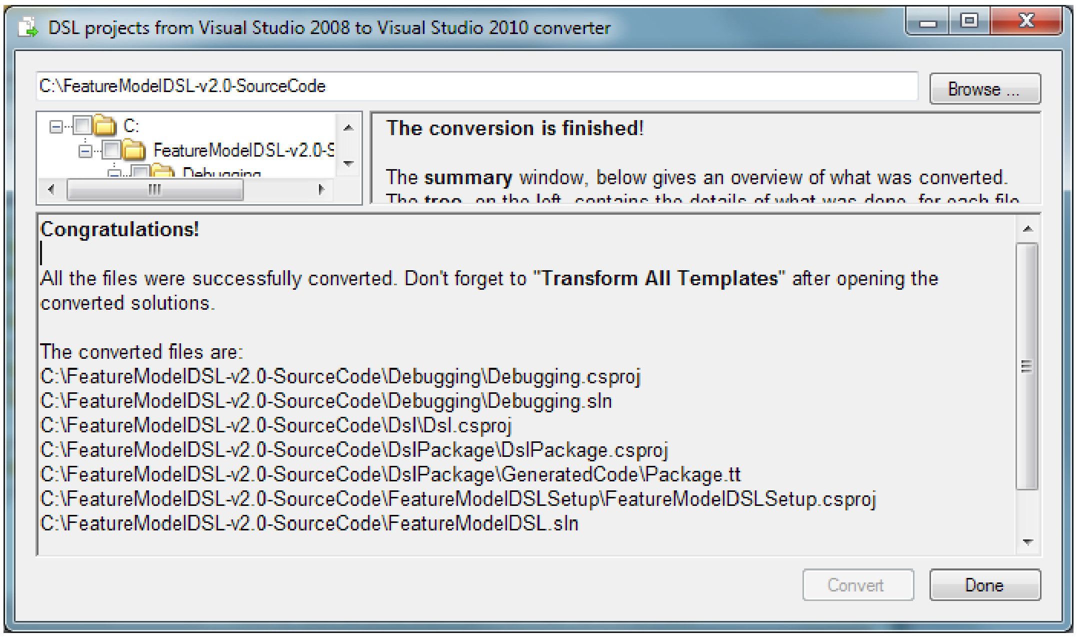
Before Feature Model DSL can be extended it has to be migrated to be a Visual Studio 2010 DSL project. The Visual Studio 2010 DSL SDK includes a migration tool to migrate existing Visual Studio 2008 DSL projects to work with the new version of Visual Studio.



```
When the Migration is done, the tool will have operated the following
changes on this file:
- Updated Modeling.Sdk reference
- Updated Modeling.Sdk.Diagrams reference
- Updated Modeling.Sdk.Diagrams.GraphObject reference
- Updated TextTemplating reference
- Updated Modeling.Sdk.Shell reference
- Updated VisualStudio.Shell reference
- Updated TextTemplating.VSHost reference
- Updated VSSDK MSBuild target
- Updated DSLTools MSBuild target
- Updated DSLTools MSBuild target
- Updated TargetRegistryRoot to 10.0 Exp
- Removed the /DesignTimeRun flag to the project StartArguments
(Toolbox registration is now static, the flags was removed)
- Service Updated (T4 Text Transformation)
- Added reference to MEF (System.ComponentModel.Composition)
- Added reference to WindowsBase
- Adding text template 'source.extension.tt' to folder to generate the
source.extension.vsixmanifest VSIX manifest (for F5 experience and
deployment)
- Adding 'source.extension.tt' to the project
- Adding 'source.extension.vsixmanifest' to the project
```

Figure 6-1: VS 2010 DSL Tools migration dialog

The tool is a separate program which cannot be executed from within the Visual Studio IDE. After the migration tool has completed its work only minor changes have to be made in order to get the plug-in working in Visual Studio 2010. The setup project had to be removed as the deployment has changed to the new VSIX container file. This is created automatically at compile time.


DSL projects from Visual Studio 2008 to Visual Studio 2010 converter
C:\FeatureModelDSL-v2.0-SourceCode
Browse ...
C:
FeatureModelDSL-v2.0-S
Debugging
Convert
Done

The conversion is finished!

The **summary** window, below gives an overview of what was converted.
The **tree**, on the left, contains the details of what was done, for each file

Congratulations!

All the files were successfully converted. Don't forget to "**Transform All Templates**" after opening the converted solutions.

```
The converted files are:
C:\FeatureModelDSL-v2.0-SourceCode\Debugging\Debugging.csproj
C:\FeatureModelDSL-v2.0-SourceCode\Debugging\Debugging.sln
C:\FeatureModelDSL-v2.0-SourceCode\Dsl\Dsl.csproj
C:\FeatureModelDSL-v2.0-SourceCode\DslPackage\DslPackage.csproj
C:\FeatureModelDSL-v2.0-SourceCode\DslPackage\GeneratedCode\Package.tt
C:\FeatureModelDSL-v2.0-SourceCode\FeatureModelDSLSetup\FeatureModelDSLSetup.csproj
C:\FeatureModelDSL-v2.0-SourceCode\FeatureModelDSL.sln
```

Figure 6-2: Migration completed

6.3 Extending the DSL of Feature Model DSL

6.3.1 Introducing Tests

The first step to extend FMD is to extend its domain model as it is the core of the DSL. So far it consists of the following domain classes (also see Figure 5-2):

- **FeatureModel**
- **FeatureModelElement**
  - **Feature**
  - **Alternative**

Feature and *Alternative* are derived from *FeatureModelElement*.

And the domain model consists of the following domain relationships:

- **FeatureModelHasFeatureModelElements**
- **FeatureModelElementReferencesSubFeatureModelElements**
- **Constrains (Requires only)**

To introduce tests to the domain-specific language a new domain class called *Test* is needed. As a test is not a feature model element, the new domain class *Test* is not derived from the domain class *FeatureModelElement*.

The *Test* domain class firstly needs a property test name. To identify a test file unambiguously in the solution the relative path to the test file and the unique project name, to which the file belongs, are needed. Besides this information, it makes sense to store if a test is actually mapped in a Boolean value. As APLD also allows test execution from the model diagram, the result of the last test run needs to be stored. To do so an enumeration *TestResult* is introduced containing the following literals:

- **None**

  There is no test result stored for the test class, which means the test of the model is not mapped to a test in the solution, the test was mapped but never executed from the model or the test result was reset.

- **Exception, Fail, Ignored, Successful**

  The execution of GreenPepper acceptance tests can have these four results.

The Test domain class and its domain properties can be seen in Figure 6-3.

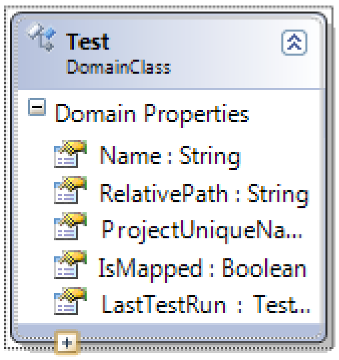


Figure 6-3: Test domain class and its domain properties

For defining the visual appearance of a test in the model diagram a shape has to be introduced that represents a test.

A geometry shape is the right choice. It is called *TestShape*. A geometry shape can also have domain properties to store information needed for the visual representation. Additionally it can have decorators. Like described in chapter 4.2.2, there are three types of decorators. Text decorators add text, and icon decorators are used to put icons on a connector or a shape. Text and icon decorators can be used on all kinds of connectors and shapes. There is a special third decorator that is used to paint a collapsing/expanding icon on shapes. It has a predefined icon and allows expanding and collapsing of compartment shapes. *TestShape* has a text decorator to present the test name on the shape. The *IsCollapsed* domain property is needed for collapsing/expanding (see chapter 6.3.6).

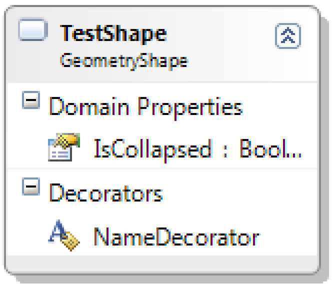


Figure 6-4: TestShape geometry shape and its domain properties and decorators

6.3.2 Ports Attached to Tests and Features

To realize the collapsing/expanding strategy designed in chapter 5.6, it is necessary to attach ports to tests and features. As mentioned in 4.2.2, a port is a shape which, except for being attached to the outline of a parent shape, is similar to a geometry shape. This means a port has also be mapped to a domain class that describes the port in the model. The port on a feature will serve as connection end point for connections to tests. As a result its domain class is called *TestPort* and its shape, the port itself, is called *TestPortShape*. Consequently, the domain class of ports attached to the outline of tests, which serves as end points for connections to features, is called *FeaturePort* and the port

FeaturePortShape. Thus, the names of the ports are exactly the opposite of their parent names. Figure 6-5 illustrates this.

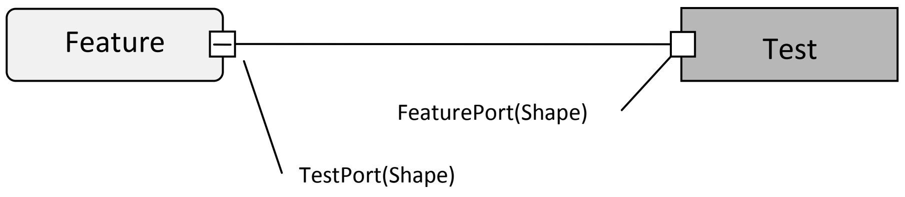


Figure 6-5: The naming of ports on features and tests

To actually attach a port to a parent in the domain model, a domain relationship is needed. A port cannot exist without its parent shape that is why it is necessary to use an embedding relationship. As a port is directly attached to the outline of its parent this relationship does not need to be visualized, thus there is no need to introduce a connector and map it to the relationship. To relate the test port domain class to the feature domain class the embedded relationship *FeatureHasTestPort* is introduced. As defined in 5.6, a feature can either have no or one test port, therefore the multiplicity is *ZeroOne*, whereas a test port can belong to a single feature, thus the multiplicity is *One*. See Figure 6-6.

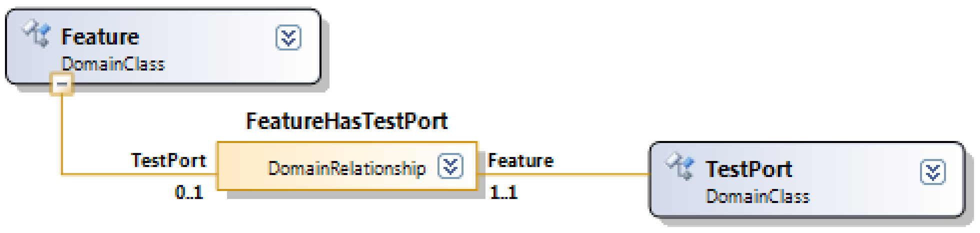


Figure 6-6: The FeatureHasTestPort domain relationship

The other introduced domain relationship is *TestHasFeaturePorts*. The name already indicates a test can have any number of feature ports. That means the multiplicity is *ZeroMany*. A feature port belongs to a single test. See Figure 6-7.

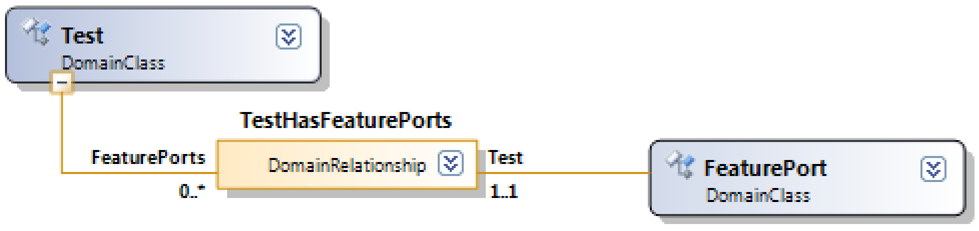


Figure 6-7: The TestHasFeaturePorts domain relationship

To present instances of TestPort and FeaturePort in the diagram, of course, ports are used as shapes, which can be seen in Figure 6-8. The *IsCollapsed* domain property is needed for collapsing/expanding (see chapter 6.3.6).

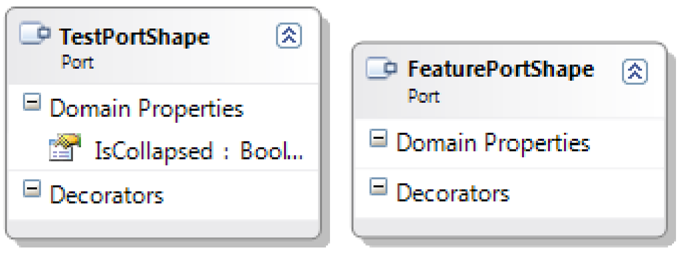


Figure 6-8: TestPortShape and FeaturePortShape

6.3.3 Introducing Relationship between Tests and Features

Because ports are introduced as connection endpoints (see 6.3.2), links between features and tests have in fact to be between test ports and feature ports. Like defined in the collapsing and expanding strategy in 5.6, links share a single test port, but every link has its own feature port. This indicates a multiplicity of ZeroOne for test ports and ZeroMany for feature ports. Tests and feature can exist in the model without being related to each other. Because of that the appropriate relationship between them is a reference relationship. Figure 6-9 shows the relationship how it is implemented in APLD.

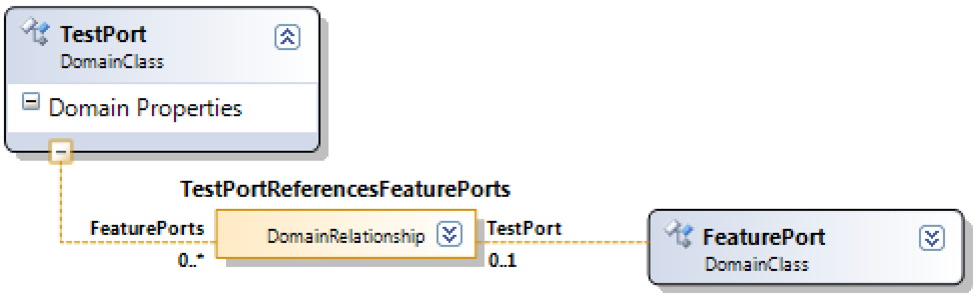


Figure 6-9: The TestPortReferencesFeaturePorts relationship

To represent the relationship, a connector is needed. The connector is called *TestConnector*. Figure 6-10 illustrates this. The *IsCollapsed* domain property is used for collapsing/expanding (see chapter 6.3.6).

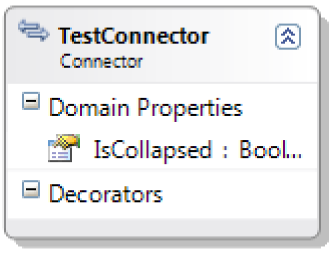


Figure 6-10: The connector to represent links between features and tests

6.3.4 Constrains Relationship

FMD includes a relationship called *Constrains*. Both roleplayers of this relationship are of type *Feature*. That means one feature constrains another feature. In FMD the relationship has a domain property of type string, which is the basis for the text decorator of the relationship. Its default value is *Requires*. There is no validation implemented except for the hard coded constraint that the relationship can only have features as source and target role.

APLD introduces an *Excludes* constraint which is based on the *Constrains* relationship. The DSL Tools offer a special enumeration domain type named *Domain Enumeration*. A new *Domain Enumeration* called *Constraint* which includes the literals *Requires* and *Excludes*. A new domain property called *ConstraintType* which is of this enumeration type is added to the *Constrains* relationship. The text decorator of this relationship is connected to the new domain property. The *Constrains* relationship is illustrated in Figure 6-11.

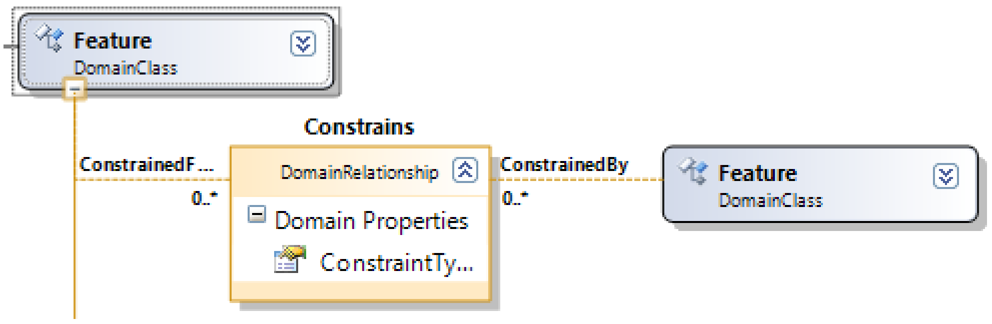
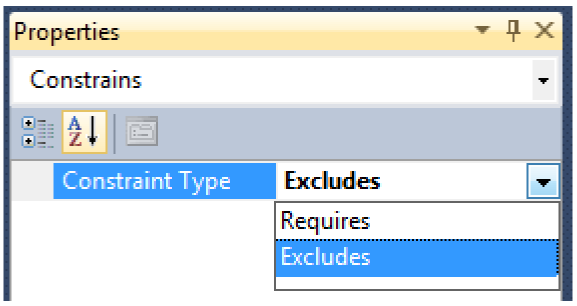


Figure 6-11: Constrains relationship with ConstraintType domain property

After a *Constrains* relationship has been placed the type can be set in its properties (see Figure 6-12).



Constraint Type: Excludes

Requires
Excludes

Figure 6-12: Constrains relationship properties

Validation

As mentioned before the *Constrains* relationship in FMD does not include any kind of validation. There are two areas where the validation of constraints applies, firstly in the model itself, which specifies which elements can constrain other elements and secondly in the configuration where it has to be determined if a certain combination of features violates constraints.

The *Exclude* and the *Require* constraint do not make sense between features of the same branch. That a child element requires its parent is implicit, while a parent that requires its child is not possible. Similarly, a child that excludes its parent is not possible, same for a parent that excludes its child. Therefore custom validation methods are introduced that warn the user if there are constraints within the same branch of the model before the diagram is saved. They are located in the *Customcode\Validation* folder in the *Dsl* project.

6.3.5 Toolbox

As described in chapter 4.2.5 to allow the user to create elements and connections tools are required. These tools are linked to the diagram editor of the created DSL and show up in the Toolbox of Visual Studio when a diagram is opened. Figure 6-13 shows the tools of APLD. Two of the tools are introduced by this work, whereas the rest were already supported by FMD. The two added tools are the element tool called *Test,* which is used to add new tests to the feature model and the connection tool *ConnectTest*, which is responsible for adding connections between features and tests.

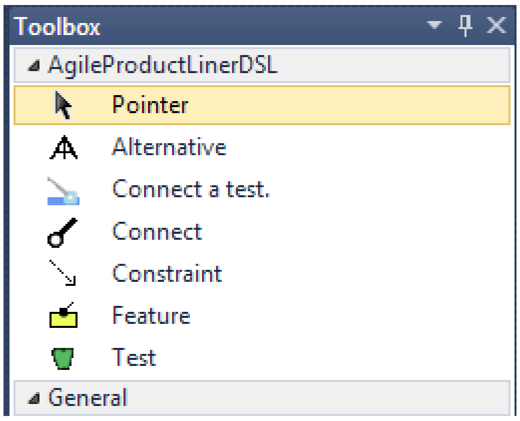
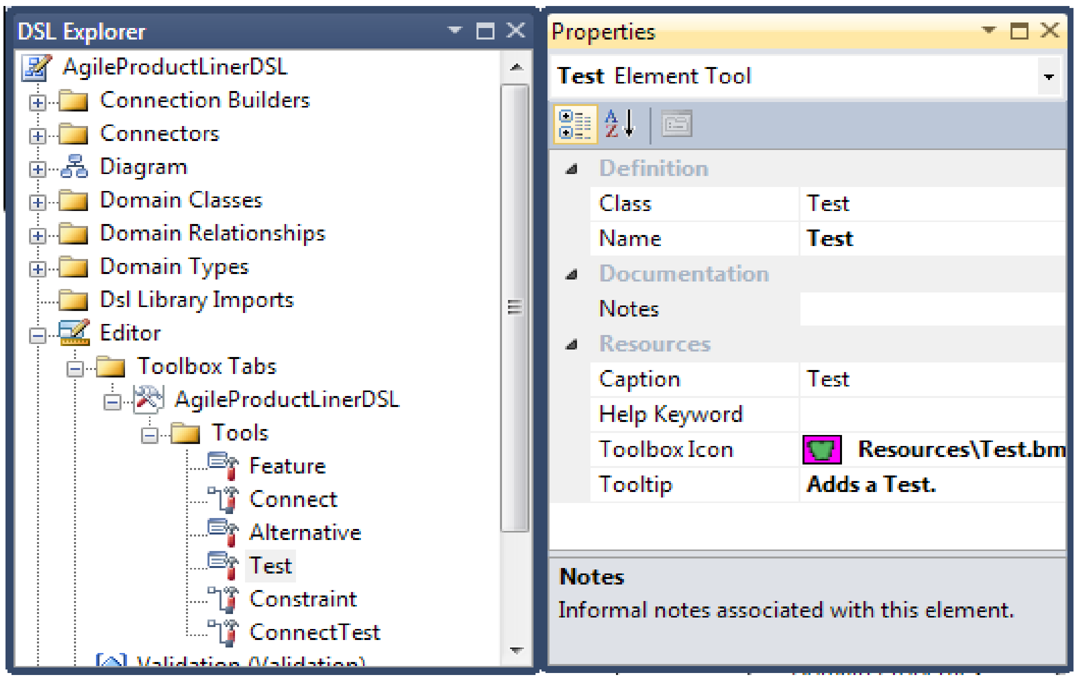


Figure 6-13: The toolbox of Agile Product Liner DSL

Tools are created in the DSL Explorer. In order to define a new element tool, the element class, a tool name and a caption have to be specified. Additionally, a tooltip, an icon, notes, and a help keyword can be set. Element classes know what has to be done in order to be created, therefore no special builder has to be specified. Figure 6-14 shows the DSL Explorer and the properties of the Test element tool.


DSL Explorer

- AgileProductLinerDSL
  - Connection Builders
  - Connectors
  - Diagram
  - Domain Classes
  - Domain Relationships
  - Domain Types
  - Dsl Library Imports
  - Editor
    - Toolbox Tabs
      - AgileProductLinerDSL
        - Tools
          - Feature
          - Connect
          - Alternative
          - Test
          - Constraint
          - ConnectTest
    - Validation (Validation)

Properties

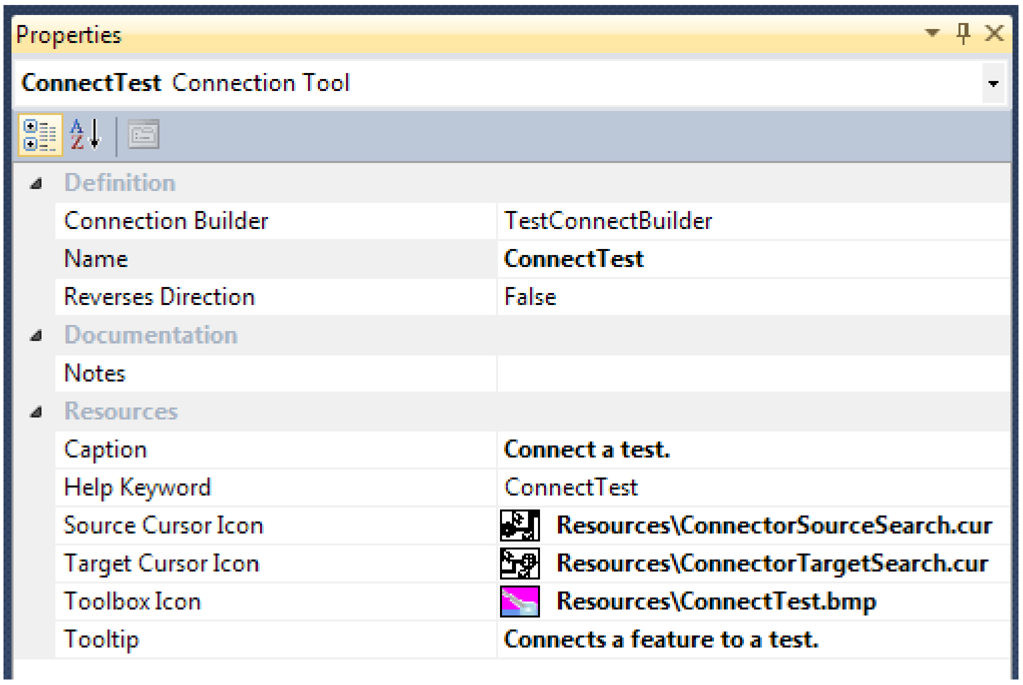
Test Element Tool

Definition	
Class	Test
Name	**Test**
Documentation	
Notes	
Resources	
Caption	Test
Help Keyword	
Toolbox Icon	**Resources\Test.bm**
Tooltip	**Adds a Test.**

Notes
Informal notes associated with this element.

Figure 6-14: Tools in the DSL Explorer and the properties of the element tool Test

Chapter 4.2.5 explained that connection tools use a connection builder to create connections. The *ConnectTest* connection tool uses the *TestConnectBuilder* class to create connections between features and tests (see Figure 6-15). A connection builder has to provide the methods *CanAcceptSource(…)*, *CanAcceptSourceAndTarget(…)* and *Connect(…)*. The first two methods check if source and target element are valid for the connection to be created. The last method performs the actual connection. The *TestConnectionBuilder* allows a feature as source and a test as target or the other way round. The built connection always has the test as target and the feature as source. The connection builder is also responsible for building *FeaturePorts* and *TestPorts* as needed (see 6.3.2) including the embedded relationships *FeatureHasTestPort* and *TestHasFeaturePorts*.



Definition	
Connection Builder	TestConnectBuilder
Name	**ConnectTest**
Reverses Direction	False
Documentation	
Notes	
Resources	
Caption	**Connect a test.**
Help Keyword	ConnectTest
Source Cursor Icon	**Resources\ConnectorSourceSearch.cur**
Target Cursor Icon	**Resources\ConnectorTargetSearch.cur**
Toolbox Icon	**Resources\ConnectTest.bmp**
Tooltip	**Connects a feature to a test.**

Figure 6-15: The properties of the ConnectTest tool of APLD

6.3.6 Collapsing and Expanding of Test Nodes

To implement the collapsing and expanding strategy developed in 5.6, several steps are necessary. The needed ports and the according domain classes have been introduced in chapter 6.3.2. Collapsing elements affects only the presentation while the domain model has to stay unchanged. Consequently only the following classes which are responsible for presentation are involved: *TestPortShape*, *TestConnector* and *TestShape*. In theory also *FeaturePortShape* has to be involved, as it also has to be hidden when collapsed. But as it is a child of the TestShape on whose outline it resides, its visibility is determined by that parent automatically.

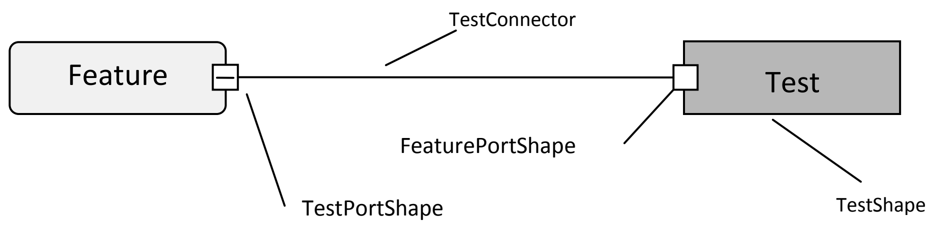


Figure 6-16: The four presentation elements involved in collapsing expanding

Each of the three presentation elements has a domain property (see 4.2.3) of the type Boolean called *IsCollapsed* to indicate if it should be visible or not. The kind of the two *IsCollapsed* domain properties of *TestConnector* and *TestShape* is set to *Calculated*. Both return the value that is returned by the *IsCollapsed* domain property of *TestPortShape,* whose kind is set to *Normal*.

TestPortShape has a custom decorator called *CustomExpCollapseField*. It is derived from the standard text decorator class *TextField* and overrides the method *GetDisplayText(ShapeElement parentShape)*. It returns a “+” or a “-“ sign depending on the *IsCollapsed* domain property of *TestPortShape.* In case *TestPortShape* is collapsed “+” is returned and “-“ otherwise.

When a *TestPortShape* is clicked the value of its *IsCollapsed* property is inverted which causes the custom decorator to show the opposite sign. Additionally, an event is raised, called *CollapsingChanged*. All instances of *TestConnector* and *TestShape* that are connected to the instance of *TestPortShape* are registered for that event. They update their visibility according to their calculated *IsCollapsed* property. If it is set to true, they are hidden and visible otherwise.

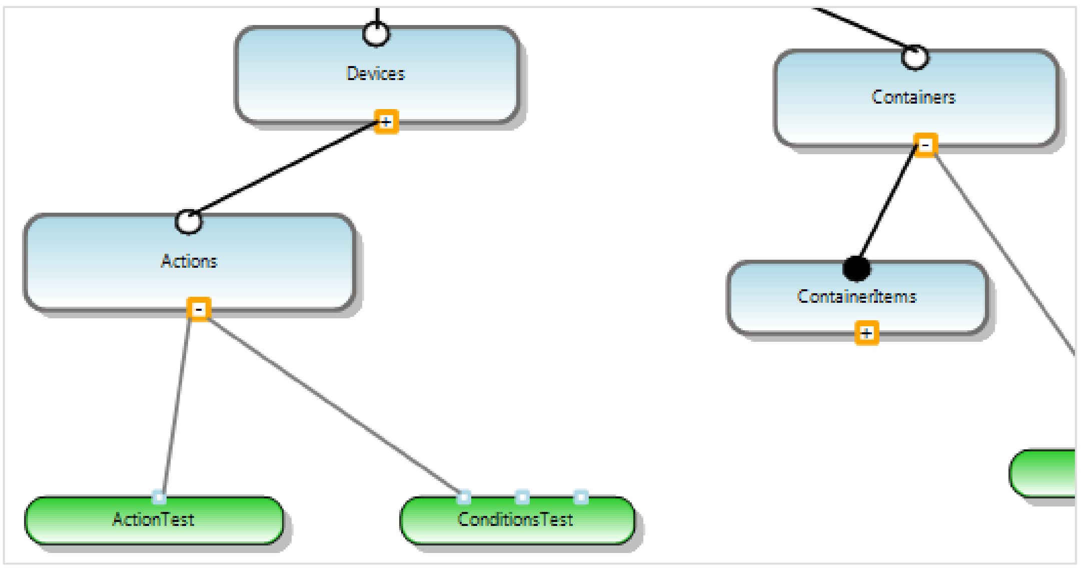


Figure 6-17: Expanding and Collapsing in APLD

Figure 6-17 shows how collapsing and expanding looks in APLD. The features *Devices* and *ContainerItems* have their children collapsed. One of its children is the test *ConditionsTest*. The two free ports indicate the connections to other features (in this case to the two above mentioned features). The collapsing of the tests under Actions would lead to *ConditionsTest* being hidden.

APLD additionally offers a hover effect. If the mouse enters a test shape all hidden connection will be shown and hidden again as soon as the mouse leaves the test shape area. If the mouse enters a port, only that connection gets visible.

6.3.7 Testing of the DSL

In order to test the DSL with automated tests a model has to be created programmatically. For this purpose a helper class called *ModelHelper* is used. The class creates an in-memory store (see chapter 4.2.4 The In-Memory Store) for the DSL. An example model is created by adding all needed model elements to the store. This includes domain classes, domain relationships between those classes, shape and connectors and shape and connecter maps. As explained in chapter 4.2.4 also the presentation elements are model elements in the store. Every model element instance is kept in a dictionary (a map collection type in C#) in the helper class for easy access. The tests get the model from the helper class and perform the tested functionality in the store.

The model created by the model helper is illustrated in Figure 6-18. The diagram shows the names of the elements in the dictionary used by the helper class. The test model currently does not include the *Constrains* relationship.

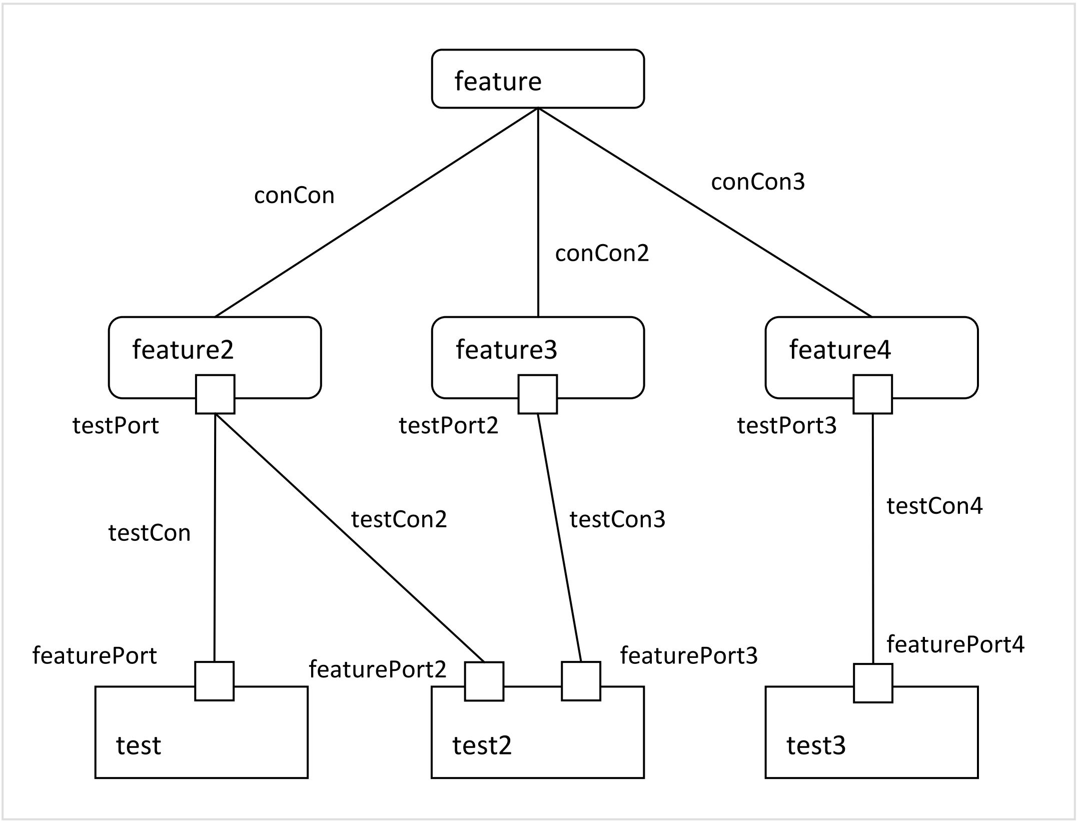


Figure 6-18: The feature model created by the model helper

6.4 Adding Commands to the Diagram Context Menu

In APLD many actions can be triggered from the diagram. To add new commands to the context menu of the DSL designer several steps are needed. The commands as well as the symbols have to be added to the *Commands.vsct* file in the *DslPackage* project. Then the command has to be added to the *AgileProductLinerDSLCommandSet.cs* class. Additionally two event handlers are needed for each command. A handler is responsible for checking if the command is applicable for the current selection, thus if the command should be visible and active in the context menu. It is recommended that this handler is called *OnStatus<commandName>*, whereas *commandName* stands for the actual name of the command. The second handler is responsible for performing the command and is recommended to be called *OnMenu<commandName>*.

The following commands were introduced in the context of this work:

- **Map to acceptance test in Solution**
  The command is visible if a single test is selected in the model. It opens the mapping dialog. (See chapter 6.6.1)

- **Unmap acceptance test**
  The command is visible if a single mapped test is select. It sets the test to be unmapped. (See chapter 6.6.1)

- **Import all unmapped acceptance test**
  The command is always visible. It imports all tests that are not mapped yet into the diagram, thus it creates new tests and maps them immediately. (See 6.6.1)

- **Run acceptance test**
  The command is visible if a single mapped test is selected. It executes the mapped acceptance test. (See chapter 6.6.2)

- **Run all acceptance tests under this node**
  The command is visible if a single feature model element is selected, thus a feature or an alternative. (See chapter 6.6.2)

- **Reset all test results**
  The command is always visible. It resets all test results by setting the test result domain property to none, which results in a blue color of the test shape. (See chapter 6.6.2)

- **Show all tests**
  The command is always visible. It shows all test shapes, thus it expands all tests that are connected to features and makes sure that all other tests are visible as well. (See chapter 6.3.6)

- **Hide all tests**
  The command is always visible. It collapses all tests that are linked to features and hides all other test shapes. (See chapter 6.3.6)

6.5 Providing Extensibility in GreenPepe 2010

With the .NET Framework 4.0 a new library is introduced called Managed Extensibility Framework (MEF). Visual Studio 2010 introduces MEF as a new way to extend the IDE and already offers several extension points based on MEF, mainly for the editor. Also the new DSL Tools offer model extensions via MEF.

6.5.1 Managed Extensibility Framework

The Managed Extensibility Framework (MEF) is a framework developed by Microsoft that simplifies the creation of extensible applications, offering discovery and composition capabilities [41]. It provides a standard way for applications to expose features and consume external extensions. It offers discovery approaches to locate and load available extensions. Moreover, it supports tagging extensions with additional metadata.

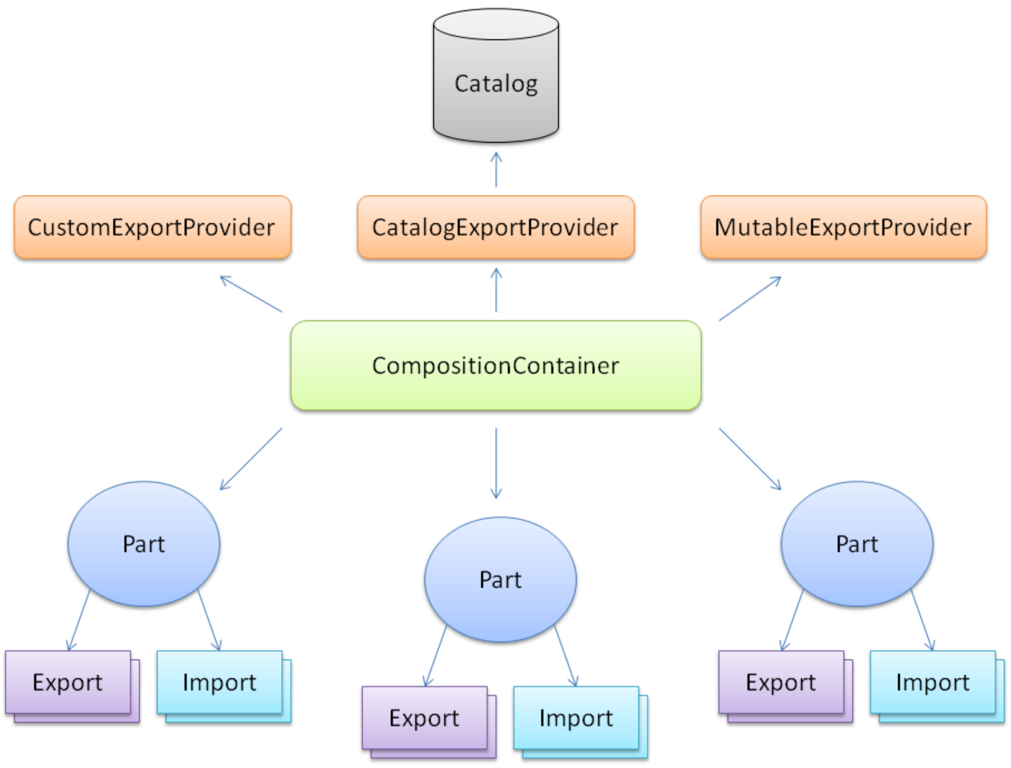


Figure 6-19: The concept of MEF [41]

The core components of MEF consist of a catalog and a composition container. The catalog offers discovery and the container coordinates creation and satisfies dependencies. To export or import services MEF introduces composable parts. These are attributed to declare exports and imports. Composable part can be added to a container explicitly or are discovered through the use of catalogs. They depend on contracts which are string identifiers. The container uses the contract information as well as the metadata to match up imports to exports.

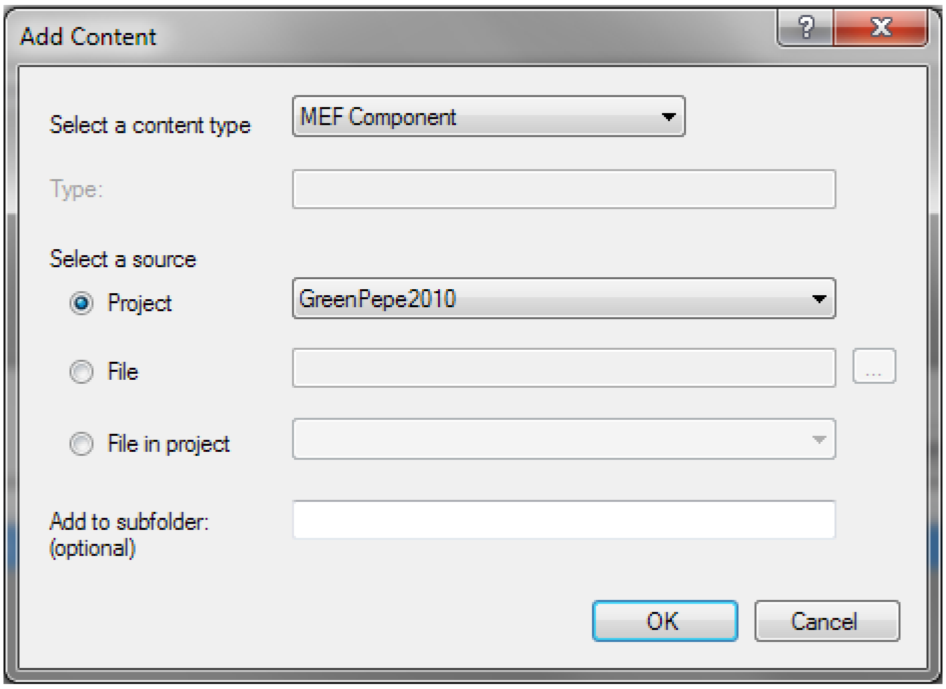
Visual Studio has its own implementation of a composition container which derives from the MEF composition container. Additionally it has its own export and catalog provider. Visual Studio scans certain directories for assemblies that are MEF Components and composes them. The directories are:

- %LocalAppData%\Microsoft\VisualStudio\10.0Exp\Extensions
- %VS10_Install_Dir%\Common7\IDE\Extensions
- %VS10_Install_Dir%\Common7\IDE\CommonExtensions

6.5.2 Exporting an Interface from GreenPepe 2010

In order to export functionality to the Visual Studio environment via MEF several steps are needed.

Firstly, GreenPepe2010 has to declare in its manifest, that it contains a MEF Component. To do so, the *source.extension.vsixmanifest* file has to be edited and the GreenPepe2010 project has to be added as MEF Component.



Select a content type: MEF Component
Type:
Select a source
[✓ Project] GreenPepe2010
File
File in project
Add to subfolder: (optional)

Figure 6-20: Add GP2010 as MEF Component

The next step is to define a composable part. This happens by attributing a class or a method. In order to ensure loose coupling an interface is defined that describes all functionality GreenPepe 2010 exports. The implementation of that interface is then attributed as export. The type is set to the type of the interface. Additionally the class is attributed as *shared*. This causes MEF to create this class as a singleton that means only one instance of that can be in the composition container.

```
    [Export(typeof(IGreenPepeService))]
    [PartCreationPolicy(CreationPolicy.Shared)]
    public class GreenPepeService : IGreenPepeService  {
       ...
    }
```

6.5.3 The exported Interface

Agile Product Liner DSL needs to query GreenPepe 2010 for all GreenPepper tests in the solution and it needs to command GP2010 to execute a given set of tests. Furthermore, it has to listen for the test results.

The *IGreenPepeService* offers the methods *GetAllTests()*, *RunTests()* and the event TestExecutionCompleted. The interface is implemented by the *GreenPepeService* class which uses three classes, *ExecutionAgent*, *GreenPepeTestExecuter* and *SolutionItems* (see Figure 6-21). The *ExecutionAgent* is responsible for executing tests for external callers. For each run a new *ExecutionAgent* instance is constructed given the list of tests to run. The *ExecutionAgent* checks if all relevant information is available and correct for each given test. This includes the path to the test file itself, the path to the project the test belongs to and the path to the assembly of the system under test.

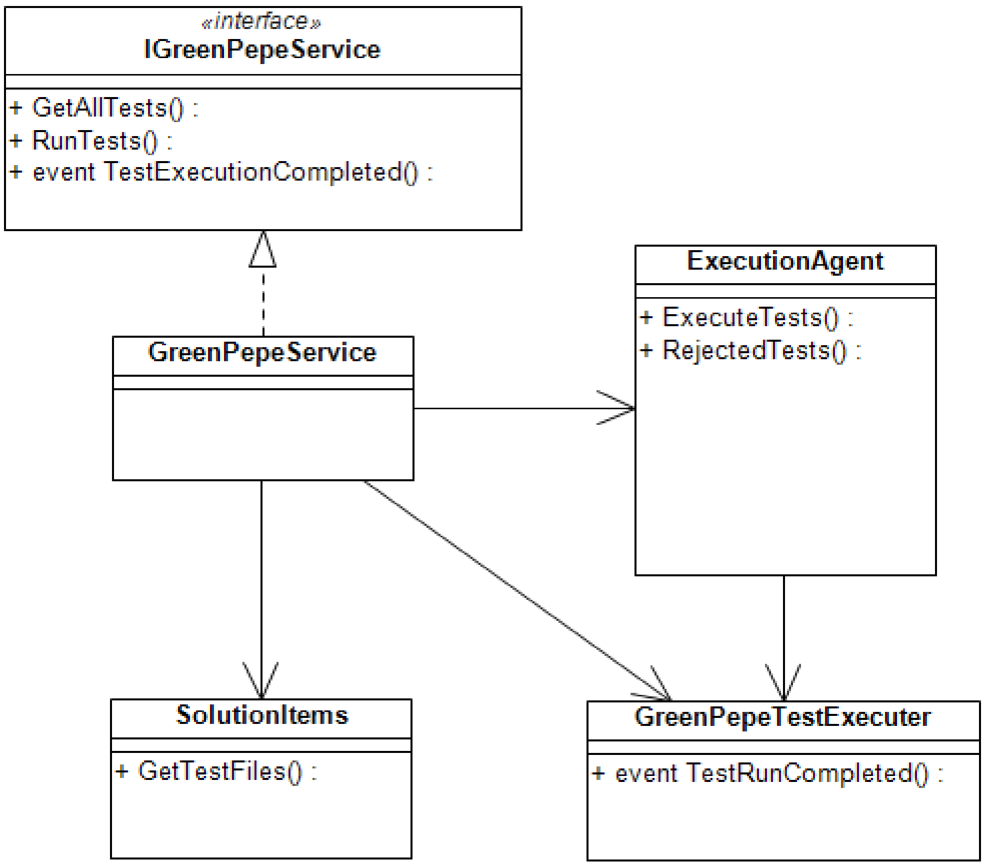


Figure 6-21: Providing extensibility in GreenPepe 2010

All tests that pass this check are then executed by the GreenPepeTestExecuter class, which is the class responsible for test execution in GreenPepe2010. The GreenPepeService also offers the TestExecutionCompleted event. In order to get informed about the test results it registers for the TestRunCompleted event raised by the GreenPepeTestExecuter.

For returning all tests in the current solution, the GreenPepeService uses the method *GetTestFiles()* of the *SolutionItems* which returns all HTML files that are marked as acceptance tests.

The sequence of calls for the execution of tests through the interfaces by another plug-in is illustrated in the following sequence diagram.

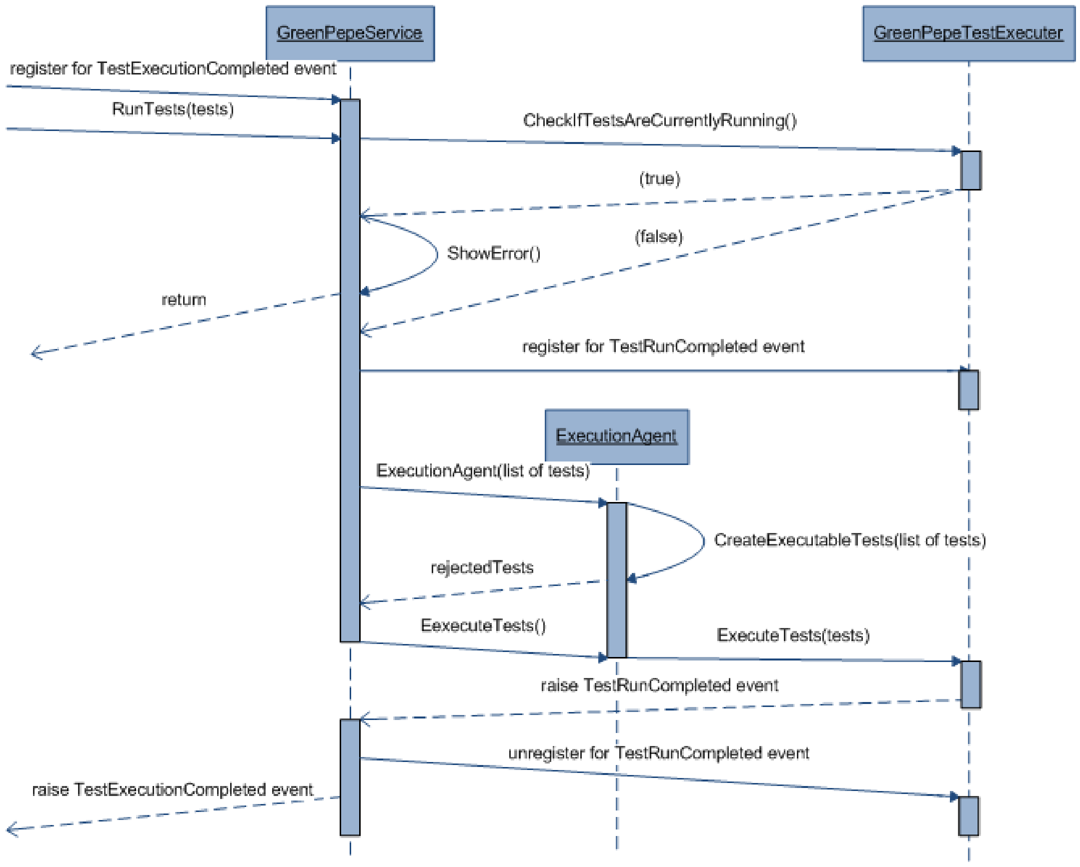

GreenPepeService
GreenPepeTestExecuter
register for TestExecutionCompleted event
RunTests(tests)
CheckIfTestsAreCurrentlyRunning()
(true)
ShowError()
(false)
return
register for TestRunCompleted event
ExecutionAgent
ExecutionAgent(list of tests)
CreateExecutableTests(list of tests)
rejectedTests
EexecuteTests()
ExecuteTests(tests)
raise TestRunCompleted event
unregister for TestRunCompleted event
raise TestExecutionCompleted event

Figure 6-22: Sequence diagram of the test execution

If the external caller is interested in the test result, it first has to register for the event *TestExecutionCompleted*. Then it can call the *RunTests()* method given a list of tests. The *GreenPepeService* then checks if the *GreenPepeTestExecuter* is currently executing tests. If so, an error message is shown and no tests are executed. If there are no tests in execution the *GreenPepeService* registers for the TestRunCompeted event of the

GreenPepeTestExecuter. Then it instantiates a new *ExecutionAgent* and passes the list of tests that have to be executed. The *ExecutionAgent* creates test objects that can be executed by GreenPepe2010. If some given tests do not meet the criteria needed to create executable tests these tests are rejected. Afterwards, the *GreenPepeService* calls the *ExecuteTests()* method of the *ExecutionAgent* which then calls the *ExecuteTests()* method of the *GreenPepeTestExecuter* passing the newly created tests. When the test execution is completed the *GreenPepeTestExecuter* raises the *TestRunCompleted* event. After being informed, the *GreenPepeService* then unregisters from the *TestRunCompleted* event and raises the *TestExecutionCompleted* event containing the test results and the list of rejected tests as arguments.

6.6 Consuming Extensibility of GreenPepe2010

Visual Studio recognizes that GreenPepe2010 has a MEF component and recognizes the export. It adds the contract to its catalogs which allows other packages in the environment to access it.

Imports are only satisfied by the composition container of Visual Studio, if the MEF Component also has exports that are needed by other MEF components. Otherwise the needed service has to be retrieved manually. To do so the component model service is needed. To get an instance of the implementation of the needed interface the method *GetService()* of *IComponentModel* interface has to be called. If it cannot get the designated instance, an exception is thrown. The following code snippet shows how the *IGreenPepeService* is retrieved.

```
IComponentModel componentModel = AgileProductLinerDSLPackage.
      GetGlobalService(typeof(SComponentModel)) as IComponentModel;
this.greenPepeService = componentModel.GetService<IGreenPepeService>();
```

Figure 6-23 illustrates the communication between Agile Product Liner DSL and GreenPepe2010.

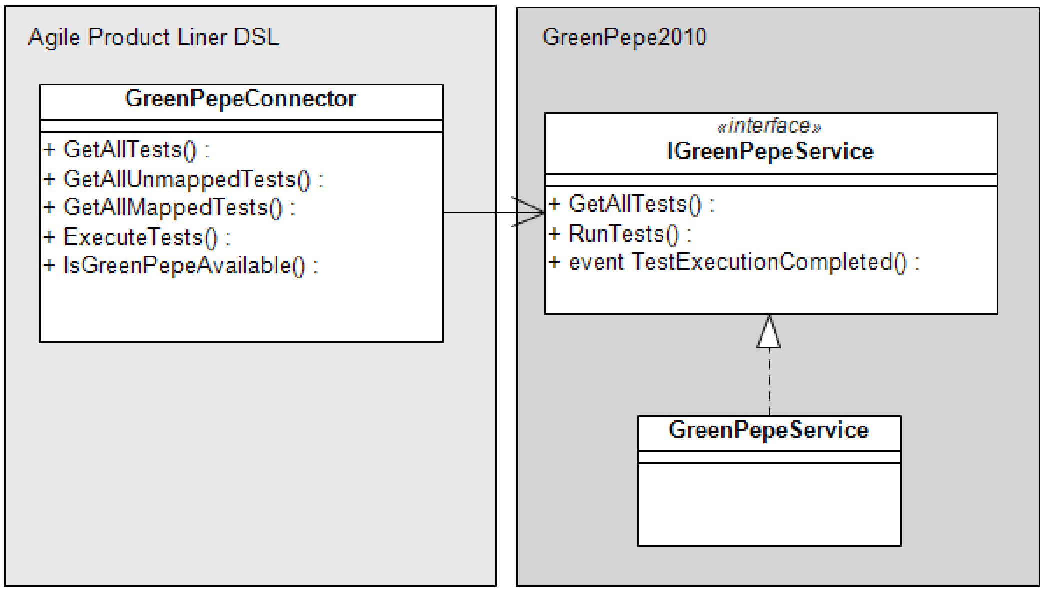


Figure 6-23: Communication between APLD and GP2010

6.6.1 Test Mapping

Mapping a test in the feature model to a test file in the solution involves several steps. First, a mapping command is needed. The command is applicable if a single test shape is selected in the diagram. Performing that command has to open a mapping dialog, which shows all tests found in the solution, that are not mapped yet. To get all tests of the current solution Agile Product Liner DSL uses the method *GetAllTests()* offered by the *IGreenPepeService* interface of GreenPepe 2010. It then checks for each returned test whether a test in the feature model is already mapped to it. This is the case when the *IsMapped* domain property of a test in the model is set to true and its *relativePath* and *projectUniqueName* domain properties are consistent with the according values of the given test. After selection of a test the actual mapping has to be performed and all necessary information has to be stored in the domain properties of the test model object.

The mapping dialog in Agile Product Liner DSL is a tree view containing all acceptance tests found in the current solution. The hierarchy of the tree view is the same as in the solution explorer of Visual Studio. Tests that are already mapped are shown in gray.

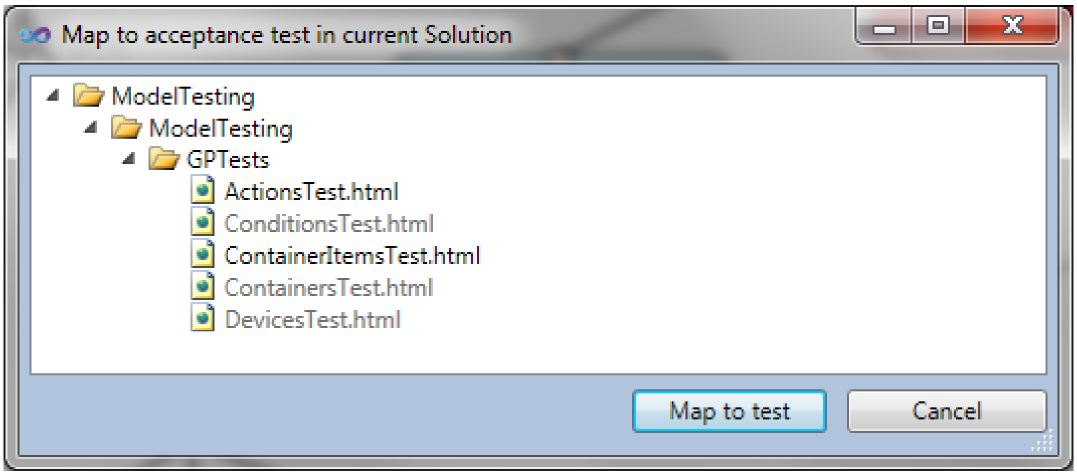


Figure 6-24: The test mapping dialog

Tests in the diagram which are not mapped to a test file in the solution have a gray background (a darker gray than model elements which are not part of the current configuration, see chapter 6.8). As soon as a test is mapped its background color is blue.

6.6.2 Execution of Tests, Result Presentation and Storage

To execute tests the method *RunTests()* in the *IGreenPepeService* interface is used. It takes a list of *TestInformation* objects. *TestInformation* is a container class that contains the relative path of the test file in the solution, the project unique name and if the test is mapped.

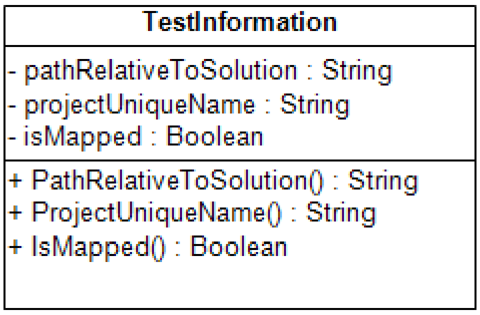

TestInformation
- pathRelativeToSolution : String
- projectUniqueName : String
- isMapped : Boolean
+ PathRelativeToSolution() : String
+ ProjectUniqueName() : String
+ IsMapped() : Boolean

Figure 6-25: The TestInformation container class

To get the test results, the *TestExecutionCompleted* event in the *IGreenPepeService* is used. It returns the test results as argument. The result is stored in the test (see chapter

6.3.1). The color of a test shape is dependent on its *TestResult* domain property. A successful test result will cause the test shape to paint itself with a green background. If the test result is set to *None*, the test shape is colored blue, whereas if the test result is *Failed*, *Exception* or *Ignored* the color will be red.

6.7 Extending the Configuration Tool Window

FMD offers a configuration tool window that allows the creation of a configuration, which means the selection of a subset of features of the feature model. The tool window is called Confeaturator. Like the domain model the Confeaturator has to be extended with tests as well. This makes it possible to check whether all tests that are mapped to the features in the current configuration pass. Additionally, the tests can be involved in the instantiation process.

The tool window was created with Windows Forms, the predecessor of WPF. Therefore the extensions are realized with Windows Forms as well. Everything related to the Confeaturator can be found in the *Confeaturator* folder which is located in the *CustomCode* folder in the *DslPackage* project.

6.7.1 Adding Tests to the Tree View

As defined before the multiplicity of the relationship between features and tests is n-to-n. The model of the tree view, of course is a tree. That means a child node can only have one parent. In order to add tests to the tree, regardless, a test might have to be added to the tree multiple times, once for each feature it is linked to. An example can be seen in Figure 6-26, where *ConditionsTest* appears twice in the tree view.

Tests in the model which are not connected to any feature do not appear in the tree view. Only tests which are connected to a feature are relevant for the configuration. To represent a test in the tree view a new node type is introduced, namely the class *TestTreeNode* which derives from *TreeNode*. It has a reference to a test object of the domain model and keeps the test name as well as an icon reference.

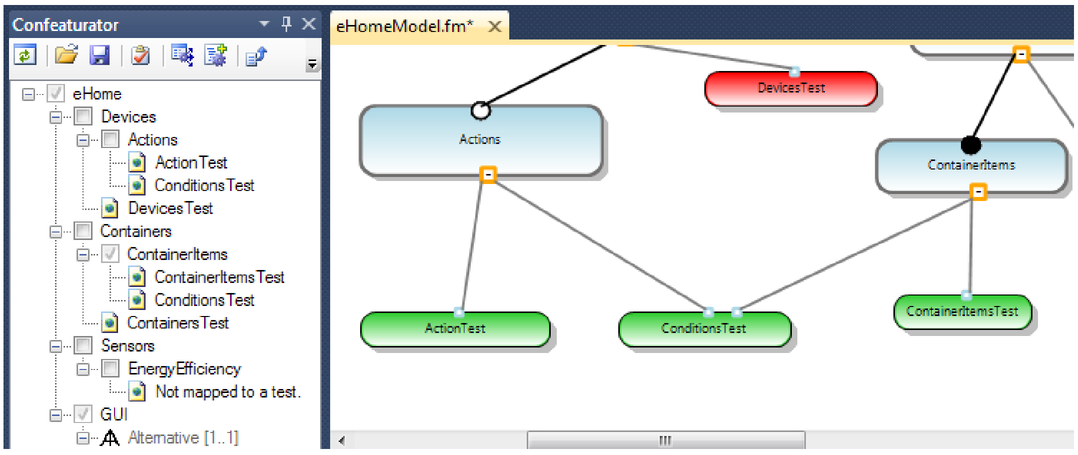


Figure 6-26: Confeaturator extended with tests, showing ConditionsTest twice

6.7.2 Execution of Tests from the Configuration Tool Window

After features have been selected in the Confeaturator and thus a configuration was created, it might be of interest, if the acceptance tests that are related to those features are passing. If the tests are passing, it can be assumed that the features are working and a working instance of the product line can be instantiated. Therefore, a new action is added to the Confeaturator that executes all tests that are linked to those features which are part of the current configuration. To give better feedback to the user, test results should be visible in the tree view after execution. Similar to the result presentation in the model diagram, test nodes get colored to indicate the test result. The background of the node gets colored green, if the related acceptance test passed, red, if the test failed. Test nodes which are not affected by the current configuration have a white background. Because of the fact that a test might occur multiple times in the tree view, all occurrences of this test get colored according to the test result, as soon as a single occurrence is part of the current configuration. This and the presentation of test results can be seen in Figure 6-27.

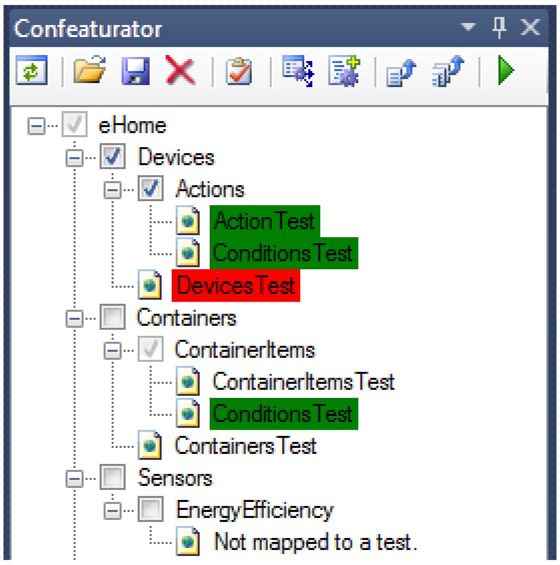
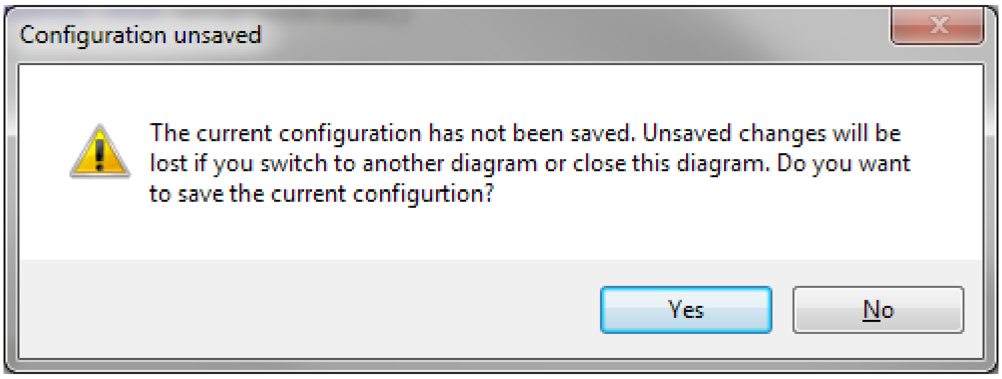


Figure 6-27: Result presentation after test execution in the Confeaturator

6.7.3 Synchronizing the Configuration Tool Window with the Diagram Editor Window

In FMD the configuration tool window Confeaturator is not synchronized with the diagram editor. If the user changes to another diagram, the tool window still shows the old tree and has to be refreshed manually. Besides that this is misleading, it causes issues when tests are executed from the Confeaturator, because test results are not only visualized in the tree but also the in the diagram. If the diagram does not represent the same model as the configuration window, this will cause errors. To avoid this, the configuration tool window in APLD is synchronized with the DSL designer. This means that, when the diagram is switched, the Confeaturator automatically refreshes its tree to reflect the model of the new diagram. When the tree view is changed programmatically the user needs to be able to save the configuration before that happens. To achieve that, the Confeaturator is set to *dirty* as soon the user selects features in the tree, which means the user entered configuration mode. When the user wants to open another file or diagram or if Visual Studio has to be closed, a warning will be shown and the user will be given the option to save the current configuration (see Figure 6-28).


Configuration unsaved

The current configuration has not been saved. Unsaved changes will be lost if you switch to another diagram or close this diagram. Do you want to save the current configurtion?

Figure 6-28: Configuration unsaved warning

As soon as the configuration is saved, the Confeaturator will be set to be *not dirty*. This can also be achieved manually by discarding the current configuration. To do so, a separate action is introduced which can be seen in Figure 6-29.

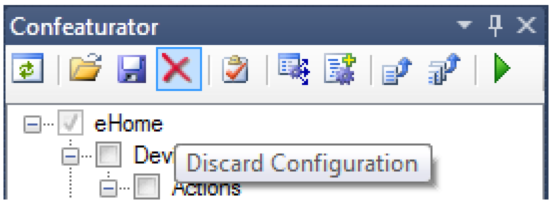


Figure 6-29: Discard configuration action

6.8 Graphical Reflection of the current Configuration in the Diagram

To give better visual feedback the current configuration is reflected in the model diagram in APLD. As mentioned in the previous chapter, as soon as a feature is selected the configuration mode is entered. This also triggers the coloring of the element in the diagram. Branches which are not part of the current configuration are colored in a light gray. This affects the connections between features, the connections between features and tests, the feature shapes, the test shapes, and the test port shapes. The color of alternative shapes and constraint relationships stay unchanged as well as the occurrence indicator of feature shapes (the small circles on the top of a feature). Figure 6-30 illustrates the model coloring. The feature *Devices* is not selected and therefore the whole branch is shown in gray.

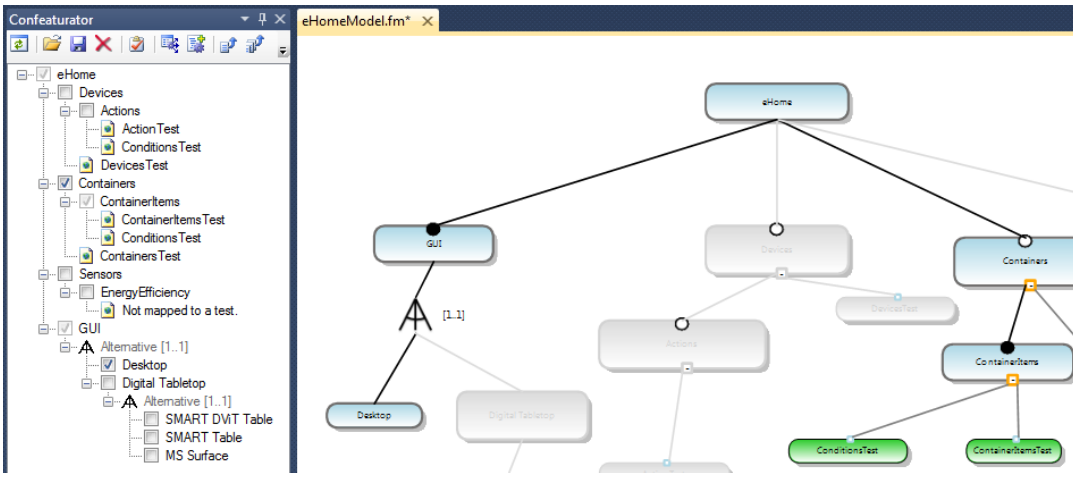


Figure 6-30: The graphical reflection of the current configuration in the diagram

Every time a feature is selected or de-selected all features that are in the current configuration are calculated and then the elements in the model are colored accordingly. To set the color, the affected classes (*TestPortShape*, *FeatureShape*, *ConnectConnector*, *TestConnector* and *TestShape*) provide the methods *GrayOut()* and *ResetColor()*. *GrayOut()* overrides the pen and brush settings of the presentation elements and *ResetColor()* clears the overrides and as a result the elements are painted in their original colors.

6.9 Summary

This chapter described how the developed concept was implemented. APLD was created by migrating FMD to VS2010 and extending its DSL and configuration tool window. The DSL was extended with tests, connections between tests and features and with the exclude constraint relationship and its validation. GreenPepe 2010 was extended to export an interface via MEF and offers querying of GreenPepper acceptance tests, execution of these, and the corresponding test results. APLD uses the provided functionality to realize the mapping of tests in the model to test files in the solution, the execution of tests from the diagram as well as from the Confeaturator.

7 Conclusion and Future Work

7.1 Problems

The implementation part of this work was conducted with the beta 1, beta 2 and the release candidate of the .NET Framework 4.0, Visual Studio 2010, the VS2010 SDK and the VS2010 DSL Tools. Even though these versions are surprisingly stable, they are not perfect.

After migrating Agile Product Liner DSL to beta 2 the following bug occurred: launching the project in the experimental instance of Visual Studio for debugging resulted in an empty toolbox. There were no items in the category *AgileProductLinerDSL*. Some research showed that this is a bug in beta 2 of the VS2010 DSL Tools and will only occur after the project has been launched in debug mode. To solve this issue, the toolbox cache has to be deleted. The cache can be found under *%UserProfile%\AppData\Local\ Microsoft\VisualStudio\10.0Exp,* where all *.tbd files have to be deleted. In order to debug, nevertheless, the following work around exists. The project has to be launched without debugging and the debugger has to be attached to the running instance afterwards. Right after the release it was not clear, what causes the bug, how it could be avoided, and that it is related to debugging. Consequently, the work around did not emerge right after the release.

Fortunately, in this case, the issue could be solved or avoided respectively. But it can very well happen that a problem with a beta version cannot be solved and the project gets stuck, the approach has to be changed and time is lost.

7.2 Contributions

APLE is a young research area that tries to integrate agile software development and software product lines. One approach to do so is to manage the commonalities and variability of a software product line using test artifacts in the form of acceptance tests. To support this approach a concept was developed wherein feature modeling can be integrated with acceptance tests in the form of an extension for Visual Studio. This extension, called APLD, was implemented and offers a test-based feature management. Existing tools were investigated and it was decided to base APLD on FMD. APLD offers a graphical acceptance test-based feature modeling as extension integrated into Visual Studio 2010. It allows the mapping of tests in the feature model to real test files in the solution. Acceptance tests can be executed from the model diagram as well as from the configuration tool window and test results are graphically presented in the model diagram and the configuration tool window. Additionally, while creating a configuration, this configuration gets visualized in the model diagram. Both, the feature model and configurations are persisted in the XML format. These files can be imported by other tools.

GP 2010 was extended to offer a public interface for querying and executing tests. APLD uses this interface to query and execute acceptance tests. This interface can be used by any VS2010 extension.

APLD was successfully used to create a test-based feature model for the eHome project. The eHome project is a smart home solution, developed by the ASE Group, that allows the controlling and monitoring of intelligent home systems. Every home has its own floor plan and different hardware capabilities, thus the infrastructure varies from home to home. At the same time they share commonalities. The ASE Group decided to adopt a SPL practice to be able to deliver systems that only encompass the requested variants, without substantial rework. The acceptance tests of the project were written for the FIT Framework, therefore, corresponding GreenPepper tests had to be created in order to use APLD for the feature modeling.

7.3 Future Work

There are problems and limitations that have to be addressed in future work.

An important part of software product lines is the instantiation process. That means a program has to be built according to the configuration, thus the selected features. This instantiated program either includes only the code needed by the selected features or it includes all code, but it is configured to only use the selected features. APLD offers a test-based feature management that allows creating test-based feature models and configurations. Even though these configurations can be saved, it does not support any kind of instantiation process. Ghanam and Maurer developed an approach how the instantiation process can be triggered from a test-based feature model in [5]. He proposes to execute the acceptance tests that are linked to the features of the configuration that has to be instantiated. Code coverage is used to identify all needed classes to build the product. This might be a possible way to achieve product instantiation. As mentioned before APLD stores the feature models as well as configurations in XML files. Another idea is that these XML files could be processed by other programs that do the actual instantiation. However, the structure and content of these files are optimized to reload information and to present it in the diagram and the configuration tool window respectively. It would make sense to introduce a new XML schema that is optimized for the instantiation process.

The usability and the usefulness of APLD have to be evaluated. Although APLD has been used to model the feature of the eHome project, which worked well, there has not been a complete formal evaluation of its usefulness and correctness.

The Constrains relationship was extended with the exclude constraint in the very end of this work. It was no time left to test this relationship. The test model described in chapter 6.3.7 has to be extended in order to test the relationship and its validation.

APLD offers capabilities to collapse and expand tests that are connected to features or to hide tests completely. Nevertheless, diagram space is limited and as feature models can get rather big, the overview can be lost. FMD already offered multi-diagram support with cross-diagram references, but for now APLD offers test mapping only on a per diagram basis. This would be a nice feature to have in the future.

APLD uses GreenPepe 2010 for the access to and execution of GreenPepper acceptance tests. The communication was realized with MEF and a simple interface was created. In the future the same mechanisms can be used to allow the mapping to and the execution of tests from other acceptance testing frameworks or even any kind of tests. This could be achieved by creating additional interfaces or to implement the existing one in other extensions. If this is done, the interface should be moved to a separate project together with all classes that carry the test information and results. Additionally, the naming of the classes should be more abstract and the container classes have to describe tests less specific.

References

[1] **Bass, Len, Clements, Paul and Kazman, Rick.** *Software Architecture in Practice.* s.l. : Addison-Wesley, 2003.

[2] **Ghanam, Yaser and Maurer, Frank.** *An Iterative Model for Agile Product Line Engineering.* s.l. : The SPLC Doctoral Symposium, 2008 - in conjunction with the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, 2008.

[3] **Hanssen, G. and Faegri, T.** *Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering.* s.l. : Journal of Systems and Software (JSS), 2008.

[4] **Carbon, R., et al.** *Integrating product line engineering and agile methods: flexible design up-front vs. incremental design.* s.l. : 1st Internationial Workshop on APLE 2006 - SPLC, 2006.

[5] **Ghanam, Yaser and Maurer, Frank.** *Extreme Product Line Engineering: Managing Variablity & Traceability via Executable Specifications.* s.l. : Agile Conference 2009, Chicago, 2009.

[6] **Ghanam, Yaser, Park, Shelly and Maurer, Frank.** *A Test-Driven Approach to Establishing & Managing Agile Produt Lines.* s.l. : The 5th Software Product Lines Testing Workshop (SPLiT 2008) in conjunction with SPLC 2008, Limerick, Ireland, 2008.

[7] **André, Furtado.** http://featuremodeldsl.codeplex.com/. *Feature Model DSL Homepage.* [Online] 2009. [Cited: January 15, 2010.]

[8] **DSL Tools.** [Online] [Cited: February 16, 2010.] http://code.msdn.microsoft.com/vsvmsdk.

[9] **Lenz, Gunther and Wienands, Christoph.** *Practical software factories in .NET.* s.l. : Apress, 2006.

[10] **Software, GreenPepper.** GreenPepper Software. [Online] Pyxis Technologies. [Cited: January 17, 2010.]
http://www.greenpeppersoftware.com/confluence/display/GPW/Home/.

[11] **Agile Alliance.** Manifesto for Agile Software Development. [Online] [Cited: January 26, 2010.] http://agilemanifesto.org.

[12] **Cockburn, Alistair.** *Agile Software Development: The Cooperative Game.* 2nd Edition. s.l. : Addison-Wesley Professional, 2006.

[13] **Smith, Greg and Sidky, Ahmed.** *Becoming Agile in an imperfect world.* s.l. : Manning, 2009.

[14] **Beck, Kent and Andres, Cynthia.** *Extreme Programming Explained - Embrace Change.* s.l. : Addison-Wesley, 2005.

[15] **Schwaber, K.** *Agile Project Management with Scrum.* s.l. : Microsoft Press, 2004.

[16] **Highsmith, J.** *Adaptive Software Development.* s.l. : Dorset House, 2000.

[17] **Clements, P. and Northop, L.** *Software Product Lines: Practice and Patterns.* s.l. : Addison-Wesley, 2001.

[18] **Pohl, Klaus, Böckle, Günther and van der Linden, Frank.** *Software Product Line Engineering.* s.l. : Springer, 2005.

[19] **Meyer, M. and Lehnard, A.** *The Power of Product Platforms.* New York : Free Press, 1997.

[20] **Kang, K., et al.** *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* s.l. : Technical Report CMU/SEI-90-TR-21, 1990.

[21] **Lee, Kwanwoo, Kang, Kyo C. and Lee, Jaejoon.** *Concepts and Guidelines of Feature Modeling for Product Line Sofware Engineering.* s.l. : Cristina Gacek, editor, Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference, Austin, U.S.A., Apr.15-19, 2002.

[22] **Version One.** [Online] [Cited: January 25, 2010.] http://www.versionone.com/Resources/FeatureEstimation.asp.

[23] **Maurer, Frank and Grigori, Melnik.** *Driving Software Development with Executable Acceptance Tests.* s.l. : Executive Report on Agile Project Management, Vol. 7, No. 11, Cutter Consortium, 2006.

[24] **Myers, Glenford J.** *The art of software testing.* s.l. : John Wiley & Sons, 2004.

[25] **Mugridge, Rick and Ward, Cunningham.** *Fit for Developing Software - Framework for Integrated Tests.* s.l. : Pearson Education, 2005.

[26] **André, Brisette and Beauregard, François.** Build the right sofware! - A white paper on accurate software development. [Online] [Cited: January 17, 2010.] http://www.greenpeppersoftware.com/confluence/download/attachments/5/accurate_developement_wp_en.pdf.

[27] **Microsoft MSDN.** Automation and Extensibility for Visual Studio. *MSDN.* [Online] Microsoft, 2010. [Cited: February 18, 2010.] http://msdn.microsoft.com/en-us/library/xc52cke4%28VS.100%29.aspx.

[28] **van Deursen, A., Klint, P. and Visser, J.** *Domain-Specific Languages: An Annotated Bibliography.* s.l. : ACM SIGPLAN Notices, 35(6):26-36, June 2000, 2000.

[29] **Mernik, Marjan, Heering, Jan and Sloane, Anthony M.** *When and How to Develop Domain-Specific Languages.* 2005.

[30] **Taha, Walid.** *Domain-Specific Languages.* s.l. : Invited Paper ICCES'08, 2008.

[31] **Fowler, Martin.** Domain Specific Language. [Online] [Cited: Janaury 10, 2010.] http://www.martinfowler.com/bliki/DomainSpecificLanguage.html.

[32] **Cook, Steve, et al.** *Domain-Specific Development with Visual Studio DSL Tools.* s.l. : Addison-Wesley Professional, 2007. p. 576.

[33] **Vlissides, John.** *Pattern Hatching - Design Patterns Applied.* s.l. : Addison-Wesley, 1998.

[34] **EMF Feature Model.** [Online] [Cited: January 12, 2010.] http://www.eclipse.org/proposals/feature-model/.

[35] **Feature IDE.** [Online] [Cited: January 12, 2010.] http://fosd.de/fide/.

[36] **Feature Modeling Plug-In.** [Online] [Cited: January 14, 2010.] http://gsd.uwaterloo.ca/projects/fmp-plugin/.

[37] **Feature Model Tool.** [Online] [Cited: January 12, 2010.] http://www.giro.infor.uva.es/FeatureTool.html.

[38] **Hydra Tool.** [Online] [Cited: January 12, 2010.] http://caosd.lcc.uma.es/spl/hydra/download.htm.

[39] **pure::variants.** [Online] [Cited: Janaury 18, 2010.] http://www.pure-systems.com/Variant_Management.49+M54a708de802.0.html.

[40] **XFeature.** [Online] [Cited: Janaury 13, 2010.] http://www.pnp-software.com/XFeature/Home.html.

[41] **Microsoft.** Managed Extensibility Framework Homepage. [Online] 2010. [Cited: January 30, 2010.] http://mef.codeplex.com/.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

Mannheim, 26.02.2010

Felix Riegger