Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3468

Provisioning of Customizable Pattern-based Software Artifacts into Cloud Environments

Andreas Schraitle

Course of Study:	Computer Science
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Christoph Fehling
Commenced:	March 6, 2013
Completed:	September 6, 2013
CR-Classification:	Cloud Computing, Patterns, Automation, Software Product Lines

Abstract

Software architects and engineers frequently face reoccurring problems, when implementing cloud computing applications, leading towards reduced productivity and an increased time to market factor. These issues can be faced by the commonly known concept of patterns. Thus, researchers identified and documented patterns for the cloud computing domain, to preserve gained knowledge about cloud application architectures and service offerings [FLMS11, FLR$^+$12]. These patterns can be used to from the foundation of aggregated cloud computing applications. Dependent on the corresponding cloud service model, such applications require different provisioning steps, which can be performed by individually implemented actions or can be executed by pre-provided cloud services.

Yet, these cloud computing patterns are offered in non-technical, written form, which does not allow to aggregate corresponding implementation binaries to pattern-based applications. Therefore, this thesis combines software product line engineering methods, open source build management tools, and open source infrastructure management tools to implement a software product line for cloud computing patterns, which allows to reduce human-driven efforts to implement aggregated cloud computing applications. This approach enables the possibility to create, aggregate and customize cloud computing pattern implementations; and store them in a so-called pattern template catalogue. Hence, each pattern, stored in such a catalogue, is associated with so-called customization points, which allow to adapt instantiated patterns to individual needs.

To accomplish these challenges, Apache Maven [Mava], an open source build management tool, is extend with means to create, customize and aggregate pattern-based cloud computing applications. Corresponding provisioning tasks are accomplished, by combining PuppetLabs' Puppet [Pup] and pre-offered cloud provisioning services. Pattern-specific customization points are stored within a serialized, so-called variability model, embedded in each pattern. Moreover, the presented structure model allows to decouple direct pattern dependencies through common interfaces, which allows to switch pattern implementations transparently, without adapting dependent patterns. Furthermore, combinable reference patterns are presented and discussed, to provide a proof of concept of the implemented software product line approach.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Introduction

This chapter contains motivation in Section 1.1, for the topics, covered in this thesis. Section 1.2 provides a focus for the discussed problems. Common abbreviation definitions are made in Section 1.3. Finally, contents of subsequent chapters are outlined in Section 1.4.

1.1 Motivation

Cloud computing has evolved to a widely adopted technology by industry as well as science, providing various services on different levels to customers via web-based technologies. These services, combined with their on-demand delivery model, enable customers to fully exploit the benefits of utility-computing: scalability, visualization, pay-per-use and automated provisioning [NIS]. As cloud computing offers new service models to cloud users, these models require software architects, developers, managers and customers to be aware of underlying concepts. The challenge to deal with these concepts, in a unified and efficient fashion, can be achieved by the usage of patterns. The concept of patterns is well known across various engineering fields, thus, software engineering researchers and industry partners have identified and documented patterns for the cloud computing domain [FLMS11, FLR$^+$12]. Due to the on-demand nature of cloud computing, provisioning tasks of pattern-based software artifacts and their run times become more complex, as the newly introduced cloud service models enable dynamic creation and decommissioning of cloud-based services. This system management can be handled by the cloud user itself, using Infrastructure as a Service (IaaS); or it can be shifted to the cloud provider, using Platform as a Service (PaaS). Still, development of software artifacts, the corresponding provisioning and system management tasks involve human interaction and form a major cost factor of cloud-driven projects [VVE10]. These issues can be challenged by means of software product line engineering: reusability of created software artifacts and variability of products - combined with flexible provisioning. Therefore, providing an easy manageable and accessible infrastructure is a key factor within this attempt. Thus, a new field within industry-driven software engineering has been established so-called DevOps [Smi11], focusing on automating the various stages of a software product and interconnecting involved departments, i.e., development, operations and quality assurance. Within DevOps, infrastructure is modeled in code-based documents, to enable provisioning automation for arbitrary amount of nodes.

Combining all of theses approaches in a single solution could speed up the development, provisioning and management of pattern-based cloud computing software artifacts, leading towards a reduced time to market factor for pattern-based products.

1.2 Focus

This thesis focuses to define a method to create, aggregate and customize reference implementations for recently identified cloud computing patterns [FLMS11, FLR$^+$12] by means of software product line engineering. Furthermore, a concept to provision these artifacts into different cloud environments is provided. Implemented reference patterns are stored in a so-called pattern template catalogue, from which they can be instantiated into a developer workspace. Developers, then, may enrich such a template with further code or aggregate it with further patterns. Afterwards, the so created application can be published, as another pattern, into the pattern template catalogue.

The variability of such pattern-based applications is a major key concept to reduce development efforts. Therefore, the idea of a variability model is adopted from the software product line engineering community and is embedded within the pattern structure. This enables developers to address configuration values, relevant for the build and provisioning process and adjust instantiated patterns to their needs.

To accomplish these goals, within this diploma thesis, open source build management and infrastructure management tools, i.e., *Apache Maven* [Mava] and *Puppet Labs' Puppet* [Pup], are extended and combined, to provide the mentioned functionalities.

1.3 Definitions and Conventions

Common definitions and conventions are introduced here, as they are used through the entire diploma thesis document.

Definitions

- The term *provisioning* describes the mechanism to start computation machines within a network and deploy required middleware stacks on top of them.
- The term *deployment* describes the mechanism required to transfer pattern-based binary artifacts into their run time environments.
- The term *undeplyoment* describes the mechanism required to remove pattern-based binary artifacts from their run time environments.
- The term *decommission* describes the mechanism required to remove deployed middleware stacks and suspend related computation machines.
- The term *user* describes an entity, using a specific software functionality, provided by the implemented software product line components.

Conventions
The terms *pattern*, *artifact* and *pattern-based project* are traded equally, as a pattern requires an implementation in the form of an artifact, such an artifact is modeled with code, which is nested in a pattern-based project.

1.4 Outline

This diploma thesis consists of six chapters: *Introduction*, *Fundamentals*, *Related Works*, *Concept and Specification*, *Design and Implementation*, and *Summary and Outlook.*

- *Chapter 1, Introduction* - *Chapter 1* provides a motivation, which introduces the challenges of pattern-based development, combined with means of the DevOps movement. Furthermore, it defines the focus of this thesis. Finally, an outline is provided, which briefly discusses the content and structure of this document in detail.
- *Chapter 2, Fundamentals* - *Chapter 2* provides an introduction in related scientific fields, including: *Cloud Computing*, *Software Product Line Engineering* and *Patterns*; to establish a common, scientific background, which forms the foundation for further chapters of this diploma thesis.
- *Chapter 3, Related Works* - *Chapter 3* discusses related, scientific works as well as established open source tools, related in the field of build management and infrastructure management. *Apache Maven* and *Puppet Labs' Puppet*, are presented as they are used in advanced chapters. Also, the concept of so-called *variability descriptors*, which model possible customization points of patterns, is introduced within this chapter.
- *Chapter 4, Concept and Specification* - First, *Chapter 4* discusses the superordinate context of this thesis; a software product line for cloud computing patterns. Then, functional, as well as non-functional requirements, are defined, for the later on implemented approach, to support the creation of reusable, customizable and combinable pattern-based artifacts. Thus, the human-driven cloud computing pattern development process is introduced, which allows pattern developers to implement such artifacts. Afterwards, the corresponding functional architecture is represented in combination with required variability and project structure models. Finally, the functional components, which implements these features, are presented.
- *Chapter 5, Design and Implementation* - *Chapter 5* first introduces an abstract reference architecture, which is obligatory for all implemented components of this thesis. Then, design decisions, relevant for the implemented approach of this thesis, are presented. Including: the implemented event propagation mechanism, corresponding handler implementations and the xml implementation of the specified variability meta-model. Afterwards, technologies and tools, used during the implementation, are presented. Finally, implemented reference patterns, i.e., the *Three-Tier Cloud Application Pattern* and the *Message-Oriented Middleware Pattern*, are presented.
- *Chapter 6, Summary and Outlook* - Chapter 6 summarizes the results of this thesis and provides an outlook of possible, further work.

Fundamentals

This chapter includes related scientific fields of research and introduces their major topics to establish a common foundation for upcoming chapters. Thus, the terms *cloud computing*, *software product line engineering* and *pattern* are covered in the following.

2.1 Cloud Computing

In this section, the term *cloud computing* is defined, using the established *NIST-Definition*. During this definition, the multi-tenancy term is briefly discussed to point out the various scenarios of resource sharing. Furthermore, so-called *cloud service models* and *deployment models* are introduced. Finally, a distinction to other, related technologies is provided.

2.1.1 Essential Characteristics

A cloud, according to the NIST-Definition [NIS], provides the following characteristics:

- **On-demand self-service** - Indicates that cloud service users are able to provision and decommission resources on-demand, in an automated fashion.
- **Broad network access** - Means that provided services are offered via network-based technologies and can be accessed by any web-capable device.
- **Resource pooling** - Large amount of resources are shared between cloud users, which allows cloud providers to exploit economies of scale and utilize load variations. Thus, virtual resources, hosted on commonly shared physical hardware, are offered.
- **Rapid elasticity** - Resources can be acquired and released in an unlimited fashion, using web-technologies. Thus, consumed resources can be horizontally scaled on demand.
- **Measured service** - Consumed resources are billed according to the offered payment-model, e.g., pay-per-use or temporary acquisition. This model is commonly based on the type of resource. A queue service, e.g, can be charged per delivered message or on a monthly base.

2.1.2 Multi-Tenancy

Resources, located in a cloud, can be shared by multiple so-called *tenants*, which allows the cloud provider to leverage economies of scale more efficient. This characteristic forms a major property of cloud services. Thus, the term *tenant* and its corresponding term *multi-tenancy* is defined in the following.

Definition: A *tenant* is an entity, which consumes a cloud service.

This generic definition allows to cover single persons, as well as groups or companies.

Definition: *Multi-Tenancy* depicts that cloud resources are shared between several tenants.

Thus, a single tenant has not to be aware of other tenants, using the same resource, when interacting with a resource [1].

2.1.3 Cloud Service Models

Offered cloud services are categorized by their so-called *cloud service model.* This model describes features, as well as properties of the offered services. These service models are introduced in the following paragraphs.

Cloud Service Model Types:

- **Infrastructure as a Service (IaaS)** - IaaS offers cloud users the possibility to provision a virtual machine on-demand, offering specified Service Level Agreements (SLAs). The specified SLAs are ensured by the cloud provider, as well as isolation of tenants. Such an acquired machine is only bound to a specific location when it's running. Therefore, rebooting the machine may result in a different location within the used cloud. Depending on the cloud provider, customers can specify a geographic region, where the created machine is hosted. This guarantees an upper communication bound, as *round trip times* (RTT) vary between instances. Yet, scaling hosted resources is liable to the cloud user himself. Well known IaaS clouds are: Amazon EC2 [EC2], Rackspace Cloud Services [Rac] and VMWare vCloud [VCl].
- **Platform as a Service (PaaS)** - PaaS offers customers a platform model to host their applications in an managed fashion. Thus, it can be distinguished in two sub-models:
  - **aPaaS** - This PaaS model focuses on hosting applications, offering scalable run time environments and various middleware services, like: queues, data stores and integration services. Well known PaaS are, e.g., *Google App Engine* [App], *Salesforce Herkoku* [Her], *Salesforce Force* [For] and *Amazon Elastic Beanstalk* [Ela].

[1]Detailed information about patterns, implementing multi-tenancy, can be found in [KM08] and [MUTL09]

  - **iPaaS [Pez11]** - This PaaS model is focused on integration tasks, offering scalable integration services, which establish communication with external applications. The iPaaS model is still evolving, thus, products vary in provided service types. FuseSource's *Fuse Fabric* [Fus], e.g., aims to provide an environment to automatically provision, configure and load balance requests on nodes, with open source integration software, like *Apache ActiveMQ* [Act] and *Apache Service Mix* [Ser]. In contrast, *CloudHub* [Clo] provides pre-build adapter services, running in a managed environment to connect to existing applications like ERP systems. In both models, the iPaaS acts as virtual intermediary to establish communication between applications. Yet, in case of CloudHub, the responsibility of hosting the iPaaS middleware lies on the iPaaS provider side, while in case of FuseFabric, its located at the consumer side.

- **Software as a Service (SaaS)** - SaaS offers pre-build, customizable software solutions, hosted in the cloud. SaaS customers subscribe to a service and pay the usage, depending on the negotiated payment model. The SaaS provider guarantees SLAs, isolates tenants and scales the application in a transparent manner. Known examples for SaaS are: GitHub [Git], SalesForce [Sal] and Google Docs [Goo].

Figure 2.1 illustrates the introduced cloud service models. Typically PaaS and SaaS offerings are implemented by extending service models beneath them. Therefore, system management tasks of sub-ordinate service layers are shielded from the service user.

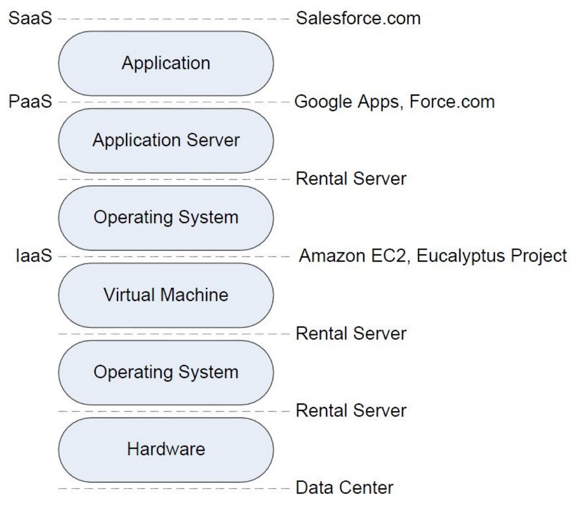


Figure 2.1: Overview - Cloud Service Models. [Feh09].

Each of the above presented layers offer different benefits to users. Hence, aggregated functionality of higher services is often established using services of previously existed layers. *Elastic Beanstalk* [Ela], e.g., is Amazon's aPaaS product, which is build on top of Amazon's IaaS [EC2] offering, combined with other Amazon products.

2.1.4 Deployment Models

Deployment models describe the way clouds are structured. This structuring has impact on the degree cloud users have to be aware of underlying resource sharing concepts, concerning potential resource access of other tenants.

Each of the following models provide different resource sharing semantics:

- **Private Cloud** - A private cloud is isolated from other tenants; the underlying resource pool can only be accessed by a single tenant. This isolation can be established by different means, e.g., the virtual machines can be isolated into virtual networks, by filtering access requests on different OSI layers [OSI]. Also, physical machines can be hosted in an isolated region of the cloud provider, to establish a physical network separation. Depending on the selected alternative, this model may be limited by the amount of machines. This may result in a resource bottleneck, when long running load peaks are expired.
- **Community Cloud** - This deployment model allows multiple tenants to share a common resource pool, therefore the amount of tenants is limited. This constellation is often the case when large amount of resources are required for a short time and the total cost of ownership is not efficient enough for a single tenant. Hence, the costs of ownership are split among the participating tenants.
- **Public Cloud** - A public cloud allows accessing its resources to a wide amount of tenants. Thus, tenants share the underlying resources in an unlimited fashion. The isolation of these tenants lies in the responsibility of the cloud provider. Such a shared resource pool is a potential security risk and may violate compliance agreements. Thus, this model requires consumers to be aware of these issues.
- **Hybrid Cloud** - A hybrid cloud is an interconnected cloud, consisting of any deployment models mentioned before. These hybrids enable customers to exploit the benefits provided by each used cloud deployment model and still fulfill legal terms. A common practice is, to host sensible resources within a private cloud, while hosting none-sensible resources in a public cloud. This allows to focus on-premise resources to secure areas and exploit benefits of public clouds.

Note: A cloud is always described by both factors: service model and deployment model; to describe its nature, since arbitrary combination of them are possible.

2.1.5 Distinction to Related Technologies

Cloud computing is a product of different, preceding technologies and approaches. Thus, this subsection covers related technologies and their distinction to cloud computing.

- **Grid Computing** - Grid Computing bundles computation centered resources, provided by several entities into a virtual resource. These resources can be used by contributing entities on a scheduled base, enabling them to gain more computation resources for a short period of time. Grid computing, in contrast to cloud computing, focuses on acquiring contributors to expand the virtual resource, while cloud computing focuses on providing independent resources in a pooled-fashion to exploit economies of scale.
- **OnPremise Resources** - This model assumes a static data center, where resources seldomly move from their physical position. Due to its nature, resources can not be provisioned or decommissioned on a on-demand self-service base. Thus, resources are paid on a subscription base. These properties distinguish on premise resources from cloud provided ones, which implement the previously defined characteristics, covered in Subsection 2.1.1.
- **Web Services and Service Oriented Architecture (SOA)** - Cloud offerings are often accessible via Web Service Technologies, like *REST* and *SOAP* based web services. Thus, these services can be embedded in service oriented architectures. Yet, they only encapsulate the functionalities of the underlying cloud offerings. Thus, these functionalities can often be accessed by other technologies as well, depending on the API of the cloud provider.

Required cloud computing terms have been briefly introduced. More information and possible solutions to occurring issues can be found in the book *Cloud Computing Patterns* [FLR+12].

2.2 Software Product Line Engineering

This chapter introduces a sub-category of software engineering: *Software Product Line Engineering.* Key concepts within this field are discussed, since a major topic of this thesis is to implement a software product line for cloud computing patterns, which is capable of creating reusable software artifacts.

Product Line Engineering is a known concept in various engineering fields, it focuses on creating reusable artifacts in a unified manner. These artifacts can be used to create new products, in a more efficient way, since development costs are shared and, thus, are reduced per product. This approach has been adopted by various industries, like aerospace and automotive industries.

Cars, e.g, within a series share major components, like: *engines*, *chassis*, *gears* etc.. Hence, manufacturers are able to produce larger amounts of these components and exploit economies of scale, while still meeting the customer specific requirements and providing several solutions to the market. These concepts can also be used in the field of software engineering. Thus, *software product line engineering* is introduced in the following subsections.

2.2.1 Terminology

Definition: "A *software product line* is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets." [2] [Spl]

Definition: A *feature* represents a certain amount of functional respectively non-functional requirements of the product to be built. [3]

Features are implemented by so-called *system artifacts.* These artifacts occur on various phases. Thus, the term *system artifact* is represented in greater detail.

Definition: A *system artifact* implements a certain number of features, encapsulates them and offers them in a unified manner.

Such system artifacts are offered to software engineers in a so-called *catalogues*, from where they may be instantiated, to become part of new products.

Definition: A *product space* consists of all possible products, that can be created by aggregating arbitrary system artifacts of the related software product line.

So-called *archetypes* provide the foundation of reusable system artifacts. They form a unified reference, how artifacts are structured and provide means to be instantiated. Thus, they preserve previously gained knowledge. Each produced system artifact is created from such an archetype and may be used as an archetype again, for further system artifacts.

The term archetype relates with the well known term *stereotype.* Yet, a stereotype is class-bound, while the term archetype is instance-bound. This means that an archetype is instantiated, while a stereotype only classifies a set of existing instances. Therefore, the Oxford English Dictionary's definition states, that an archetype is "*a very typical example of a certain person or thing.*" [Oxf] This definition implies that an archetype provides implementations for various patterns. Yet, the definition does not fit the terminology of software product line engineering, thus a different definition is introduced.

Definition: An *archetype* preserves gained knowledge about previously realized features, it provides methods to instantiate reusable system artifacts from it.

To enable a unified creation process, which allows to create reusable artifacts, a generic *reference architecture* is mandatory for all system artifacts, created by the software product line. This common reference architecture guarantees that the structure of all created artifacts is equal and that parts of such an created artifact can be exchanged easily, as the different layers

[2]The *particular market*, mentioned in the above definition, is referred to as *domain.* The related engineering process inside Software Product Line Engineering is referred to as *Domain Engineering.* Combined with the *Application Engineering*, these two processes form the foundation of the so-called *Software Product Line Process* and are discussed later on in Subsection 2.2.3

[3]Functional requirements represent use cases and communication semantics, since they describe the way the system interacts with external entities, while non-functional requirements, like reliability, scalability, response times, throughput etc., have no influence on the correctness of the system, but describe conditions which have to be met to guarantee an *up and running semantic* for various, possible conditions.

are decoupled through unified interfaces. Such a set of reference architectures is combined within the so-called *productline architecture*.

Definition: A *productline architecture* covers a set of reference architectures, mandatory for all developed system artifacts. It contains common features for functional as well as non-functional requirements for the complete engineering cycle.

The offered features, of such an architecture, are expressed via *variability points* [4], which need to be bound when the reference architecture is instantiated.

Figure 2.2 offers an overview of the introduced terms and points out their relations within the *software product line process* to create a new product out of existing archetypes.

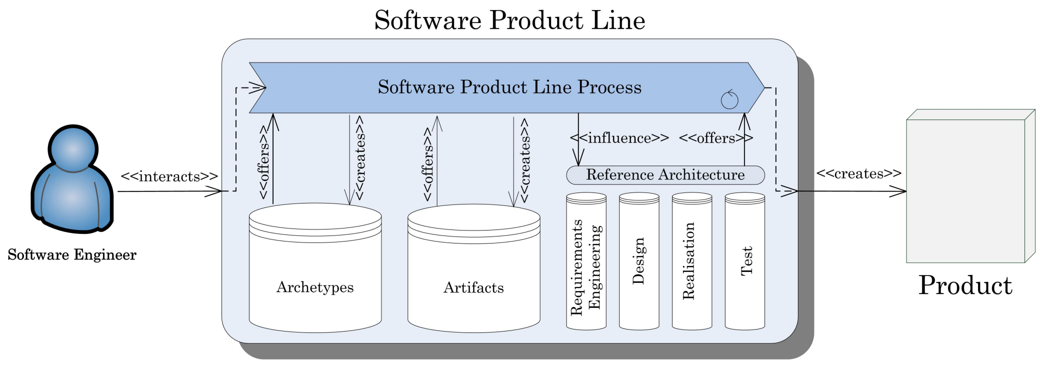


Figure 2.2: Overview - The Software Product Line Methodology.

As a result of this process, software product line engineering offers various benefits, which are discussed in the following.

From a software engineers point of view:

- *Reusability* - Enabled through unification and combinable *artifacts*.
- *Stable core features* - Since testing efforts and the achieved quality increase is shared.
- *Increased development speed* - Reusability enables software architects and developers to focus on their project-specific business requirements.

From a managers point of view:

- *Cost reduction* - As a result of reduced development time.
- *Increased competitiveness* - Due to *cost* and *time to market* reduction.
- *Potentially increased customer confidence* - As quality may increase, while costs and delivery times decrease.

[4]Variability points form a major concept of software product line engineering and are thus covered in detail in Subsection 2.2.2.

From a customers point of view:

- *Reduced total cost of ownership* - Since maintenance as well as acquisition costs decrease.

These benefits, along with corresponding expectations of users, are illustrated in Figure 2.3.

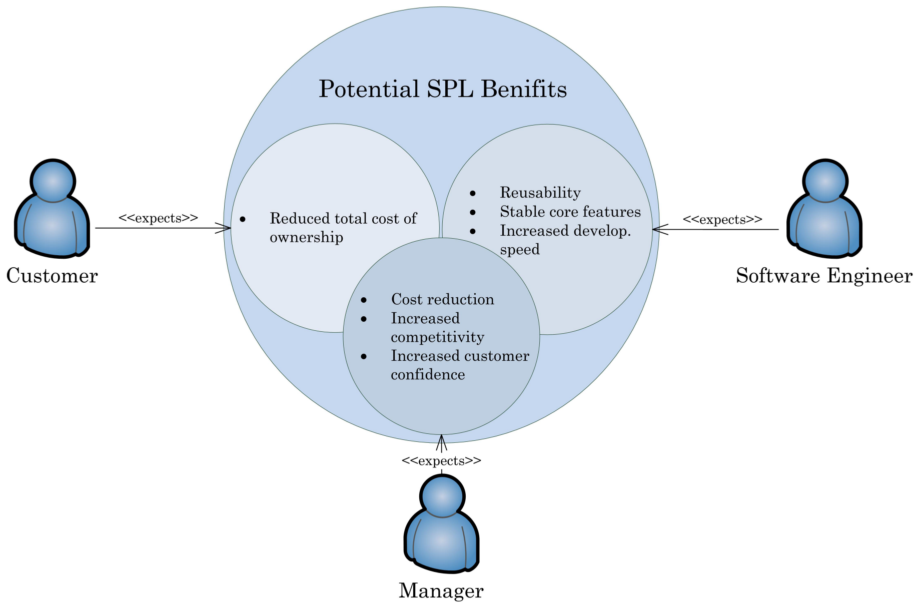


Figure 2.3: Overview - Benefits of Software Product Line Engineering.

2.2.2 Product Variability

Variability is captured in so-called *variability models*, which enable flexible product spaces and allows to adapt them to individual needs. Such variability models cover different kinds of *variability types*. These variability types, are discussed in the following paragraphs.

Product Space Variability
Variability, occurring on a product-base, describes that used artifacts may vary. Four major dependency types [GBS01] can be identified and are described in the following:

- *Mandatory* - Indicates that this artifact is required in any case.
- *Optional* - States that this artifact is an optional part of the product.
- *Variant* - Indicates that this artifact is part of a set of artifacts, from which exactly one has to be chosen. Thus, the relation is often expressed with a *xor-operation*.
- *External* - This artifact is provided by an external source. Thus, it is not required to be built. Yet, it has to be present when the product is running.

Artifact Variability
Artifacts, used to build a product, offer customization points to adapt the artifact during the instantiation phase. These so-called *variability points* are discussed in the following.

Definition: *Variability points* occur on the archetype and architecture level and influence the delivered features of the product.

These variability points need to be *bound*, to instantiate a concrete system artifact. The so customized artifact then becomes part of the final product.

The binding time [RH06] of such a variability point can be done at:

- *Design-time* - Set by a designer during the system specification.
- *Compile-time* - Set during the compilation of the source code.
- *Development-time* - Set by a developer during the implementation.
- *Deployment-time* - Set during the deployment process by the deployment engine.
- *Run-time* - Set when the artifact is running within its run time.

2.2.3 The Software Product Line Process

After introducing the mandatory terms of software product line engineering, the process of creating artifacts and products is introduced; *the software product line process.* The process itself consists of two interrelated sub-processes: *domain engineering* and *application engineering*; which implement an abstract engineering life-cycle that creates reusable artifacts.

Domain Engineering focuses on creating common reusable artifacts within the product space. Thus, artifacts created within this cycle preserve domain specific knowledge and form a platform for further artifacts or products. This platform consists of code-based as well as diagram-based artifacts to support the full application engineering life-cycle.

Application Engineering focuses on creating new products with domain engineering generated artifacts as well as artifacts from previously developed products. Generated artifacts, identified for common usage then may move back to the domain specific catalogue.

The passed phases within an abstract cycle are [5]:

- *Requirements Engineering* - This phase generates artifacts that contain specific features, which implement functional requirements. Such artifacts are commonly document-based, since they model logic functionality as functional components.
- *Design* - The design phase generates document-based design artifacts, which encapsulate related functional and non-functional requirements.

[5]These phases are well known, since they form the foundation of all software engineering methodologies.

- *Realization* - During the realization phase, generated artifacts implement common business requirements. Usually these artifacts are code-based.
- *Testing* - Artifacts, generated within this phase, are used to establish a unified testing approach. Such an artifact, e.g., may be a tool, capable of generating load test scenarios; a generic test scenario or an extendable test suite.

Furthermore, the domain engineering cycle extends the ones stated above and adds:

- *Product Management* - This phase focuses on making artifact centered decisions about resource distribution, time schedules, product-variant planning and project coordination[6].

Figure 2.4 illustrates the above introduced terms and their relations.

Each artifact, generated during one of the cycles mentioned above, is instantiated by using: its defined reference architecture, to meet non-functional requirements; its corresponding archetype, to meet structure agreements and comply reusability; and its variability point bindings, to select concrete features.

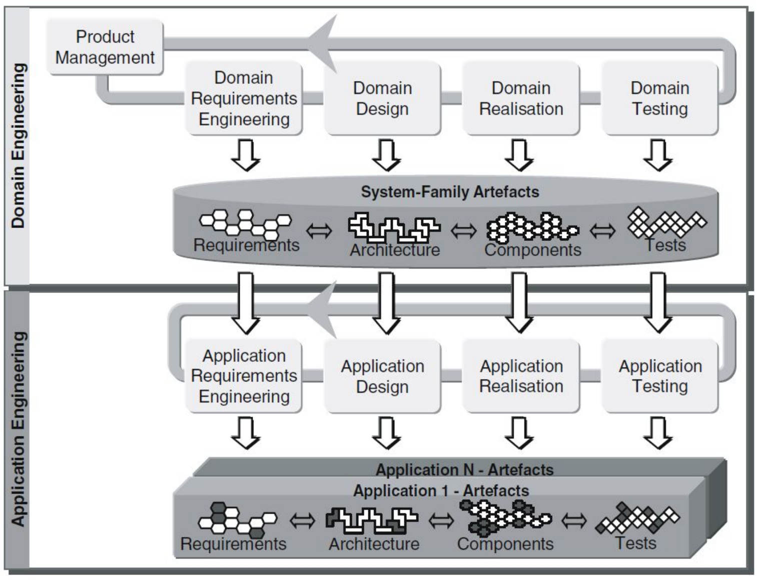


Figure 2.4: Overview - The Software Product Line Process and its Sub-processes [LSR07].

[6]**Note:** Since various definitions exist for the software product line process, the management phase is often not only bound to the domain engineering cycle, but coordinates the complete process, including the application engineering cycle.

2.3 Patterns

Patterns are a well known and widely accepted phenomenon, since they occur widely in aggregated entities in nature and - as a result - in science. Such aggregated entities may be models, describing reality, like: biological models, physical models, behavior models or construction models. Even in models themselves such patterns may occur. Thus, the term is discussed and different pattern domains are shortly presented. The concept of using such patterns to solve common, reoccurring problems has been adopted by various engineering fields such as: *Architectural Engineering*, *Electrical Engineering*, *Mechanical Engineering*, *Automotive Engineering*, *Aviation Engineering* and *Software Engineering* as well [7]. These patterns are often stored in so-called *catalogues* to document knowledge, preserved by them, and offer a non-technical platform for solving problems in the related domain. Furthermore, patterns can be interrelated and composable, which may result in forming a pattern again. Thus, documenting such patterns is often done with an approach to document this pattern structure as well. The inner structure of such interrelations are modeled by a *pattern meta-model.* These interrelations form a so-called *pattern-language*, where a single pattern can be interpreted as a word, while the interrelations form the grammar [8].

To establish a common sense of the nature of patterns in software engineering, various domains are discussed in the upcoming subsections.

2.3.1 Christopher Alexander's Architecture Patterns

Within the field of engineering, the term pattern, as well as the corresponding term pattern-language, have first been identified and defined by the architect Christoph Alexander in his books: *The Timeless Way of Building* [Ale79] and *A Pattern Language: Towns, Buildings, Construction* [AIS77]. Alexander introduces pattern-based concepts in *The Timeless Way of Building* by example and discusses design theory from a pattern-based point of view. Additionally the concept of "the quality without a name" is discussed. This property describes the obvious observable quality of a being, which, yet, can not be named. Even the way Alexander discusses these things underlines his intention of restructured thinking of architecture, since the structure of the book itself is a pattern, allowing the reader to "read the whole chapter in a couple of minutes, simply by reading the headlines"[Ale79]. This outlines a property of patterns; to occur on macroscopic as well as microscopic levels.

Alexander's pattern definition
"*A pattern is a careful description of a perennial solution to a recurring problem within a building context, describing one of the configurations that brings life to a building. Each pattern describes a problem that occurs over and over again in our environment, and then describes*

[7]**Note:** These are only some engineering fields, as the term pattern is of generic nature. Thus, it can be used in nearly any engineering or scientific field.

[8]**Note:** A possible implementation of such a pattern langugage meta-model can be found in [Gri11], which allows to define pattern-based languages.

the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice." [AIS77]

This definition was done in a *building context*, but it is not bound to it, since the term *building* could be easily replaced by the term *software* or any other domain-describing term, without loosing correctness. Because of that, Alexander's books became widely accepted by various domain experts, adopting his ideas for their domains.

Alexander's second book, *A Pattern Language: Towns, Buildings, Construction*; provides patterns by example for architecture, building and planning of communities. Within the book he discusses patterns on various levels. Again, he uses a pattern-based approach to document the identified patterns.

Each pattern definition consists of the following, ordered sections:

1. *Introduction* - The intention of the pattern is briefly pointed out. Then, a picture is presented to get familiar with the context. After that, the impacts of the issue are shortly discussed.
2. *Solution* - A solution, fitting the introduced issue, is presented to the reader, including a sketch, which presents the used components.
3. *Discussion* - This part is used to discuss the impacts of the originated problem constellation as well as the corresponding solution itself, including: observed effects, historical origins and alternatives. It is designed to be skipped by the reader, thus it is placed between two region symbols, build of three asterisks (*****).
4. *Related Patterns* - At the end, interrelated patterns are listed as links.

These links allows the reader to descent or ascent from the graph based hierarchy. A *night life square*, e.g., designed for nightly activities may consist of a cluster of *night spots*; shops, offering different services; and *benches*, allowing people to rest. Each of these artifacts are again *patterns*, described in the book.

Alexander's third book, *The Oregon Experiment* [Ale78], documents the usage of his previously introduced design theory and his architectural patterns. The book describes the implementation of some of those patterns and documents the master plan of the University of Oregon. It again guides the reader with illustrations of the solutions and discusses six major design principles: *patterns*, *participation*, *organic order*, *diagnosis*, *piecemeal growth*, and *coordination.*

In the field of software engineering, different domains exist, where analogous patterns occur. Thus, related domains are introduced briefly in the following.

2.3.2 Software Engineering Patterns

Software engineering focuses on the process how software is created in a guided manner, by executing various phases, i.e., *requirements engineering, design*, *realization*, *test* and *maintenance.* Each phase requires abstract tasks to be executed with concrete methods and technologies. Within this execution of phases, reoccurring issues and solutions can be documented by patterns. These patterns offer software engineers the possibility to speed up development and increase quality, since they provide means for unified solutions in prevailed fashion and form the basis for efficient communication within projects. Hence, patterns have been identified for the phases stated above, technologies and project management strategies.

Note: This thesis - especially the implementation - is strongly based on patterns, thus, major accepted pattern collections are briefly discussed.

Gang of Four Patterns
In object-oriented design, the term *design pattern* is often recognized with the patterns identified by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides - forming the so-called *Gang of Four* - and their book *Design Patterns Elements of Reusable Object-Oriented Software* [GHJV95]. These patters focus on the design of reusable software modules and provide detailed ideas how the implementation of software modules can be made flexible, decoupled and structured in an efficient manner. Moreover, the patterns are designed at an abstract level, independent of concrete programming languages, to be applicable on any object-oriented programming language. Although Gamma et.al. demonstrate their approach with C++ implementations, these patterns can also be adopted to other object-oriented languages, like Java [Java].

The Gang of Four patterns are divided in three categories:

- *Creational Patterns* - These patterns focus on decoupling the dependencies of objects from concrete classes, which implement the creation of the requested object.
- *Structural Patterns* - Structural patterns focus on simple and logical structures within the overall system. They provide means of dependency reduction, to build large objects.
- *Behavioral Patterns* - Are used to hide algorithms or state. They simplify event handling and offer methods to iterate and modify collections in a efficient fashion.

In UML-based design documents, these patterns can be found in component diagrams and sequence diagrams, since they document structure and behavior of a system.

Enterprise Application Integration Patterns
Real systems often face the challenge to interact with various legacy systems, using different programming languages, data models and communication models. The challenge to interconnect such systems, without establishing a tight coupling can be faced with so-called *Enterprise Application Integration Patterns* (EAI-Patterns). These patterns occur on the software architecture level and enable architects to choose from existing solutions. The main objective of EAI-Patterns is to provide *loose-coupling*, manifesting in decoupling of space or/and time, meaning that a system communicating with such patterns, has not to be aware of the amount of potential message receivers nor the concrete time, when the message is consumed by a receiver.

Such patterns have been identified by Gregor Hophe et al. in *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* [HW03]. These patterns provide ideas for structuring components in such messaging solutions, transform messages and route them to their destination, while not depending on concrete technologies.

Quasar 3.0 Patterns

Within companies, divisions, departments and even teams problem constellations are faced several times. Thus, companies have identified patterns, capable of solving these reoccurring issues. While the previously discussed patterns mainly focus on technical issues, such patterns focus on guiding decision making in the software development process. These patterns are categorized according to the software development phase they occur in.

Inside Quasar 3.0 [EKN+12], the overall software engineering framework of CapGemini sd&m, these categories are:

- *Requirements Engineering* - System Requirements are specified and modeled according to the needs of the customer. Results of this phase form the foundation of later on created documents.
- *Analysis & Design* - Functional as well as technical components are identified, designed and modeled. The analysis documents provide a generic solution to the problem, independent of the later on used technologies. In contrast, the design documents adapt the analysis models, by adding technical relevant components as well as specific, technical knowledge.
- *Development* - The pre-identified components are implemented, tested and integrated in the overall business context.
- *Software Configuration Management* - Strategies for managing change requests, build as well as release plans are nested in this phase, which provide foundations for technical as well as operational decision making.
- *Analytics* - Strategies for testing, compliance and quality measurements are defined within this domain, to increase the quality of the overall system. This, especially, includes integration testing scenarios as well as measurements of code quality, to predict quality changes.

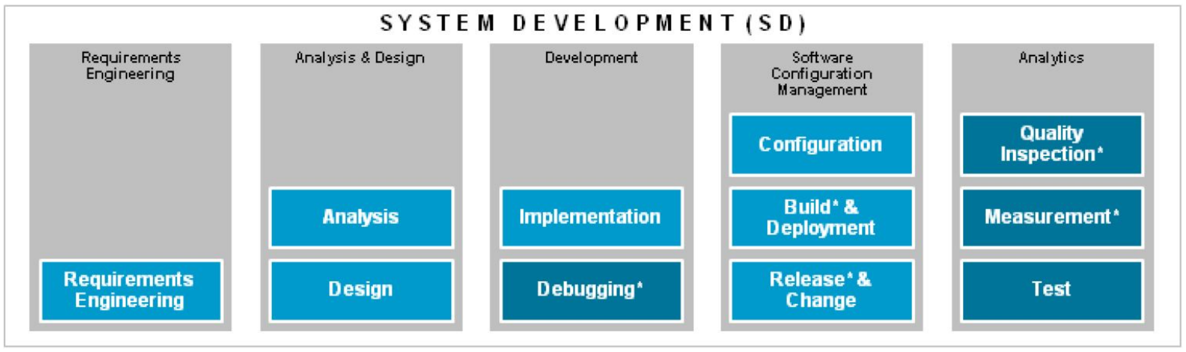


Figure 2.5: Overview - The CapGemini sd&m Quasar 3.0 Domain Model [EKN+12].

Related Works

This chapter presents scientific as well as industrial related works, located in the field of *patterns*, *cloud computing* and automated infrastructure provisioning.

First, Section 3.1, presents recently identified *cloud computing patterns* and the corresponding *pattern catalogue*, since they form the foundation of later implemented software artifacts. Afterwards, Section 3.2, introduces a related framework for creating composable applications - *Cafe* - which is discussed, along with its underlying variability model. Then, Section 3.3 presents the build automation tool *Apache Maven*. Finally, Section 3.4 presents Puppetlabs' *Puppet* to illustrate methods provided by recently developed *infrastructure as code* projects, located within the DevOps movement.

3.1 Cloud Computing Patterns

Patterns in general have been discussed in Section 2.3, to establish a scientific background about patterns, within this thesis. Later on implemented pattern-based artifacts are based on the here discussed work of Fehling et al. [FEL+12, FLMS11, FLR+11, FLRS12] and, as conclusion of those previous works, the book *Cloud Computing Patterns* [FLR+12].

3.1.1 Intention

Since cloud computing patterns are patterns in the sense of Alexander's definition, introduced in Section 2.3.1, they provide a unified structure to document captured knowledge and, furthermore, offer solutions to reoccurring challenges within the cloud application life-cycle. Fehling adopted these ideas for the cloud computing domain during his PhD studies and published a pattern-based book. Within his book he documents the complete cloud computing domain by the means of patterns; also including *cloud service levels* and *multi-tenancy* - discussed in Fundamentals Section 2.1. This enables various persons: students, developers, architects, managers and customers; to read the pattern book within a short period of time, offering them the possibility to solve their requirements. This idea is based on the structure of Alexander's pattern books and had been adopted to the cloud computing domain.

Note: Since only a few of the documented patterns are implemented within this thesis, the captured categories and their nested patterns are only discussed briefly, detailed information can be found in the book *Cloud Computing Patterns* [FLR+12].

3.1.2 Structure of a Pattern

Each pattern is documented in a unified fashion, to guarantee compatibility between the identified patterns and capture the knowledge of the underlying solution. The used structure splits the problem to be solved into several topics, to discuss its complexity from various point of views and provide possible solutions. Moreover relations to other patterns are provided. These patterns can be used to solve the problem or they solve similar issues.

The structure of a pattern is stated as follows:

- *Pattern Name* - Each pattern is assigned a unique name to identify it.
- *Intent* - The intent briefly outlines the purpose of the pattern and the embedded solution.
- *Driving Question* - The driving question enables readers to quickly identify whether the pattern fit to their current needs, since it describes the core problem and the corresponding solution of the pattern.
- *Icon* - The icon provides a visual representation of the pattern. It can be used to be combined with other patterns graphically, forming architecture sketches. Furthermore, it identifies the pattern visually. Since human comprehension is strongly based on visual impression, it enables readers to remember the pattern in a more effective way.
- *Context* - The context provides detailed information about the environment, to which the pattern can be applied to. It provides a description of the initial situation, including the *cloud deployment models*, *cloud service models* and *cloud offerings*.
- *Solution* - The solution section provides an answer to the driving question and points out how the initial problem can be solved in an abstract way, thus, an architectural sketch is provided.
- *Result* - The solution, provided by the architectural sketch, is discussed with great detail afterwards. Possible impacts on related patterns are outlined as well.
- *Variations* - The variations section focuses on describing variations of the reviewed pattern and discusses the differences between them. Yet, these variations belong to the identified pattern and do not form a new one.
- *Related Patterns* - This section is responsible for forming the so-called *pattern language*, since it covers related patterns and, thus, outlines interrelations between them. The discussed patterns may have similar initial problems, which could be used to solve the driving question or they exclude each other.
- *Known Uses* - Products and applications, using the discussed pattern, are discussed here.

3.1.3 Cloud Computing Pattern Catalogue

Since a cloud computing application consists of a specific selection, from a set of various models, like: *service models*, *computation models*, *storage models*, *communication models*, *deployment models*, etc.; each identified pattern has been assigned to one of these categories, to provide a categorization within the cloud computing pattern space. These categories and their nested patterns form a so-called pattern catalogue. Such a catalogue, within the cloud computing domain, has been identified by Fehling et al. [FLMS11] and forms the foundation of the book of Fehling [FLR$^+$12], on which this thesis is based on.

Thus, the identified categories and their sub-categories are discussed briefly:

- *Cloud Computing Fundamentals*
  - *Application Workloads*: These patterns describe the different workloads, applications may be exposed to and outline possible handling strategies.
  - *Cloud Service Models*: These service models are presented according to the NIST definition [NIS] and have also been discussed in Subsection 2.1.3.
  - *Cloud Deployment Models*: Likewise *cloud service models*, *cloud deployment models* are based on the NIST definition [NIS] - they can also be found in Subsection 2.1.4.
- *Cloud Offering Patterns*
  - *Cloud Environments*: These patterns describe the functionality and semantics of cloud environments, like: *elastic infrastructure* and *elastic platform*; and the possible, corresponding availabilities: *node-based availability* and *environment-based availability*.
  - *Processing Offerings*: This pattern-map consists of: The *execution environment* pattern, which could be an application server or a similar application container; The *map reduce* pattern, a computation paradigm, which offers the possibility to slice data into chunks and compute them in parallel; and the *hypervisor* pattern, responsible for creating virtual environments.
  - *Storage Offerings*: These Pattern describe the way data can be stored and retrieved in the cloud. This sub-category covers different storage solutions like: *block storage*, *blob storage* and *relational database*, as well as the corresponding consistency semantics.
  - *Communication Offerings*: Within this sub-category different communication semantics - known from distributed systems - are defined, since they are widely used within cloud computing. The discussed patterns focus on message delivery - using different delivery semantics - as well as, required *virtual network* capabilities. Furthermore, *message oriented middleware* - acting as message delivery provider - is presented.

- *Cloud Application Architecture Patterns*
  - *Fundamental Cloud Architectures*: This sub-category covers fundamental architecture styles, to enable easy composable and distribution-transparent applications. Thus, the patterns: *loose coupling* and *distributed application*; are presented.
  - *Cloud Application Components*: This sub-category covers the patterns required to build an application, it identifies major component types like: *stateless component*, *stateful component*, *multi component*, *user interface component*, *data access component* and the *processing components*; which are required to build a scalable and layered application that is accessing stored data.
  - *Multi-Tenancy*: These patterns cover the possibilities of implementing multi-tenancy. The term *multi-tenancy* has been briefly discussed in Subsection 2.1.2 - further literature references can be found there.
  - *Cloud-Integration*: Cloud-Integration patterns focus on integrating various applications - running in the cloud or on-premise. Thus, they provide means to bridge application boundaries, with defined semantics.
- *Cloud Application Management Patterns*
  - *Management Components*: These patterns aim to provide: abstraction of provider specific APIs; configure components, to guarantee the loose coupling and the stateless component paradigms; enable application elasticity; and observation of deployed functionality, by the means of the *watchdog* pattern.
  - *Management Processes*: This sub-category provides processes dealing with: updating components, under different circumstances; recovery of failed components, while the system is running; adjusting functionality, depending on the current load; suspending resources, in a hibernate fashion to safe costs; and the possibility to scale the application in a managed fashion.
- *Composite Cloud Application Patterns*
  - *Native Cloud Applications*: These kind of applications run completely in the cloud. Thus, they do not have dependencies to on-premise applications. They can be created in different architecture styles - three-tier or two-tier layered - resulting in different scaling behavior and in different benefits and drawbacks, depending on the used environment. Such applications may also use transparent, distributed caches - often referred as *content distribution network* - to speed up data retrieval and thus, reduce the overall computation time.
  - *Hybrid Cloud Applications*: Hybrid Cloud Applications are composed of: components running in the cloud, as well as components running in a static environment. However, core components are kept within the static run time environment. Cloud resources are only requested to provide support, when components face workloads, which exceed their local capabilities. These additional cloud resources can be added - from an architectural point of view - within all application layers, depending on the needs of the application.

3.2 Cafe - Composite Application Framework

Cloud service models have been introduced in Fundamentals Subcection 2.1.3. On top Mietzner and Leymann proposed a possible, additional service model: *Composition as a Service* (CaaS). This layer focuses on creating new applications from existing ones, provided by any service model layer beneath. Based on this idea the *Cafe* platform [Mie10] was developed as part of Mietzner's PhD studies.

Cafe is an acronym and stands for: *Composite Application Framework*. Hence, the platform provides means: to model composed applications, provision required resources and execute them in the specified environment, which can be hosted in the cloud or on-premise. As part of the Cafe idea, Mietzner developed a *variability model* to provide generic components with *variability points*, which allows the configuration of the instantiated application template, to fit the new situation.

This variability model is presented in greater detail below, as it forms the foundation of the later on developed variability model, which is used to address customization points within implemented patterns.

3.2.1 Cafe's Variability Model

To address variability of developed application templates, Mietzner proposed a XML-based variability model [Mie08], which provides means to: specify references to the concrete locations of variability, model dependencies between such *variability points* and specify possible *alternatives* as well as *enabling conditions* for them.

Terminology
The *variability descriptor* element forms the root element of such a variability model. It contains a set of *variability points* and a set of *dependencies* between those variability points. Each variability point has a set of so-called *locators*, which reference the locations of variability in the application template, furthermore, it contains definitions for the possible values; so-called *alternatives.*

As the mechanism referencing variability varies, various locator types exist:

- *XPath-Locator* - This locator references a XML-based variability point via a single XPath expression.
- *Properties File Locator* - This locator references a property within a property file, which is embedded in the application template.

Also different alternative types exist, to model different scenarios:

- *Explicit Alternative* - An explicit alternative specifies values at design time, within a pre-defined range.
- *Expression Alternative* - An expression alternative consists of an expression, which is used to query the required information.

- *Empty Alternative* - An empty alternative marks that a variability point may be filled with an empty string, to indicate an optional point.
- *Free Alternative* - A variability point marked with this alternative can be filled with any arbitrary text.
- *Locator-based Alternative* - A locator-based alternative marks that the referenced default values are not touched.

Enabling Conditions are bound to dependencies and enable the referenced variability point when the enabling condition is met.

Figure 3.1 illustrates a sample application, consisting of an application template, which is referenced by its variability model.

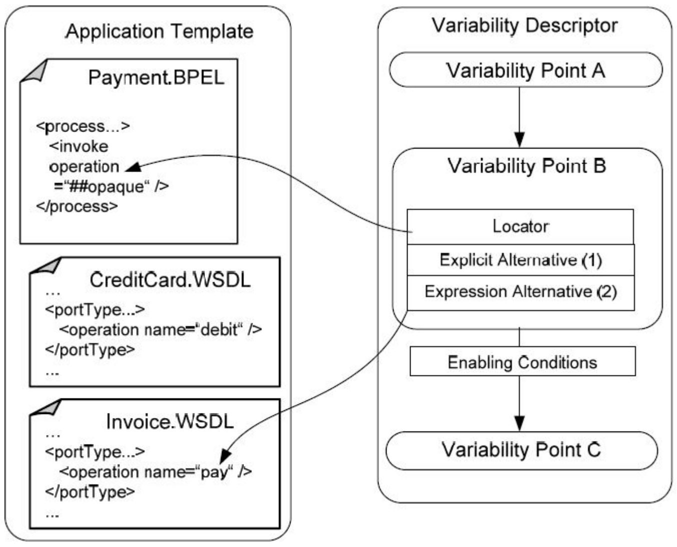


Figure 3.1: Example - Cafe Application Template and Variability Descriptor [Mie08].

Note: Further literature about Cafe can be found at Mietzner's PhD thesis [Mie10] and the Cafe Website [Mie].

3.3 Apache Maven

Patterns in general provide the ability to be combined and reused again. This is also true for cloud computing patterns, since they inherit this pattern property. To implement this property is a major approach towards automated aggregation and provisioning of pattern-based solutions. Thus, *Apache Maven* [Mava] is used within this thesis. Maven offers the possibility to manage Java-based projects in a unified fashion. This means, that projects can be combined in a recursive way, forming new artifacts, which can be used for combining new ones again. This core feature allows unlimited reusability of generated artifacts and speeds up development drastically, since integration of third party code can be done within seconds, as long as the third party code obeys the maven conventions. Maven evolved from the need to unify project management and the corresponding project build life-cycle. It covers the complete project specific workflow, by providing a unified, generic build workflow, on which developers can bind actions fitting their needs. The unlimited ability to aggregate projects also results in the aggregation of the corresponding build workflows. Hence, building a top-level project results in building all dependent projects too. Because of those capabilities, Maven is often used within the so-called *software configuration management* (SCM), which focuses on providing methods and processes to rapidly change software, according to new requirements.

Maven is used within this thesis due to its primary objectives [1]:

- *Making the build process easy* - Since Maven unifies the project structure and the build life-cycle, it reduces frequent configuration efforts.
- *Providing a uniform build system* - This enables developers to rapidly aggregate and build large projects with unified methods.
- *Providing quality project information* - Since Maven supports various plugins for generating reports about: code quality, test results and custom reports.
- *Providing guidelines for best practices development* - This enables a mutual understanding, as the structure, the build process and the project itself is unified.
- *Allowing transparent migration to new features* - Maven supports a transparent update process of the maven core, this guarantees to use new features, while still being able to support features of previously versions.

Maven's Core Paradigm
The above outlined objectives point out Maven's core paradigm: *Convention over Configuration*; Meaning, that unification of configuration at any level of the project configuration, leads to reduced development efforts.

[1]These objectives have been formulated by the *Apache Maven Community* [Mava]

3.3.1 Terminology

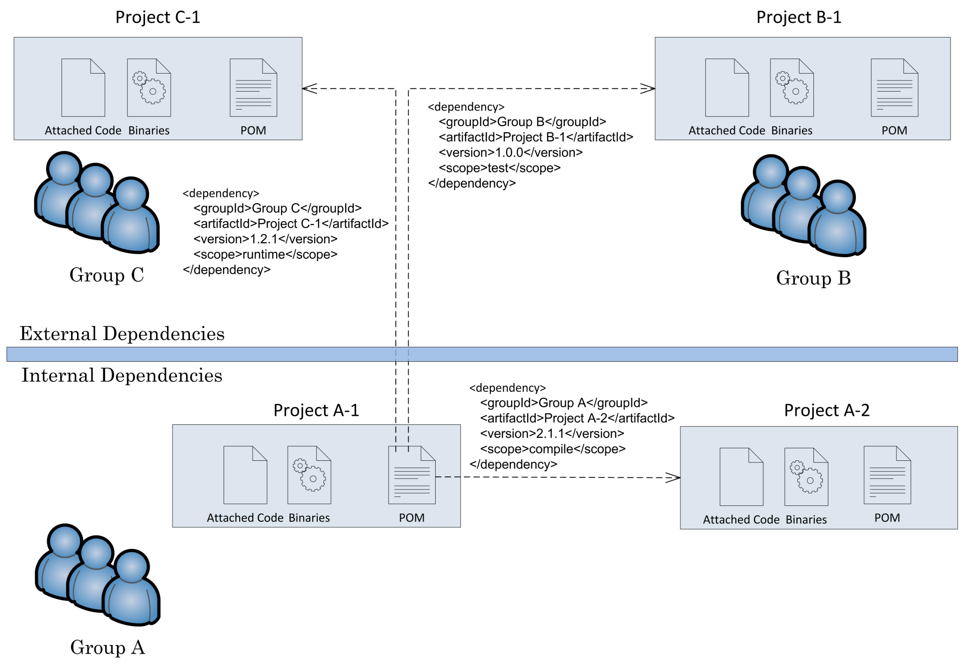
This subsection covers basic terminology required for *Apache Maven*.

- *Project* - A project consists at least of a *pom* file and generates, in case of a successful build, a so-called *artifact*, which then can be referenced by other projects.
- *Group* - Within Maven, the term group can be applied to any entity publishing code artifacts. These may be: persons, teams, projects, companies or organizations.
- *Artifact* - An artifact is the result of a successful project build life-cycle. It consists of the created binary files, e.g. .jars, .wars, .ear and .class files, and is identified by a tuple, consisting of a unique *group id*, *project id* and a *version id*.
- *Archetype* - An archetype acts as a template for new projects. Thus, it provides the project structure and required files, related to the required project type.
- *Dependency* - A dependency is a relation between a project under development and the binaries of a referenced project, identified by its group id, artifact id and version id.
- *Scope* - A scope specifies the phase of the standard build life-cycle, when the referenced artifact has to be fetched by Maven and processed. Furthermore, it influences whether the referenced artifact is embedded in the resulting build artifact or not.

Possible dependency scopes are:

- *Compile* - The artifact is visible on all build paths within the project. The dependency is also propagated to all child projects. The generated artifact needs to be embedded in the final package.
- *Provided* - The artifact is provided within the container, where it is going to be deployed. It is mainly used for container specific libraries e.g. parts of the *Java EE* API.
- *Run time* - The artifact is provided within the target run time.
- *Test* - The artifact is only required during the test phase.
- *System* - The artifact is installed on the target system. Since it can not be loaded from a repository, the location to the artifact need to be specified as well.
- *Import* - Imports the dependencies listed in the dependency management section of the referenced target pom.

Figure 3.2 illustrates the above mentioned terms and relations - except the term *archetype*, since an archetype acts as template for projects to be created.



```
<dependency>
  <groupId>Group B</groupId>
  <artifactId>Project B-1</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
</dependency>
```

```
<dependency>
  <groupId>Group C</groupId>
  <artifactId>Project C-1</artifactId>
  <version>1.2.1</version>
  <scope>runtime</scope>
</dependency>
```

```
<dependency>
  <groupId>Group A</groupId>
  <artifactId>Project A-2</artifactId>
  <version>2.1.1</version>
  <scope>compile</scope>
</dependency>
```

Figure 3.2: Example - Dependencies between Maven-based Projects.

Group A holds two projects: *Project A-1* and *Project A-2*. *Project A-1* has an internal dependency to *Project A-2* and the related artifacts, these artifacts will be fetched at compile time and will be placed into a resulting package. *Project A-1* also has external dependencies to *Project C-1*, created by *Group C*, these artifacts are present during development but are removed from the resulting package, since they are provided by the target run time. *Project A-1* also references *Project B-1*, created by *Group B*, which will be present only during testing. Each Maven project consists of: a `pom.xml` file, specifying dependencies; binary artifacts, implementing the features of the project; and possible attached source code, mainly used for debugging purposes. When building *Project A-1* Maven automatically fetches all required artifacts, defined within the pom.xml of *Project A-1*, and generates the binary artifacts for *Project A-1*, these artifacts then, again, can be referenced.

3.3.2 Life-Cycles, Phases, Goals and Plugins

Maven unifies various life-cycle phases within the development of Java-based software. Thus, the Maven specific terms: *life-cycle*, *phase*, *goal* and *plugin* are introduced briefly, as they form the foundation for the later on presented concept.

Definition: A *life-cycle* focuses on executing various sub-tasks in an ordered fashion, to solve an overall, artifact-centered, aggregated task. Such aggregated tasks are: build the application, clean the workspace and generate relevant reports. Each sub-task, within those abstract flows, is only executed when all of the sub-tasks defined above have been executed successfully.

The following standardized life-cycles exist:

- *Clean* - Results in cleaning the current workspace.
- *Default* - Results in building the current workspace, i.e., performing tests, creating required code formats and deploying the created binaries into the local respectively remote artifact repository.
- *Site* - Results in generating reports for the current workspace and putting them on the maven generated project site. This may include: testing reports, code quality reports or any other development-centered reports.

Life-cycles depend on each other, in the order mentioned above, meaning that executing the site life-cycle results in the execution of the clean life-cycle, the default life-cycle and finally, the site life-cycle. Yet, the clean cycle, will not force any other to be executed, since there is no causal dependency.

Definition: A *phase* is a process within a life-cycle. It consists of various goals, also called *Mojos* (Maven Plain Old Objects), which are invoked during the requested phase execution.

Definition: A *goal* is nested within a phase. It forms an atomic action, in the sense of Maven, and is wrapped within a so-called *plugin*.

A goal can be: starting an application server, deploying application relevant files, or any other implemented action.

Definition: A *Maven plugin* encapsulates the functionality of multiple goals. It is bound to a phase of a specific life-cycle within the pom-file.

Within Maven standard plugins exist, e.g. performing steps to clean the workspace, package the application or run unit-tests. Moreover, Maven provides the possibility to develop custom plugins, which can be bound to any life-cycle phase.

3.3.3 Maven Repository Types

Repositories form the foundation of Maven's reusability approach, as they store code artifacts with related meta-data. These artifacts can be referenced and retrieved within the pom.xml - which is going to be explained later on. Therefore, as different repositories exist within a Maven-based project, common constellations are outlined in the following.

The following repositories are commonly used during Maven-driven development:

- *Local Artifact Repository* - This repository is hosted on each machine using Maven to build the project. It stores locally generated code artifacts as well as artifacts, which have been fetched from remote repositories from different groups, required to build the project. Thus, it acts as a local binary cache.
- *Remote Artifact Repository* - This repository, from a developers point of view, is hosted on a remote machine and stores generated code artifacts from a group or company. Artifacts can be retrieved and stored within the local repository, it is also possible to push locally built code artifacts into the remote repository to make them accessible for other developers.
- *Code Repository* - A code repository contains the source code of the related binary artifacts. Such repositories may be: *SVN* [SVN], *CVS* [CVS], *Git* [HT] or any other tool, capable of version control.

Code repositories are not Maven-specific and they are often used to manage source code, while Maven is used to manage project dependencies and their related binary artifacts.

Figure 3.3 illustrates a common constellation between source repositories and maven repositories. Developers fetch source code from code repositories, manipulate code, build the project and publish artifacts to their local artifact repository. Manipulated code is pushed back into the corresponding code repository. After building the project, the generated artifact binaries can be pushed into the related remote repository, from where other groups/teams are able to fetch them.

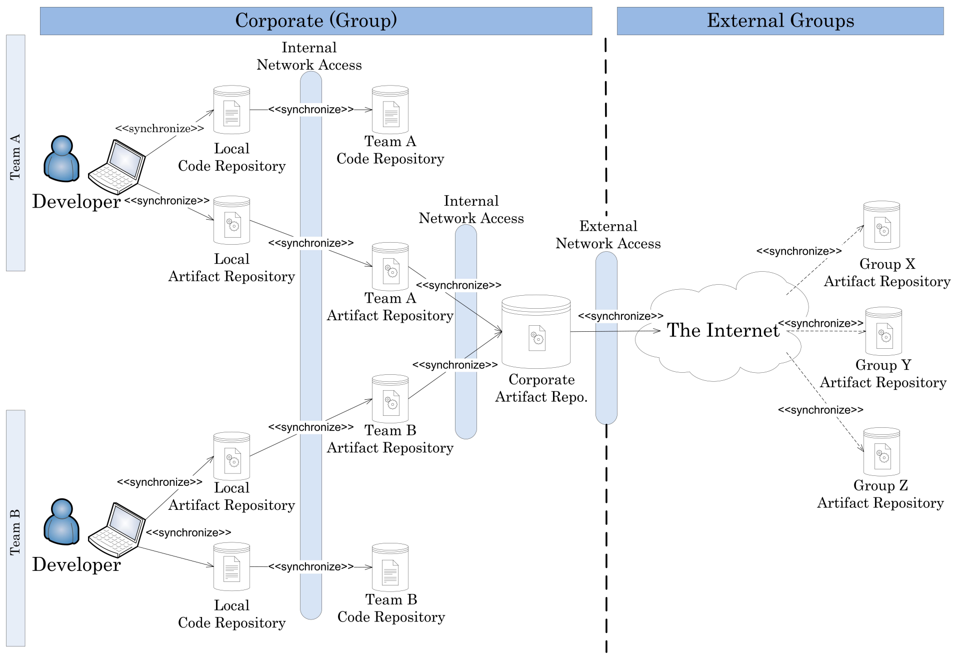


Figure 3.3: Example - Constellation of Maven and Code Repositories within Development.

3.3.4 The Project Object Model

The *Project Object Model* (POM) forms the foundation of a Maven-based project, since it contains definitions of dependencies to required projects and their related code artifacts. The information, nested within a pom, is represented via XML, furthermore, pom files can reference each other via aggregation or inheritance, enabling a so-called dependency graph, which is resolved by Maven automatically, when the build process is invoked. As a result the so-called *effective pom* is computed, which contains all information, reachable within the dependency and inheritance graph, beginning from the initial project.

To provide a full overview of Maven's capabilities of dependency management, the structure of a pom.xml file and related parts are listed in the following.

A `pom.xml` file is a structured XML file, it mainly consists of:

- *Potential References to a Parent POM* - From where a child pom file can inherit dependencies and properties.
- *Potential Properties* - Which are acting as variables for the pom and it's children.
- *Mandatory Artifact Information* - This information consists of a group id, artifact id and version id, to identify the generated code artifact uniquely.
- *Optional Artifact Information* - This information consist of authors, licenses etc..
- *Dependencies* - Dependencies define which artifacts have to be fetched and integrated by Maven, during the build process.
- *Life-Cycle Attachments* - These definitions can be used to tell Maven which actions should be performed when the standard life-cycle has reached a defined phase; such actions are implemented in plugins.
- *Profile Definitions* - These definitions are used to influence the behavior of the build, since profiles act as triggers, which enable or disable build functionality.

Further configuration options exist within the pom.xml, like: mailing list options, software configuration management options, artifact repository configurations. Those can be found on the Maven project site [Mava].

3.3.5 Maven Profiles

Profiles can be defined within the pom file of a Maven-based project, as different builds require different dependencies, plugins, properties and configurations. Thus, profiles offer the possibility to enable or disable parts of the effective pom, resulting in different build behavior. Therefore profiles are used to express variability within Maven-based projects. Such variability may affect the packaging format, deployment behavior, report generation, testing scenarios or any other activity related to the corresponding maven life-cycle. As a major part of this thesis is to identify customization points in the implemented cloud computing patterns, profiles are introduced with greater detail in the following paragraphs.

The following profile types exist: [2]:

- *Per User* - These profiles are defined within the *user* `settings.xml` file.
- *Per Project* - These profiles are defined within the project's `pom.xml` file.
- *Global* - These profiles are defined within the *global* `settings.xml` file.

These profiles can be activated:

- *Via Explicit Invocation* - Done by explicitly passing the profile name.
- *Via POM Declaration* - Frequently a property is declared acting as a trigger for the related profile. Depending on the entered value, the profile is enabled or disabled.
- *Via User Variables* - These variables are nested within the user environment variables. Maven is capable of querying them to activate profiles.
- *Via Environment Variables* - Like user variables, Maven is able to read operating system variables to trigger profiles.
- *Via Defined Conditions* - These conditions can be defined within the profile definition itself, e.g. the presence or absence of a specified file can be checked.
- *Via Values in the Maven settings.xml files* - Values defined within a settings.xml file can also be used to trigger profiles.

Furthermore, two settings.xml files exist, which potentially store profiles. Maven includes these profile definitions and variables during its executions.

The following types of `settings.xml` files exist:

- *Global settings.xml* - The global settings.xml stores *maven run-time specific* information, which is used when building *any* Maven-based project.
- *User settings.xml* - The user specific settings.xml stores *user specific* information and profiles, like: login credentials, server addresses, user centric profiles and proxy configurations. Information specified in the user settings.xml file may override information form the global inherited settings.xml file.

[2]Note: Maven 2.0 provided the possibility to define profiles within a separate `profiles.xml` file. Since this thesis uses Maven 3.0, this type is not discussed here. Further information can be found on the Maven site [Mava].

3.3.6 Maven Project Types

As Maven provides the possibility to reference pre-build code artifacts, it also provides the possibility to reference local sources, nested within Maven-based projects. This is used to structure the applications to be developed into several sub-systems. Such sub-systems are then mapped to single Java-projects. Each project consists of at least a pom file and may have source code nested within. This enables developers to structure their projects according to their design documents, giving the possibility to define spanning-projects as well as subordinate projects.

All Java-based code developed in this thesis - especially Maven plugins and implemented cloud computing patterns - use Maven as build management tool. Thus, the maven project types are represented in detail:

- *POM*-Project - A *POM*-Project is an abstract project consisting only of a pom-file. It is used as root-node within the project structure. It references projects of the type *project* or *module*, to enable aggregation.
- *Project*-Project - A *Project*-Project is an independent project, which *may* be referenced by other project types or *may* reference a root project, to inherit properties or dependencies.
- *Module*-Project - A *Module*-Project is part of an aggregated project, thus, it has to reference a root project.

3.3.7 Distinction to Related Tools

This subsection briefly covers the differences to related build and project management tools.

Differences between Apache Maven and Apache Ant

While Ant [Ant] tries to provide means to automate development with build scripts, required to be parametrized to the concrete project structure, Maven focuses on standardizations on all relevant development levels, like: project structure, build life-cycle and dependency management. This allows Ant to be more flexible than Maven, since it lacks a closed build workflow, but makes it more complicated to integrate projects, as the structure and the build workflows varies. Furthermore, Ant lacks the possibility of dependency management. This is included in Maven by default, since Maven uses *Apache Ivy* [Ivy] as dependency management provider.

Differences between Maven and Gradle

Gradle [Gra] combines the benefits of Maven and Ant. Therefore it provides a standardized project structure, based on the Maven project-structure as well as an open build workflow, which can be extended at any time by Ruby-based code. This enables Gradle developers to simply integrate Maven-based projects without additional effort, while still be able to add custom code, fitting their needs. Yet, the used domain specific language (DSL) is a major drawback when building fully automated provisioning systems. As manipulating these configuration files may require to parse the DSL file, which may not be possible all times, since

the DSL is not restricted by default. Also gradle-based projects can not be easily integrated into Maven-based projects, since the invocation of the Gradle build workflow is not supported by default.
Thus, Maven is preferred within this thesis instead of Gradle, as manipulating XML-based configurations can be done, in contrast to manipulating DSL-based files. Also Maven is compatible with Ant and can be integrated into Gradle projects, while the opposite does not hold true. Furthermore, Maven's closed build workflow forces developers to wrap custom code within plugins, this promotes reusability, since plugins can be plugged-in in the standardized life-cycles.

Hence, within this thesis, Maven is preferred over Ant and Gradle, as closed standardization is a major issue when implementing reusable software artifacts.

3.3.8 Relation to Software Product Line Engineering

Various terms have been introduced during this chapter, which can be mapped on terms of the previously discussed software product line engineering field, like: *system artifact*, *archetype* and *variability.* These relations are outlined briefly within the following paragraphs.

Note: Software product line engineering demands methods for reusing created *system artifacts*, Maven provides the implementations to these demands, as it offers possibilities to create, store and retrieve artifacts in an automated fashion. Thus, Maven is used in this thesis to provide reusability of implemented software artifacts.

Relations between Maven and Software Product Line Engineering:
The Maven-specific term *artifact* directly correlates to the software product line engineering term *system artifact.* Likewise, the Maven-specific term *archetype* directly correlates to the software product line engineering term *archetype*, as a maven archetype provides the structure - respectively the pattern - for a well known project type. In contrast, the software product line engineering term *variability* can not be mapped directly to Maven's terminology, as the Maven-specific files - *pom* and *settings.xml* - lack capabilities to model *variability points* and the corresponding binding process.

Combining benefits, provided by build management tools, like Maven; the variability, provided by software product line engineering; and the capabilities of infrastructure provisioning tools, like Puppet; is a main objective of this thesis, as applying these benefits on the development of pattern-based cloud software artifacts speed up the time-to market factor.

Thus, the following section covers functionality provided by puppet, which is required for advanced sections.

3.4 Puppet Labs Puppet

Automating processes within software development and operation life-cycles is a key factor within technology driven IT businesses, as the amount of managed nodes grows rapidly with new technologies, like *Cloud Computing* and the related *Big Data Movement*. Thus, the complexity of provisioning nodes and deploying software on large clusters grows, while developer capacities keep constant. This outlines the challenge of the DevOps Movement, which tries to solve development and operations tasks with code-based infrastructure definitions; called *infrastructure as a code*.
While Maven aims to unify and automate the software artifact life-cylce, similar tools exist to automate infrastructure life-cycle tasks as well. Thus, this chapter focuses on introducing *Puppet Labs' Puppet* [Pup] - an open source infrastructure managing tool, which implements the *infrastructure as code* idea.

3.4.1 Terminology

This subsection covers basic terminology required for Puppet Labs Puppet.

Puppet Terminology:

- *Action* - An action is a pre-defined task, which can be executed by *puppet*.
- *Middleware* - Middleware depicts the software artifacts puppet is supposed to manage, like: application server, message oriented middleware, databases etc..
- *Puppet Manifest* - A *manifest* specifies infrastructure management actions that have to be executed by *puppet*, in order to install demanded middlware.
- *Puppet Module* - Manifests can be aggregated in so-called *modules*, forming a collection of actions, which can be executed by corresponding puppet agents.
- *Puppet Node* - A puppet node is a machine hosting the puppet specific environments.
- *Puppet Agent* - A puppet agent is a software environment that is able to perform *actions*, defined in modules and manifests, in order to install middleware.
- *Puppet Server* - A puppet server dispatches actions, defined in modules respectively manifests, to a set of subscribed puppet agents. It is installed on a puppet node.
- *Puppet Client* - A puppet client is a software client, which is able to connect to a puppet agent and deploy manifests respectively modules into the agent's environment.
- *Puppet User* - A puppet user is an entity, which interacts with puppet, using a client.

The required middleware configuration is stored in separate modules, thus, composing application stacks is simplified, since combining those code-based modules is supported by Puppet. This enables Puppet users to generate any combination of middleware, allowing them to provision those solutions to any amount of nodes.

As Puppet encapsulates system details, within its domain specific language, it is possible to configure systems from an abstract point of view, enabling users to speed up configuration time, as system specific details are executed by Puppet transparently. Furthermore, Puppet provides the possibility to integrate custom code, as its DSL is Ruby-based. Thus, provisioning of cloud-based services can be integrated as well.

3.4.2 Puppet Constellations

Puppet provides two constellations to execute manifests on computation nodes. The *standalone constellation* is used when the required middleware is only required on a single machine. The puppet agent, running on the that node, is therefore invoked directly by the developer. The *master/agent constellation* is used when the required middleware has to be deployed on various nodes. Thus, the interaction takes place via a puppet client, which transfers the defined configurations to the *puppet master*. Puppet agents, subscribed to that puppet master, then, fetch the specified configurations and execute them on their nodes. Such nodes can be hosted on-premise or in the cloud, as both deployment models run the same agent installation.

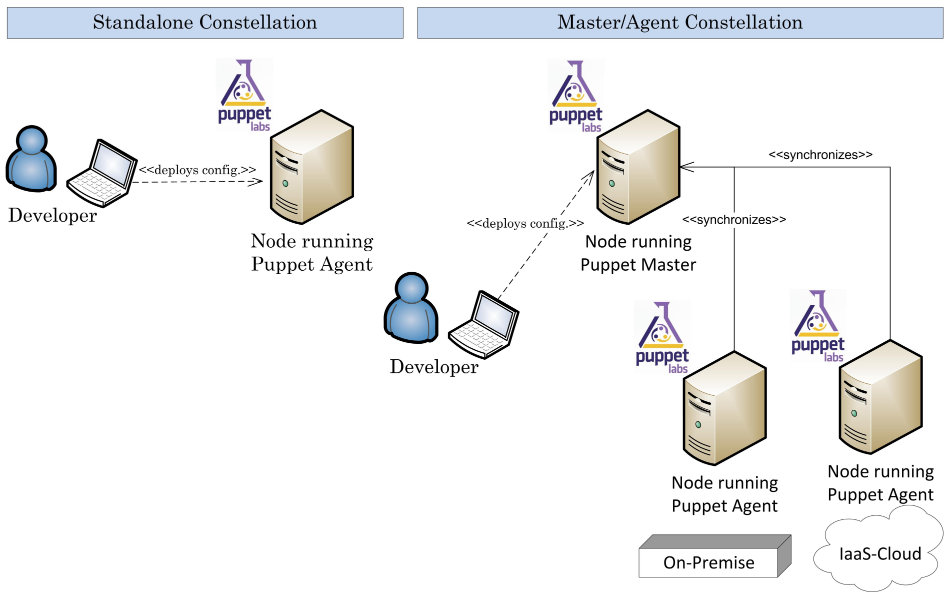


Figure 3.4: Puppet Constellations - Standalone and Master/Agent Constellation.

3.4.3 Related Technologies

Puppet implements ideas of the DevOps Movement, which tries to shift development and operations together. During this movement different tools emerged, providing functionality like Puppet. Thus, this subsection covers related tools as well as relations to build management tools.

Relations to Opscode Chef

Opscode's Chef [Che] also implements the *infrastructure as code* idea and provides equal functionality compared to Puppet. Single configurations are stored in so-called *recipes*, while aggregated configurations are stored in so-called *cookbooks*. These terms directly correlate with the puppet-specific terms *manifest* and *module*. Also the provided deployment constellations can be mapped directly. Like Chef, Puppet provides a standalone constellation as well as a client/server constellation.
The major difference between Puppet and Chef is, that Puppet uses a closed configuration model. This means that custom actions have to be wrapped within custom manifests or modules, while Chef provides the possibility to directly define custom actions within the configuration model.

Relations to Build Management Tools

The ideas of Chef, as well as Puppet, correlate with the previously introduced build management tools *Maven* and *Gradle*. Maven and Puppet aim to establish a closed configuration model, where custom actions have to be implemented within pre-defined code containers: plugins respectively manifests. Chef and Gradle provide an extensible configuration model, which allows to directly include custom actions. This, however, results in reduced reusability, as code is not wrapped within a pre-defined interface; and increased querying complexity, as imperative code-based definitions complicate automated data retrieval.
Both ideas share benefits as well as drawbacks, thus, the usage of Gradle combined with Chef also forms a possible tool combination to challenge the issues covered within this thesis as well. Yet, the ability to query information within configuration files is an important topic towards advanced automated pattern aggregation, as manipulations on the configuration files are mandatory. Thus, Maven combined with Puppet is used within upcoming chapters.

Concept and Specification

The recently identified *cloud computing patterns*, presented in Section 3.1, may serve as a base for aggregated cloud applications, which may form a cloud computing pattern again. Implementing such aggregated patterns still involve redundant human interaction, when adding such pattern artifacts to an application, as they need to be configured manually, to suit the new situation. This configuration also requires the adaption of provisioning, deployment, decommissioning and undeployment actions. Storing such created artifacts in a customizable fashion, within a so-called *pattern template catalogue*, offers the possibility to store, instantiate and combine crafted pattern artifacts in a reusable, automated fashion. A concept, capable of implementing these benefits, combined with the possibility to configure and provision such instantiated pattern artifacts, is presented in following.

Section 4.1 provides an overview of a possible *software product line for cloud patterns*, to classify this thesis within its superordinate research context. This software product line is capable of creating and managing aggregated cloud computing pattern artifacts. Therefore, an abstract overview of the required architectural components and roles is provided. Afterwards, Section 4.2 discusses functional and non-functional requirements of the implemented components, covered in this thesis, which implement major parts of the presented software product line. Then, Section 4.3 covers the functional architecture of the developed concept and introduces mandatory components, roles, their interrelations and associates corresponding use cases. Section 4.4, describes the human driven pattern artifact development process, which is assisted by the functionalities of the presented concept. Section 4.5 and Section 4.6 provide subordinate, required models for the introduced concept, i.e, the *functional project structure model*, which defines how aggregated, pattern-based applications are structured; and the *functional variability meta model*, which defines how variability, respectively variability points and their components, can be modeled. Section 4.7 formally introduces the so-called *variability graph*, which forms the input of the *configuration flow computation algorithm*, presented in Section 4.8. Finally, Section 4.9 introduces functional components, which form the foundation for the *Design and Implementation Chapter*.

4.1 Superordinate Context

Offering pattern artifacts in a reusable fashion to various users in a so-called *cloud computing pattern template catalogue* is an ongoing research topic [FLRS12]. This idea can be extended by concepts provided by software product line engineering, discussed in Fundamentals Section 2.2. Such a *software product line for cloud computing patterns* allows to model aggregated applications, instantiate them into a workspace and perform various tasks on the so created artifacts. Moreover, it allows to store aggregated, created artifacts in a reusable fashion, which enables the recursive composition of pattern artifacts.

Implementing essential parts, of the above mentioned software product line, forms the main objective of this thesis. Thus, this chapter provides a concept to support: the *storage*, *initialization*, *aggregation*, *configuration*, *provisioning*, *deployment*, *decommissioning* and *undeployment* of pattern-based cloud applications.

The following paragraphs provide an abstract overview of the architecture of the outlined software product line, including their components, roles and interrelations. Succeeding sections then refine components and scenarios, implemented within the work of this thesis.

A Software Product Line for Pattern-based Cloud Applications
A software product line requires different components to: model, store and instantiate pattern-based applications; these components are outlined in the following paragraph.

The *software product line* consists of:

- *The Pattern Template Catalogue* - The pattern template catalogue stores pre-implemented pattern archetypes, which offer customization points to developers. These artifacts form the foundation of aggregated cloud applications and are offered for further usage.
- *The Catalogue Console* - The catalogue console offers a visual editor to an end-user and provides functionality to model aggregated applications with stored, pattern-based artifacts. These artifacts are stored within the pattern template catalogue. Therefore, it provides a human readable interface and allows users to create a so-called *pattern solution archive*, which contains meta-information about the modeled solution.
- *The Pattern Factory* - The pattern factory interprets the *pattern solution archive* and executes required actions, specified within the passed meta-information. Such actions may be the instantiation of artifacts into a *workspace*, the execution of a required configuration process on the instantiated pattern artifacts, as well as the provisioning of defined environments and middleware.
- *The Workspace Agent* - The workspace agent implements the tasks, triggered by the pattern factory. Dependent on the selected pattern implementation, the agent performs the requested actions on the pattern artifacts, like pattern instantiation, aggregation, configuration and provisioning.

Figure 4.1 illustrates the above introduced components and defines various roles.
A *pattern developer* stores created pattern reference implementations, by installing pattern artifacts into the pattern template catalogue repository and adding meta-information how these pattern implementations can be addressed. These patterns, then, become visible within the *pattern template catalogue*. *Architects* then create new pattern applications, by aggregating pre-existing pattern artifacts and bind corresponding variability points. This can be done via a graphical, web-based interface, called the *pattern catalogue console*. These aggregated pattern artifacts may then again be stored in the pattern template catalogue. The *pattern factory* forms a container environment for the created *pattern solution archive*, which hosts the required meta-information about the created solution. This container, then, depending on the defined actions, triggers functionality, provided by the *workspace agent*. The *workspace agent* offers its functionality via an interface, that allows *developers*, as well as the *pattern factory*, to trigger processes on pattern implementations, nested within the workspace.

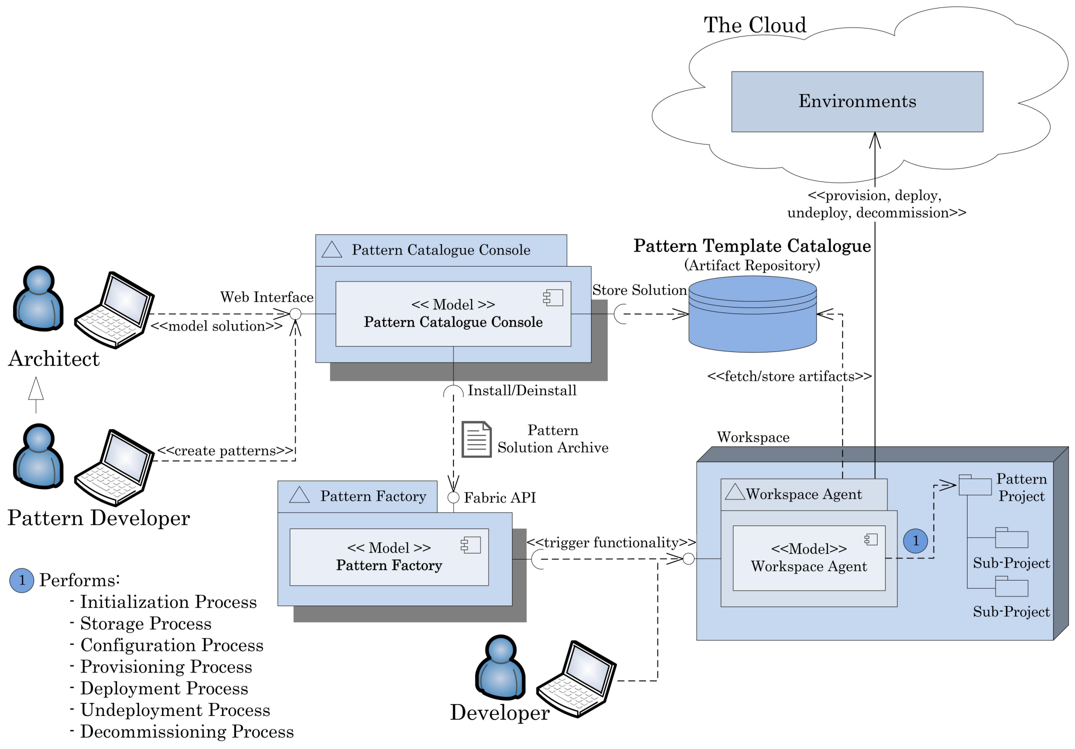


Figure 4.1: Overview - Components of a Software Product Line for Cloud Patterns.

Implementing the *pattern catalogue console* as well as the *pattern factory* is out of scope of this thesis, as it involves the development of a structural meta-model for aggregated, pattern-based applications. Besides, the focuses of this thesis lies on the aspects of creating reusable, customizable pattern artifacts and their automated configuration as well as their provisioning and decommissioning. Hence, the following sections provide a concept, capable of these tasks.

4.2 Requirements

This section covers the required functional as well as non-functional requirements, to store customizable pattern artifacts within the *pattern template catalogue* and to support the automated instantiation into a workspace, by invoking goals on the *workspace agent*, presented in Section 4.1. The so created pattern implementation can then be configured and various actions can be invoked on it. These requirements are presented in detail in Subsection 4.2.1 and 4.2.2.

4.2.1 Functional Requirements

- *Creation of a Pattern Artifact* - Pattern artifacts should be implementable within a pre-defined structure, which allows to store them in the *pattern template catalogue.*
- *Instantiation of a Pattern Artifact* - A previously stored pattern implementation should be instanceable into a Java-based development *workspace.*
- *Configuration of a Pattern Artifact* - A pattern implementation should provide customization points, which allow to configure the application individually.
- *Provisioning of Nodes and Middleware* - The selected cloud environment affects the execution steps, required for the provisioning of nodes and middleware.
- *Undeployment of a Pattern Artifact* - A deployed pattern implementation should be removable after a successful deployment.
- *Decommissioning of Nodes and Middleware* - Acquired resources should be deallocated, at any time, after the pattern implementation has been removed.

4.2.2 Non-Functional Requirements

- *Reusability* - A created pattern artifact should be storable and retrievable, from a *pattern template catalogue* to support reusability.
- *Modularity* - A created pattern artifact should be composable, to enable composition of larger pattern-based projects by aggregating other pattern artifacts.
- *Compatibility* - The developed system should be able to integrate any external DevOps tool, like *Puppet* or *Chef.*
- *Independency* - The mechanisms to deploy pattern artifacts and provision related middleware should be independent of any concrete cloud environment.
- *Customizable* - Implemented patterns should be customizable, to provide a wide scale of possible configurations.

Note: The requirements discussed within this section are covered by the use cases supported by the *workspace agent* and are described in detail in Use Case Appendix A.

4.3 Functional Architecture

This section presents the functional architecture, implementing the requirements introduced in Section 4.2. The presented architecture focuses to support a managed creation, instantiation, customization and provisioning process of cloud computing patterns, by utilize *Apache Maven* and *PuppetLabs' Puppet*, discussed in Related Work Section 3.3 and 3.4. These tools are combined with the introduced concept of *variability descriptors*, presented in the Related Works Section 3.2.1. The architectural roles, components and their interrelations are introduced in the following section and are then presented in Figure 4.2.

Architectural Roles
The following architectural roles exist:

- *Workspace User* - A workspace user is an abstract entity, which instantiates, combines, configures and provisions stored, pattern-based artifacts.
- *Developer* - A developer is a human *workspace user*.
- *External System* - An external system is a non-human *workspace user*.

Architectural Components
The following architectural components exist:

- *Development Machine* - The development machine is a node running *Apache Maven*, to manage created, pattern-based artifacts. It hosts the so-called *development workspace*.
- *Development Workspace* - The development workspace is a directory, which hosts sources for creating pattern-based artifacts. These sources are Maven-based, meaning that each source is part of a *maven project*, consisting of a *pom.xml* file and a *variability descriptor*.
- *Apache Maven* - Within Maven, a plugin is running to support different workspace-users tasks, these tasks are presented in detail in Use Case Appendix A.
- *Workspace Maven Plugin* - Dependent on the requested actions, the workspace plugin, installed on the development machine, invokes actions on the pattern artifact, hosted within the *development workspace* and performs the required actions to configure it, provision its environment and to deploy its binaries.
- *DevOps Tools* - DevOps tools, like Chef or Puppet, can be integrated to execute required provisioning tasks of nodes and middleware. Dependent on the bound maven plugin, that is defined within the pattern's *pom.xml*, the corresponding DevOps tool is invoked and provisioning scripts are read from the workspace.
- *Artifact Repository* - The artifact repository stores required and generated artifacts, i.e. *Maven plugins*, *code artifacts* and *pattern archetypes*, and offers them to external usage. The workspace first accesses a local repository to fetch artifacts. If the requested artifacts are not present, known remote repositories are traversed.
- *Network* - Network capabilities are used by Maven and the selected DevOps tools to communicate with the addressed cloud environment.

- *Environments* - Environments are nodes within a network, hosting required artifact run times. Such an environment might be:
  - *A IaaS-Cloud with installed middleware* - A IaaS-Cloud environment provides nodes to cloud users, yet, the provisioning of required middleware artifacts has to be done by the corresponding Maven workspace plugin.
  - *A PaaS-Cloud with pre-installed middleware* - A PaaS-Cloud environment provides a pre-defined run time model to the cloud user, thus, created artifacts can be deployed without the need to provision nodes and middleware
  - *An On-premise Environment* - An on-premise environment is a non cloud specific, static environment, hosting required middleware artifacts. These artifacts might be pre-installed and thus, do not need to be provisioned.

Figure 4.2 illustrates the presented functional architecture. Maven-based abstract projects and corresponding implementation projects are stored within the development workspace, where they are managed by the workspace agent. In addition, possible provisioning and deployment constellations for a tomcat-based application [Toma] are illustrated, along with their environments, i.e., the Amazon IaaS-Cloud, running an EC2-instance with a Tomcat server on top (1.); the Amazon PaaS-Cloud, running Elastic Beanstalk - a managed Tomcat server (2.); and an on-premise environment, running a node with a Tomcat server on top (3.).

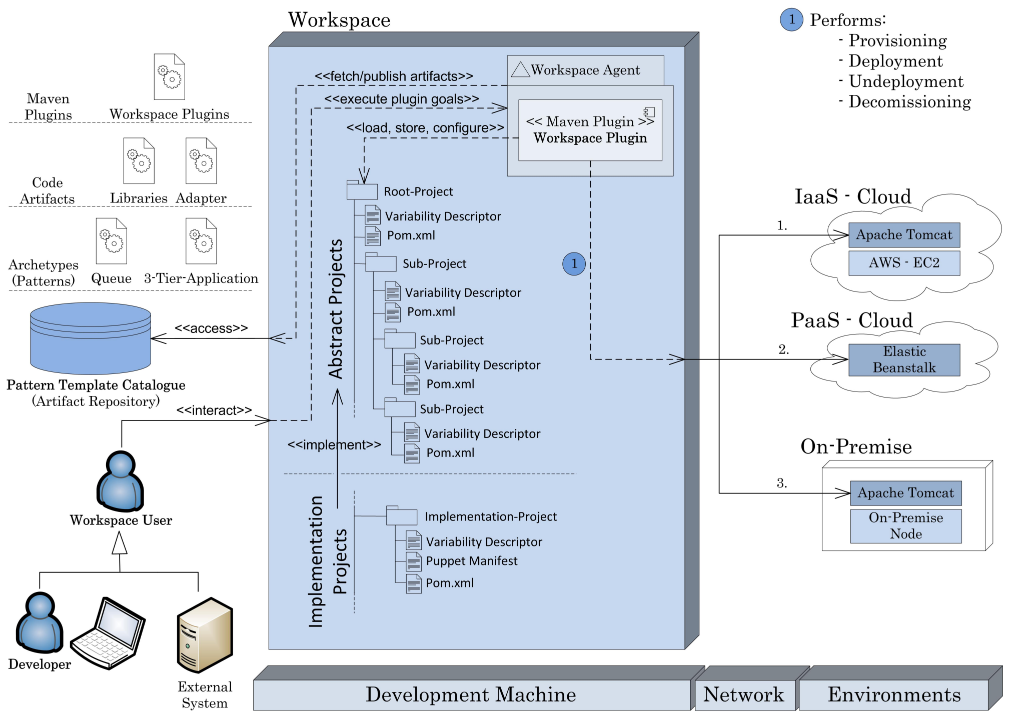


Figure 4.2: Overview - Functional Architecture of the Developed Concept.

4.4 The Cloud Computing Pattern Development Process

This section discusses the human-driven *cloud computing pattern development process*, which consists of two sub-process: *the pattern creation process* and the *pattern usage process*. These processes are supported by the *workspace agent* and the *pattern template catalogue*, presented in Section 4.3. Therefore, the processes include actions that can be performed on pattern implementations, i.e, create, store, initialize, configure, provision and deploy [1]. These actions are described in the following. Their interrelations and the corresponding workflows are then presented in Figure 4.3 and 4.4; using the *Business Process Model Notation (BPMN)* [BPM].

The Pattern Development Process
Figure 4.3 illustrates the *pattern creation process* that is performed by a *pattern developer*, to implement and store a pattern in the *pattern template catalogue*. The pattern developer creates a Maven-based project, which represents a pattern implementation. This pattern implementation is stored locally within the developer's workspace. Afterwards, the pattern developer generates a *pattern archetype*[2], out of the existing pattern implementation, using the *generate archetype* goal of the workspace agent. If the generated archetype fulfills all required features, then the pattern developer persists it into the *pattern template catalogue*, via the invocation of the *persist archetype* goal of the workspace agent. Otherwise the developer manipulates the pattern implementation and again generates a new archetype from it.

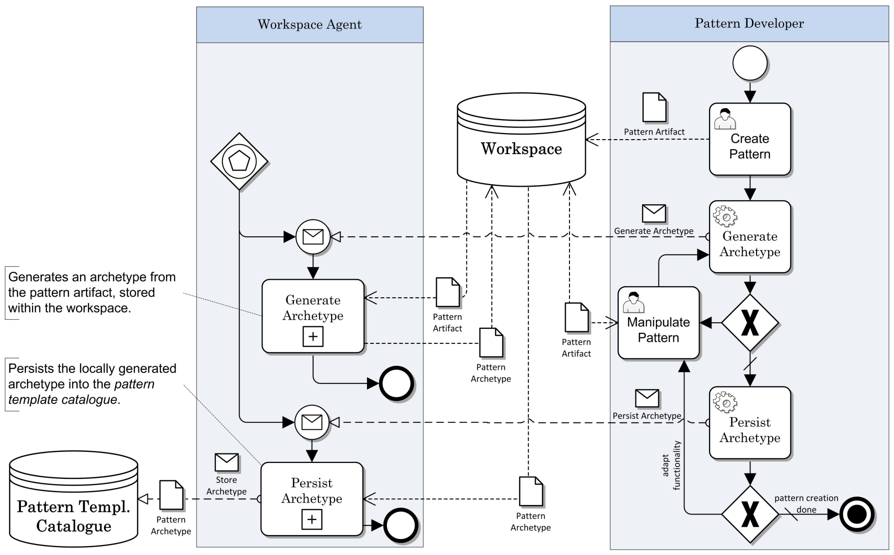


Figure 4.3: BPMN Representation - The Pattern Creation Process.

[1]Corresponding actions are covered in detail in Use Case Appendix A.
[2]Discussed in Fundamentals Section 2.2.1 and Related Works Section 3.3

Figure 4.4 illustrates the *pattern usage process*. This process is executed by the *workspace user*, to load pattern implementations into its workspace and build new, pattern-based applications. Therefore, he initializes a set of pattern archetypes, stored in the *pattern template catalogue*, into its workspace, via the *initialize archetype* goal of the workspace agent. Then, custom features can be added to the pattern-based application. Afterwards, the workspace user customizes its application, by binding the variability points of the application components. This is done by enabling, respectively disabling, variability points and alternatives in the variability descriptors of the pattern projects. Then, the *configure pattern* goal, which pushes bound variability points into the addressed variability locations, can be triggered. The pattern implementation binaries are, then, built by invoking the *build pattern* goal. After that, the required provisioning steps can be executed by invoking the *provision environments & middleware* goal. Finally, the generated pattern implementation binaries can be deployed into their provisioned environments.

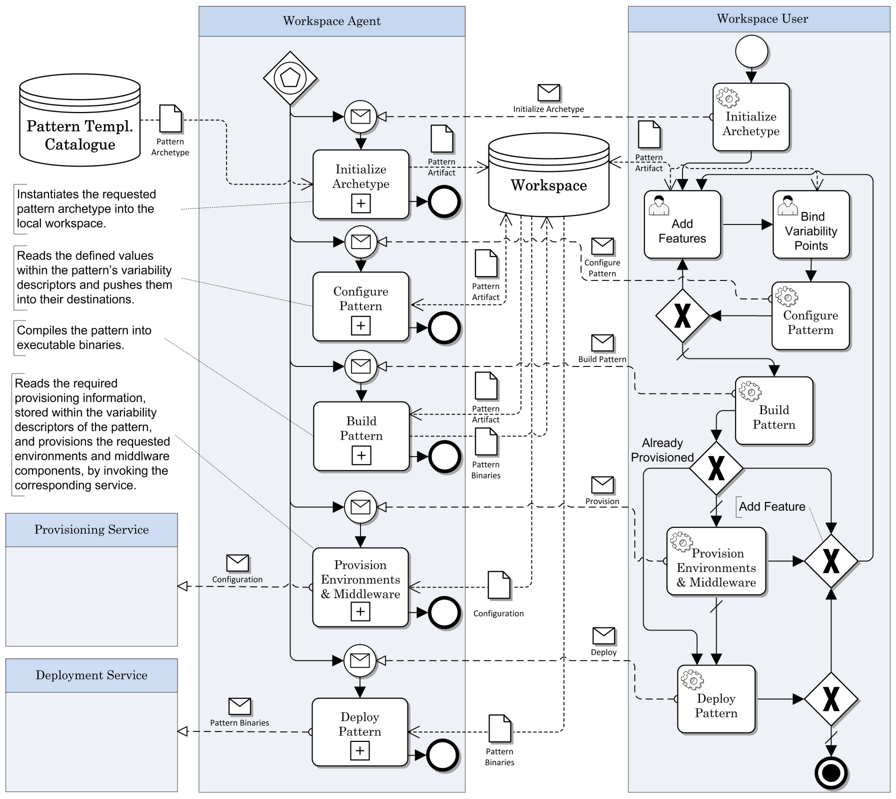

Workspace Agent
Workspace User
Pattern Templ. Catalogue
Pattern Archetype
Initialize Archetype
Pattern Artifact
Workspace
Instantiates the requested pattern archetype into the local workspace.
Reads the defined values within the pattern's variability descriptors and pushes them into their destinations.
Compiles the pattern into executable binaries.
Reads the required provisioning information, stored within the variability descriptors of the pattern, and provisions the requested environments and middlware components, by invoking the corresponding service.
Configure Pattern
Build Pattern
Pattern Binaries
Provision
Provision Environments & Middleware
Configuration
Deploy
Deploy Pattern
Provisioning Service
Deployment Service
Add Features
Bind Variability Points
Already Provisioned
Add Feature

Figure 4.4: BPMN Representation - The Pattern Usage Process.

Note: The data exchange between the *provision environments & middleware* process and the corresponding *provisioning service* is handled by passing a reference to the required configuration data. This reference is stored within the exchanged message. The provisioning service then interprets this reference and loads the addressed configuration from the workspace. This is analogously done for the *deploy pattern process* and the *deployment service*.

4.5 Functional Project Structure Model

This section provides the *functional project structure model*, required for the functional architecture, outlined in Section 4.3. Each pattern implementation, located in the *workspace* (see Figure 4.2), is structured according to this model. This allows to decouple required patterns, of aggregated pattern-based applications, from their implementations. Thus, contained entities and their interrelations are introduced. The presented concept allows to initialize an abstract application archetype and to plug in pattern implementations afterwards, in order to instantiate individual, concrete applications. This concept allows to instantiate so-called *application skeletons*, which can be enriched by concrete pattern implementations afterwards. This allows developers, using the pattern development process, presented in Section 4.4, to benefit from a wide product space of pattern-based applications.

The *functional project structure model* and the thereby enabled *application skeleton* concept are introduced in the following.

The Functional Project Structure Model
The following entities and relations exist:

- *Project* - The *project* entity forms an abstract base entity for projects. It contains a Maven *Project Object Model*, the *pom.xml*; and a *variability descriptor*, the *vd.xml*.
  The following project subtypes exist:
  - *Abstract Project* - An *abstract project* contains configuration information about referenced *implementation projects*, this information is kept within its variability descriptor. Referenced *implementation projects* implement a *common interface*, which defines a common semantic for all referenced implementation projects.
    Subtypes of *abstract projects* are:
    - *Root Project* - A *root project* is an *abstract project.* It forms the root of every project structure and contains arbitrary sub-projects.
    - *Sub-Project* - A *sub-project* is an *abstract project.* It is a child project of a root project or another sub project and provides means to structure the workspace. *Leave sub-projects* reference so-called *implementation projects.*
  - *Implementation Project* - An *implementation project* can be a *custom component* project or a so-called *commercial off-the-shelf* project. It contains *provisioning scripts* and offers implementations for required features.
    - *Custom Component* - A *custom component* hosts Java-based code, which is required to implement features of its related pattern.
    - *Commercial off-the-shelf* - A *commercial off-the-shelf* - also called *COTS* - is a pre-build component, which has to be configured and provisioned in order to implement the features of the corresponding pattern. Such a COTS might be a database, queue or any other arbitrary middleware component.

- *Project Object Model* - The Maven *Project Object Model* defines possible sub projects and build specific configurations [3].
- *Variability Descriptor* - A *variability descriptor* is a file, which hosts the variability model of the the corresponding project [4].
- *Provisioning Script* - A *provisioning script* contains provisioning code, to provision nodes and install required middleware stacks on-top.

Figure 4.5 illustrates the functional project structure model and its interrelations.

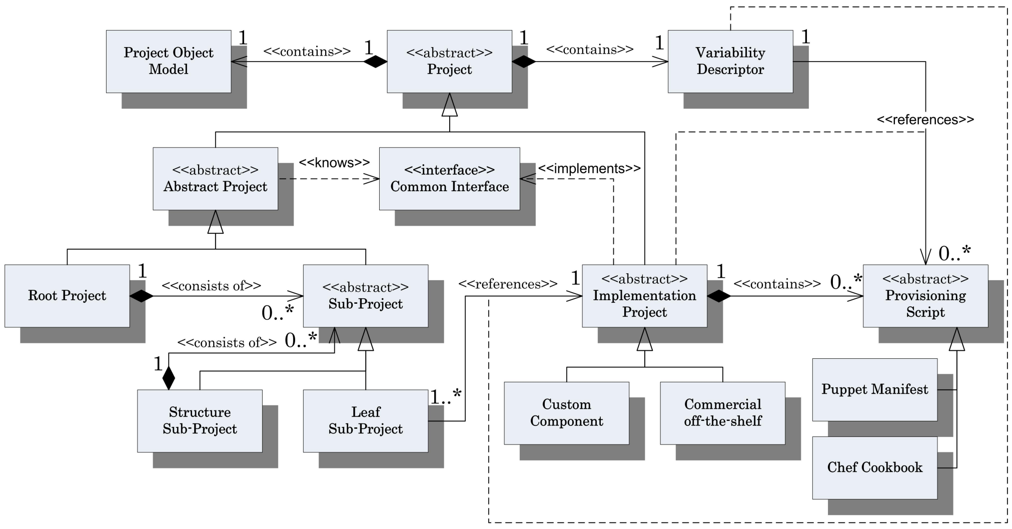


Figure 4.5: Class Diagram - The Functional Project Structure Model.

The Application Skeleton Concept

The presented project structure model enables the possibility to offer so-called *application skeletons*, which allow to plug in and switch implementations, since it provides unified slots for pattern implementations, enabled through *commonly known interfaces.* These skeletons can be stored in the *pattern template catalogue* to enable a variety of possible, individual applications. From there, each skeleton can be instantiated by a *workspace-user* into his workspace. Then, the *workspace-user* selects the offered implementation patterns from the offered variability descriptors that are located within the *abstract projects* of the *application skeleton.* The *workspace-user* then instantiates the selected patterns into his workspace and performs the configuration for his newly created application. After that, the *workspace-user* compiles the application into binaries. Finally, the workspace-plugin goals, to provision the environments and deploy the binaries, using the previously made configurations, are invoked. Figure 4.6 illustrates the required steps, to initialize a *skeleton application* into an individual application.

[3] The Maven *Project Object Model* is discussed in Related Works Section 3.3.4

[4] The concept of variability descriptors is outlined in Section 3.2.1 and refined in the subsequent Section 4.6.

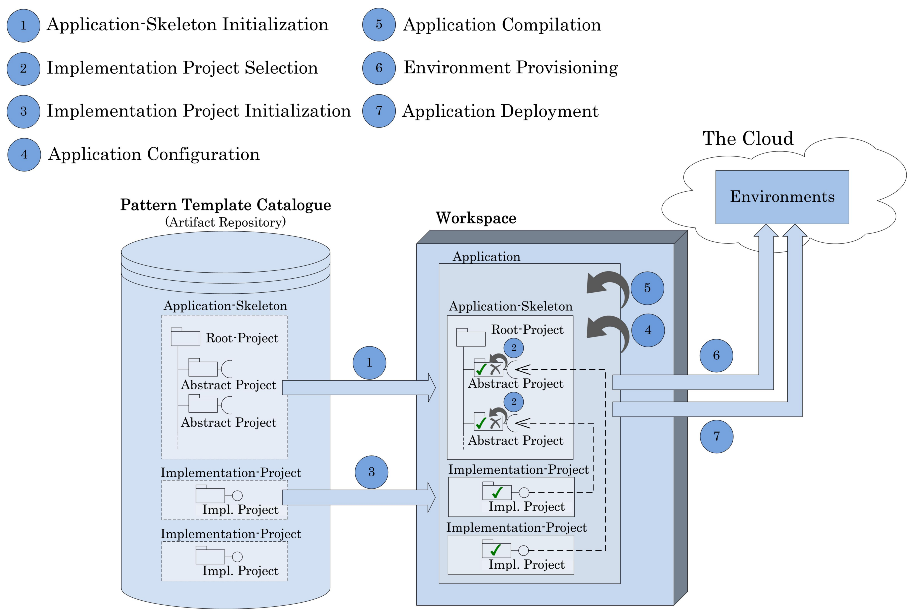

1 Application-Skeleton Initialization
2 Implementation Project Selection
3 Implementation Project Initialization
4 Application Configuration
5 Application Compilation
6 Environment Provisioning
7 Application Deployment
The Cloud
Environments
Pattern Template Catalogue
(Artifact Repository)
Workspace
Application
Application-Skeleton
Root-Project
Abstract Project
Implementation-Project
Impl. Project

Figure 4.6: Example - Skeleton Instantiation, Configuration, Provisioning and Deployment.

4.6 Functional Variability Meta-Model

Throughout this section, the functional variability meta-model, based on the variability model of Mietzner - discussed in Related Works Section 3.2.1 - is presented. The *functional project structure model*, introduced in Section 4.5, requires each pattern *project* to contain a variability descriptor, which hosts its *variability model*, respectively its *variability points*. This model is deserialized, interpreted and its defined actions are executed by the *workspace-plugin*, when tasks are requested by the *workspace-users*. Thus, this section presents the modifications made to support the OVM Notation [PBL05, MPH$^+$ct], as well as the extensions required to support the functionalities specified in Section 4.2. Afterwards, the OVM Notation, which is used to represent variability graphs, is presented.

The variability model, presented by Mietzner [Mie08], is stored in a single variability descriptor, pointing into the corresponding application template. This approach lacks the possibility to distribute the variability model among the components that form the entire application. Yet, this distribution is required, as the concrete combination of pattern implementations is not known, since the final pattern artifact has been added to the overall solution. Therefore, several modifications had been made to support the distributed definition of variability and corresponding dependencies.

The following entities have been added to the model of Mietzner [Mie08]:

- *Script Locator* - Script locators point to scripts, nested in the developer-workspace. Each script locator defines, which symbol indicates a variability location within the referenced script. These parts of the document are then replaced later on by the workspace plugin by their corresponding values. Referenced scripts may be *Puppet Manifests*, *Chef Cookbooks* or arbitrary files, which are executed afterwards.

- *Alternative Cardinalities* - Alternative Cardinalities define a selection semantic, by defining a minimum and maximum amount of alternatives, that have to be selected.
- *Neutral Alternative* - Neutral alternatives offer the possibility to reference variability points, respectively alternatives, without pushing any value into any variability location.
- *Locator Alternative* - The locator alternative entity forms the base for all alternatives that reference variability locations. The values, defined within these alternatives, are later on pushed to their referenced variability locations.
- *Property Alternative* - The property alternative entity extends the locator alternative entity. It reads a property value from a defined properties file and pushes it into its referenced variability location.
- *Implementation Alternative* - Implementation alternatives are offered within the variability descriptors of *abstract projects.* They store required meta-information about the location of referenced implementation projects.

The following entities have been modified:

- *Dependency* - Dependencies are classified into *require* and *exclude* dependencies. *Require* indicates, that the referenced entity is required for the source of the reference. *Exclude* indicates, that the referenced entity has to be disabled, when the source of the reference is enabled.
- *Free Alternative* - The entity *free alternative* of Mietzner's Model has been renamed into *String Alternative*, to indicate that a string value is inserted.
- *Explicit Alternative* - The entity *explicit alternative* of Mietzner's Model has been renamed into *XML Alternative*, to indicate that an arbitrary xml value is inserted into the referenced variability location.
- *Variability Point/Alternative* - *Locators* are hosted within *alternatives*, this allows the definition of abstract variability points and arbitrary alternative combinations.

The following entities have been excluded:

- *Locator-based Alternative* - Locator-based alternatives have been excluded, as default values are indicated by a default-flag attribute, nested within *locator alternatives.*
- *Empty Alternative* - Empty alternatives have been excluded, as optional choices are indicated by so-called alternative-cardinalities, required by the OVM Notation.
- *Expression Alternative* - Expression alternatives have been excluded, as values determined for variability locations are directly defined in the variability descriptor itself.

The following relations have been modified:

- *Variability Model to Variability Model* - To support a variability model hierarchy, each variability model references a potential set of child variability models.
- *Alternative to Variability Point* - Alternatives may reference a variability point via a dependency. A *require* dependency indicates, that an alternative of the referenced variability point has to be selected. An *exclude* dependency indicates, that the referenced variability point has to be disabled and thus, none of its nested alternatives is allowed to be selected.
- *Variability Point to Alternative* - An enabled variability point may directly reference an alternative, to indicate that the referenced alternative has to be selected, in case of a *require* dependency, respectively disabled, in case of an *exclude* dependency.

The following status attributes have been added:

- *Enabled Status* - An *enabled status* has been added to the variability point entity, to indicate whether the variability point has been enabled, due to an enabling condition.
- *Selected Status* - A *selected status* has been added to the alternative entity, to indicate whether the alternative has been selected or not.

Figure 3.1 illustrates the modified variability meta-model, based on the work of Mietzner [Mie08] and the entities, added to support the requirements, stated in Subsection 4.2.

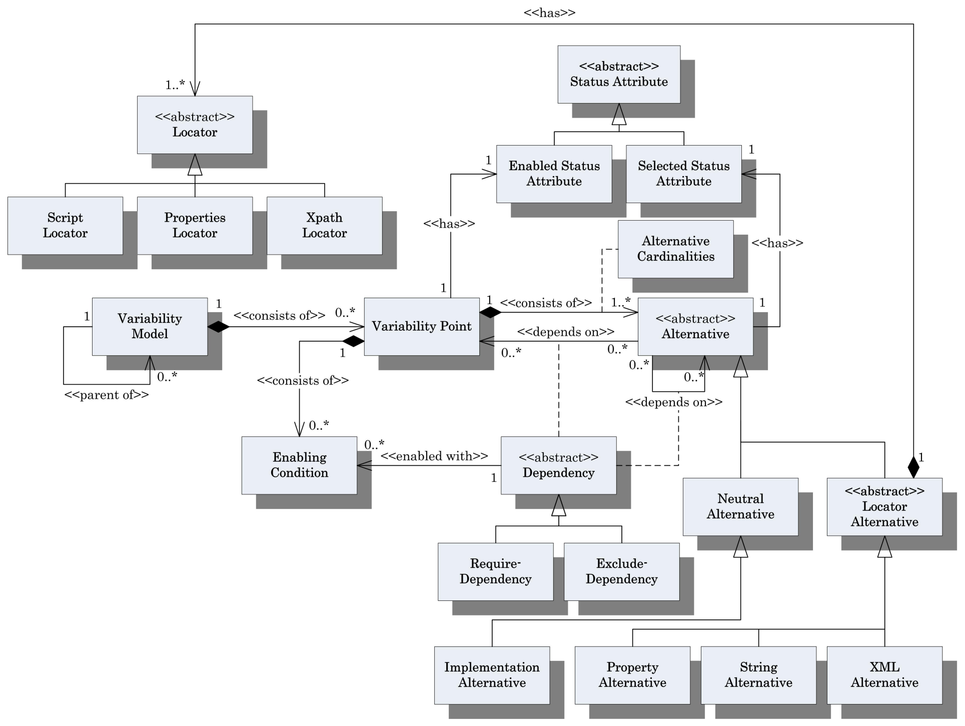


Figure 4.7: Class Diagram - The Functional Variability Meta-Model and its Interrelations.

The adapted version of the *OVM Notation*, supporting the previously introduced meta-model, is briefly introduced in the following paragraph, as it can be used to illustrate single variability points or the entire variability graph of an aggregated pattern-based application.

The OVM Notation
In the OVM Notation [PBL05, MPH$^+$ct], *variability points* are illustrated as triangles, storing references to *alternatives*. At the top, of each variability point, the corresponding name attribute is located. The status of a variability point, as well as its description, is located in its center. At the bottom, alternative place holders are nested, which reference the alternatives, assigned to the current variability point. Below the variability point triangle, a curve, crossing all referenced alternatives, is placed, which illustrates the alternative cardinalities, and, thus, the alternative selection semantic. This curve is associated with the corresponding defined minimum and maximum.
Alternatives are illustrated as boxes, which, again, contain boxes that store alternative values. Alternative names are stored in register tabs at the top of each alternative box. Corresponding *attributes* are located in alternatives at segregated boxes.
Locators, stored in an alternative, are stored in separate boxes, nested within the corresponding alternative. Each alternative may depend on a variability point or an other alternative. These dependencies are modeled with arrows, pointing from the source alternative to its target. Each dependency is labeled with its corresponding type.

Note: Appendix Figure B.1 illustrates an exemplary variability model, instantiated form the recently presented meta-model, using the adapted version of the OVM Notation.

4.7 Formal Variability Graph Model

Section 4.6 introduced the functional variability meta-model, stored within each pattern. This model allows to define variability dependencies and, thus, to define a so-called *variability graph*. The pattern configuration sequence is implicitly stored within this graph, as modeled dependencies depict, which components have to be executed before others. These dependencies guarantee, that flow-generated configuration values are present before referencing components are configured. Thus, this section formally introduces the *variability graph*, to illustrate its graph-nature. Therefore, its corresponding canonical and inverted forms are introduced, which are combined, during the pattern configuration flow computation, presented in Section 4.8.

Formal Definition
Each pattern artifact defines its configuration dependencies by its local variability model, forming a *variability graph*. These variability graphs can reference nodes, defined within their corresponding variability model, as well as nodes, defined in external variability models. Hence, these variability graphs can be combined into a global, resulting one, which defines the customization semantic of the overall, aggregated pattern. This graph serves as input for the configuration process, to compute the pattern configuration flow. Thus, the following paragraphs formally define the variability graph, its corresponding canonical as well as its inverted form.

Definition: A *variability graph G* is a *directed, acyclic graph*, consisting of a set of *variability points V*, defining child alternatives of the *alternative* set *A*; and *dependencies D*, referencing *variability points* or other *alternatives.*

Thus, a variability graph is defined as:

$G = (V, A, D)$

$A = \{a \mid a\ is\ an\ alternative\ of\ G\}$

$V = \{v \mid v\ is\ a\ variability\ point\ of\ G,\ v\ has\ child\ alternatives\ V_{children} \subset A\ \}$

$D \subseteq ((A \times A) \cup (A \times V))$

Each variability graph fulfills the acyclic graph condition:

$C = \{(a_1, ..., a_n) \in A^n \mid \exists n \in \mathrm{N} : a_1 = a_n\} = \emptyset$

In case that a variability point is referenced by an alternative, all selected child alternatives have to be executed during the configuration process[5]. These dependencies can be replaced by multiple references, which results in forming a *canonical variability graph.*

Definition: The *canonical variability graph* G^* of G consists of all alternatives and variability points defined in G; and the set of transformed dependencies D^*.

Thus, the canonical variability graph is defined as:

$G^* = (V, A, D^*)$

$D^* \subseteq (A \times A)$

$D^* = D_\Delta \cup \{(a_1, a_2) \in (A \times A) \mid \exists(a_j, v_i) \in (A \times V) \wedge a_j = a_1 \wedge a_2\ is\ child\ of\ v_i\}$ [6]

$D_\Delta = D \backslash \{d \mid d \in (A \times V)\}$

Inverting a variability graph G results in inverting its dependencies D. Yet, this operation is only defined for dependencies of the type $d \in (A \times A)$, while inverting a dependency of the type $d \in (A \times V)$ is undefined, as the resulting dependency $d^{-1} \in (V \times A)$ violates the variability graph dependency definition. However, inverting a canonical variability graph always results in a defined, valid variability graph, as canonical variability graphs only consists of $d \in (A \times A)$ dependencies.

Definition: The *inverted variability graph* G^* of G consists of all alternatives and variability points defined in G; and the set of dependencies D^{-1}, that are generated by the so-called *graph inverter operator* $f^{-1}(G)$.

Therefore the inverted variability graph and the graph inverter operator are defined as:

$G^{-1} = (V, A, D^{-1})$

$D^{-1} = \{(a_1, a_2) \in (A \times A) \mid (a_2, a_1) \in D\}$

$$f^{-1}(G) = \begin{cases} G^{-1}, & \text{if } \forall d \in D : d \in (A \times A) \wedge d \notin (A \times V), \\ undefined, & \text{otherwise.} \end{cases}$$

[5]This assumes, that all required variability points have been bound and the selected alternatives fulfill the conditions, defined by their corresponding selection cardinalities.

[6]Dependencies in G, defined between alternatives and variability points, are replaced, by multiple dependencies between the originated alternative and related child alternatives.

4.8 The Configuration Flow Computation Algorithm

Generating an execution sequence out of a *dependency graph* is a well known problem constellation and relates to the widely known *topological sorting* algorithm; documented by Knuth [Knu98]. Therefore, to transform the problem constellation of dependent configuration entries into a topological sorting problem, the graph dependencies are inverted. However, a normalization procedure, creating the previously introduced canonical form, is applied first, to guarantee that all dependencies can be inverted. Thus, the configuration flow is computed according to:

$$\mathit{ConfigurationFlow}\ (G) = \mathit{InvertedTopsort}(G^{*}) = \mathit{Topsort}((G^{*})^{-1}\ )$$

Figure 4.8 provides an exemplary overview of the executed steps, using the OVM Notation.

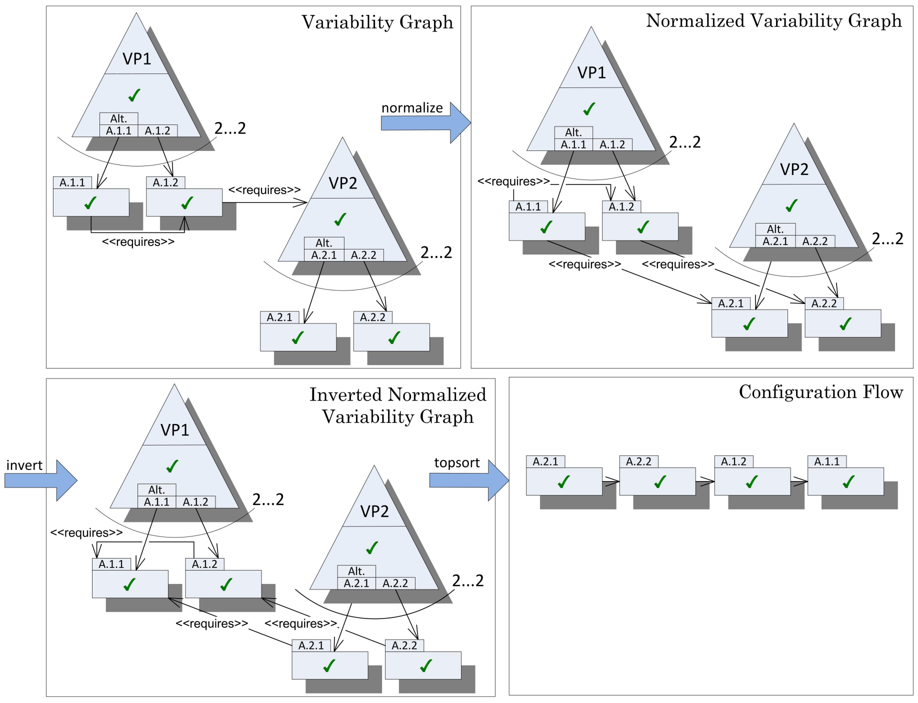


Figure 4.8: Overview - The Configuration Flow Computation Steps

Complexity

The presented algorithm, illustrated in Algorithm 4.1 is has linear time complexity of $O(2|D| + |A|)$, as the variability graph normalization and inverting is performed in $O(|D|)$ and the topological sorting is performed within $O(D| + |A|)$.

Algorithm 4.1 illustrates a pseudo algorithm, performing the required computation steps. First, the inverted normalized variability graph is computed, by replacing $(A \times A)$ dependencies with their inverted ones and $(A \times V)$ dependencies with dependencies starting at the corresponding child alternatives, pointing to the originated alternative. Afterwards, a topological sorting is performed on the set of alternatives and their corresponding dependencies.

Algorithmus 4.1 The Configuration Flow Computation Algorithm.

```
procedure ComputeConfigurationFlow($V, A, D$)
// Normalize & Invert
    for all $d \in D$ do
        if $d \in (A \times V)$ then
            for all $a \in d.target.alternatives$ do
                a.addDependencyTo(d.source)
            end for
        else if $d \in (A \times A)$ then
            d.target.addDependencyTo(d.source)
        end if
        d.source.removeDependencyTo(d.target)
    end for
// Topsort
    result $\leftarrow \emptyset$
    workingSet $\leftarrow \{a \in A \mid \#a.incomingDependencies = 0\}$
    while $workingSet.size > 0$ do
        current $\leftarrow workingSet.first$
        result $\leftarrow$ result $\cup$ current
        for all $d \in current.outgoingDependencies$ do
            d.source.removeDependencyTo(d.target)
            if $\#d.target.incomingDependencies = 0$ then
                $workingSet.add(d.target)$
            end if
        end for
        workingSet.remove(current);
    end while
    if $workingSet.size > 0$ then
        $Error - Case : Cyclic\ dependencies\ exist.$
    end if
     return result
end procedure
```

Note: Algorithm 4.1 is based on the provisioning flow computation algorithm proposed by Mietzner [Mie10]. Yet, the normalization procedure has been added, since the presented variability model of this thesis is an adapted model of Mietzner's variability model.

4.9 Functional Workspace Agent Components

This section provides a black box view of the required components for the *workspace agent*, respectively the *maven workspace plugin* - introduced in Section 4.3 Figure 4.2. The workspace agent implements the requirements presented in Section 4.2. Maven plugins consists of *goals*, each use case, described in Use Case Appendix A, is associated with such a goal. The required functional components, their interrelations and goals are outlined briefly in the following section, as they form the foundation for the later implemented technical architecture[7] of this diploma thesis.

The workspace agent, respectively the maven workspace plugin, consists of the following components:

- *Controller Component* - The controller component offers the plugin goals to external users. It coordinates incoming requests and dispatches it to the corresponding goal-implementation components.
- *Archetype Generation Component* - Implements use case A.1.
- *Archetype Persistence Component* - Implements use case A.2.
- *Archetype Initialization Component* - Implements use cases A.3 and A.4.
- *Pattern Configuration Component* - Implements use case A.5.
- *Environment Provisioning Component* - Implements use case A.6.
- *Pattern Deployment Component* - Implements use case A.7.
- *Pattern Undeploy Component* - The undeployment component implements the removal of deployed artifacts, using the same mechanisms like use case A.7.
- *Environment Decommissioning Component* - The environment decommissioning component releases acquired resources, using the same mechanisms like use case A.6.

Figure 4.9 illustrates the components of the *workspace agent*, respectively of the maven workspace plugin. The components are ordered (left to right) according to their usual execution order within the pattern development process, discussed in Section 4.4.

[7]The technical architecture is later on presented in Chapter 5 - *Design and Implementation*

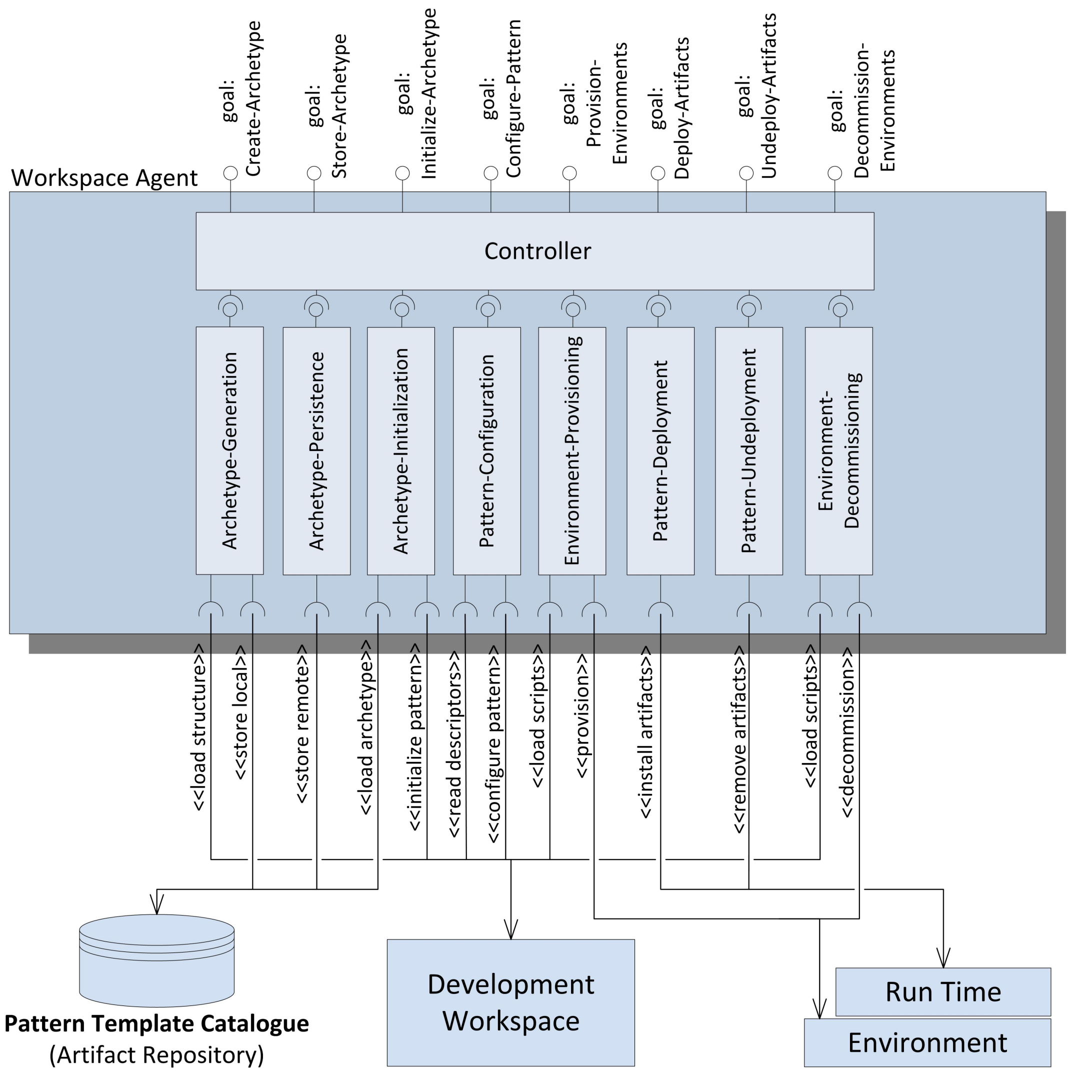


Figure 4.9: Component Diagram - Functional Workspace Agent Components.

Design and Implementation

This chapter provides detailed design and implementation information about the developed *maven workspace plugin*, which implements the presented *workspace agent*, introduced in Section 4.9. The realized plugin goals implement the use cases, defined in Use Case Appendix A, to support the *pattern development process*, presented in Section 4.4. Moreover, implemented *cloud computing patterns* are presented, as they can be used by the developed *maven workspace plugin* to create, store and configure patterns. Besides, the implemented concept allows to provision related pattern environments and to deploy aggregated pattern artifacts to them. This provisioning can be handled by: pre-implemented services, by self-defined provisioning actions or by any other provisioning action that is bound to the pattern.

First, Section 5.1 discusses the used *abstract reference architecture*, used in this thesis. This architecture defines best practices about structuring an application and its corresponding layers. It is presented, as the implemented *maven workspace plugin* is structured according to this reference architecture. Furthermore, later on presented *reference pattern implementations*, which can be managed with the implemented *maven workspace plugin*, are also implemented according to this reference architecture. Afterwards, Section 5.2 covers technical design decisions of the implemented maven workspace plugin. Thus, components, their tasks and interrelations are presented in Subsection 5.2.1. Subsection 5.2.2 then covers the so-called *pattern goal invocation flow* which propagates abstract events along the pattern project structure and triggers concrete handler actions, defined in the corresponding patterns. Subsection 5.2.3 then illustrates the implementation of such handlers, within the *pom.xml* files of corresponding patterns. Subsection 5.2.5 provides graphical extracts of the variability model xml schema, which implements the *functional variability meta-model*, presented in Section 4.6. Subsection 5.2.4 then presents the syntax of the implemented workspace plugin goals, along with their invocation possibilities. Then, Section 5.3, presents technologies and tools, which have been used during the development of the implemented reference patterns and the *maven workspace plugin*. Finally, Section 5.4 outlines implemented reference patterns in detail, i.e., the *Three-Tier Cloud Application* pattern, discussed in Subsection 5.4.1, and the *Message-Oriented Middleware* pattern, discussed in Subsection 5.4.2, which can be managed by the implemented approach.

5.1 Abstract Reference Architecture

The *software product line*, presented in Section 4.1, requires a reference architecture, which is obligatory for all created patterns. This abstract architecture enables *loose coupling* between application layers, by forcing the usage of interfaces. Thus, it allows the exchange of implementation patterns, without any modification of dependent code. Due to its nature, this architecture can be applied on arbitrary applications. Thus, relevant parts of the presented reference architecture are used in Subsection 5.2.1, to implement the *maven workspace plugin*. Section 5.4 then uses the presented reference architecture to implement reference patterns, serving as artifacts of the presented *software product line*, introduced in Section 4.1. This reference architecture, along with its corresponding layers, is introduced in the following.

The different layers are:

- *Presentation Layer* - The *presentation layer* dispatches incoming requests to the internal representation of the application. It decouples request processing technologies from internally used processing technologies.
- *Process Layer* - The *process layer* offers a superordinate controller, which is responsible for coordinating request processing. It dispatches different requests to their processing components, monitors requester access rights and handles exceptions. Moreover, it decouples the orchestration of request processing from concrete component implementations.
- *Business Logic Layer* - The *business logic layer* hosts implementations of business use cases. Each component is responsible for a single use case and may access data or external systems via the *data layer*.
- *Data Layer* - The *data layer* offers a transparent view on data, by providing an API to query data objects. It decouples technology specific data representation from their internal object representation, by so-called *Data Transport Objects* (DTOs) [1].
- *Integration Layer* - The *integration layer* implements the required actions to interact with external systems, like databases, message queues or web services, it dispatches requests via adapters to middleware service providers.
- *Access Control Layer* - The *access control layer* monitors the privileges of requesters, by allowing or denying requested operations, according to defined permission rules.
- *Common* - *Common* offers commonly shared functionality between all layers, such functionality might be: object definitions, converters or validation components.

Figure 5.1 illustrates the layered software product line reference architecture, which implements all of the above mentioned layers. Each dependency between the offered layers is decoupled by interfaces. This abstraction allows to switch layers without effecting layers that dependent on the switched one.

[1] A Data Transport-Object is a design pattern, which is widely used in the JavaEE design community.

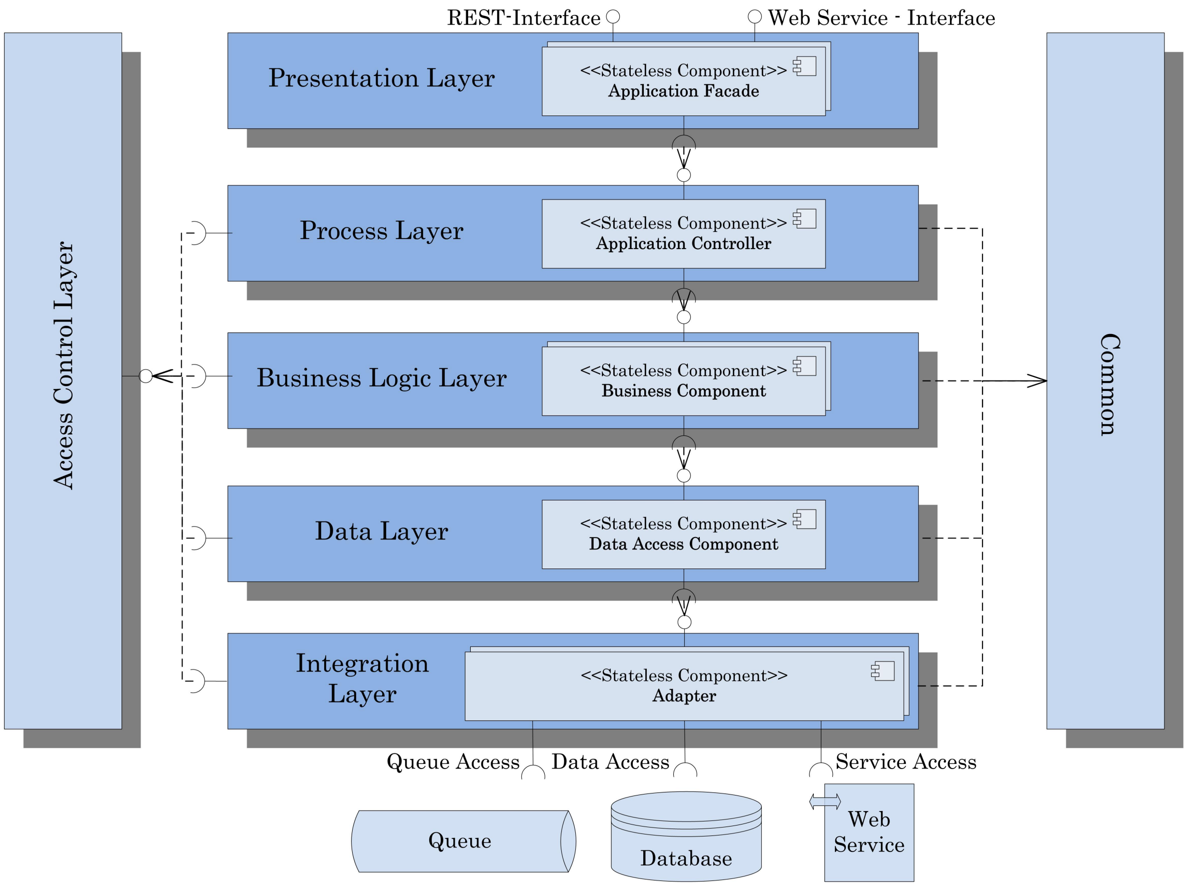


Figure 5.1: Reference Architecture of the Software Product Line for Cloud Patterns.

5.2 Maven Workspace Plugin Design

This section provides design and implementation details about the *maven workspace plugin*, which implements the concepts defined in Chapter 4. The plugin provides means to invoke several actions on a pattern, stored within the developer workspace. The components, implementing these actions, are discussed in detail, as well as their interactions. Subsection 5.2.1 covers the implemented technical components of the workspace plugin and illustrates these components, by providing a component diagram. Subsection 5.2.2 covers the abstract project invocation flow that results, when a plugin goal is invoked on the root-project of a pattern. This flow is created by propagating the invoked goal to all reachable child projects. Bound actions are then executed by the invoked pattern projects. Thus, subsection 5.2.3 covers the technical details, which implement this behavior. Finally, subsection 5.2.5 provides graphical extracts of the implemented variability model xml schema, which forms the foundation for all implemented variability descriptors.

5.2.1 Technical Components

This subsection provides a glass box view of the implemented maven workspace plugin, specified in *Concept and Specification Chapter* Section 4.9. Implemented components, their tasks and their interrelations are discussed and are presented, afterwards, in Figure 5.2.

The following components have been implemented:

- *Application Access Layer* - The *application access layer* receives incoming plugin requests, it consists of multiple, so-called *Maven Mojo facade* components.
    - *Maven Mojo Facade* - A *Maven Mojo Facade* offers maven plugin goals and dispatches the incoming invocations to the *process layer.*
- *Process Layer* - The *process layer* receives incoming plugin goal requests and coordinates their execution. It consists of the *controller facade* and the *plugin-goal controller.*
    - *Controller Facade* - The *controller facade* dispatches incoming goal requests to their corresponding *plugin-goal controllers.*
    - *Plugin-goal Controller* - A *plugin-goal controller* coordinates the execution of a single plugin goal, by invoking the corresponding implementation components, nested within the *business logic layer.*
- *Business Logic Layer* - The *business logic layer* offers goal implementations via an interface to the *process layer.* It consists of *goal-implementations*, the *variability graph merger*, the *variability graph sorter* and the *customization engine.*
    - *Goal-Implementations* - A goal-implementation implements a use case specified in Use Case Appendix A. These implementations require various sub use cases, like merging multiple variability graphs into a single result, performing an adapted topological sorting on the selected alternatives and pushing selected values into their destinations.
    - *Variability Graph Merger* - The *variability graph merger* joins multiple variability graphs into a single one and replaces soft references by hard object references.
    - *Variability Graph Sorter* - The *variability graph sorter* receives a variability graph as input and computes the configuration sequence of alternatives that have to be executed to perform the configuration goal. This sequence is computed in such a way that for each alternative, which is referencing another one, is guaranteed that the referenced alternative is configured first. Thus, the *variability graph top sorter* component inverts the edges of the variability graph, by using the *variability graph inverter* component, and then performs the topological sorting [2]. This guarantees that the leafs of the original graph are executed last.

[2]These steps are introduced in detail in Section 4.7.

  - *Customization Engine* - The *customization engine* receives a sequence of selected alternatives and interprets their defined values, by resolving their values. These values are then pushed into the locations, defined within the locators of the current alternative.

- *Integration Layer* - The *integration layer* provide means to integrate external infrastructure tools, like *Opscode's Chef*, *PuppetLabs' Puppet* or any other technology, capable of provisioning environments. This layer is implemented across *patterns* and the *workspace plugin*, as the concrete actions that have to be performed, are defined within the pattern itself. Thus, the workspace plugin only propagates an abstract goal along the project path invocation[3].
  - *Infrastructure Tool Plugins* - *Infrastructure tool plugins* perform the tasks, required to provision environments and middleware. These actions are also encapsulated within maven plugins and are bound to concrete pattern implementations.
- *Data Access Layer* - The *data access layer* provides means to access variability descriptors and arbitrary files, nested within the developer workspace. It consists of the *file access facade* and various *file manipulators*.
  - *File Access Facade* - The *file access facade* offers the possibility to manipulate files, to components of the *business logic layer*. It dispatches the calls to the corresponding *file manipulators*.
  - *File Manipulators* - A *file manipulator* accesses a file in the workspace. Two types of *file manipulators* exist: The *xml file manipulator*, which manipulates xml files; and the *properties-file manipulator*, which manipulates properties file.
- *Common* - *Common* offers commonly used functionalities to all components of the workspace plugin. It consists of the *variability descriptor converter*, the *XPathEngine*, several *validators* and common *object definitions*.
  - *Variability Descriptor Converter* - The *variability descriptor converter* receives a *DOM* representation of a variability descriptor and converts it, into the workspace plugin specific, internal object representation; the *variability graph object*.
  - *XPathEngine* - The *XPathEngine* component offers the possibility to query *Document Object Model* (DOM) nodes in the passed *DOM* object [4].
  - *Validators* - A *validator* component receives a *variability graph* and performs a single validation goal on it, i.e, check cyclic dependencies, check self references, check contradictions and check selection cardinalities.
  - *Object Definitions* - The objects defined within the *object definitions* component are commonly used by all components of the workspace plugin, and thus, form the foundation for data exchange between components.

[3]This methodology is discussed in greater detail in section 5.2.2

[4]This component is mainly used by the *variability descriptor converter* to create the *variability graph*.

Figure 5.2 illustrates the implemented plugin components and their relations. Each subsystem and component is implemented according to the presented reference architecture, defined within section 5.1, to guarantee loose coupling between the different plugin layers.

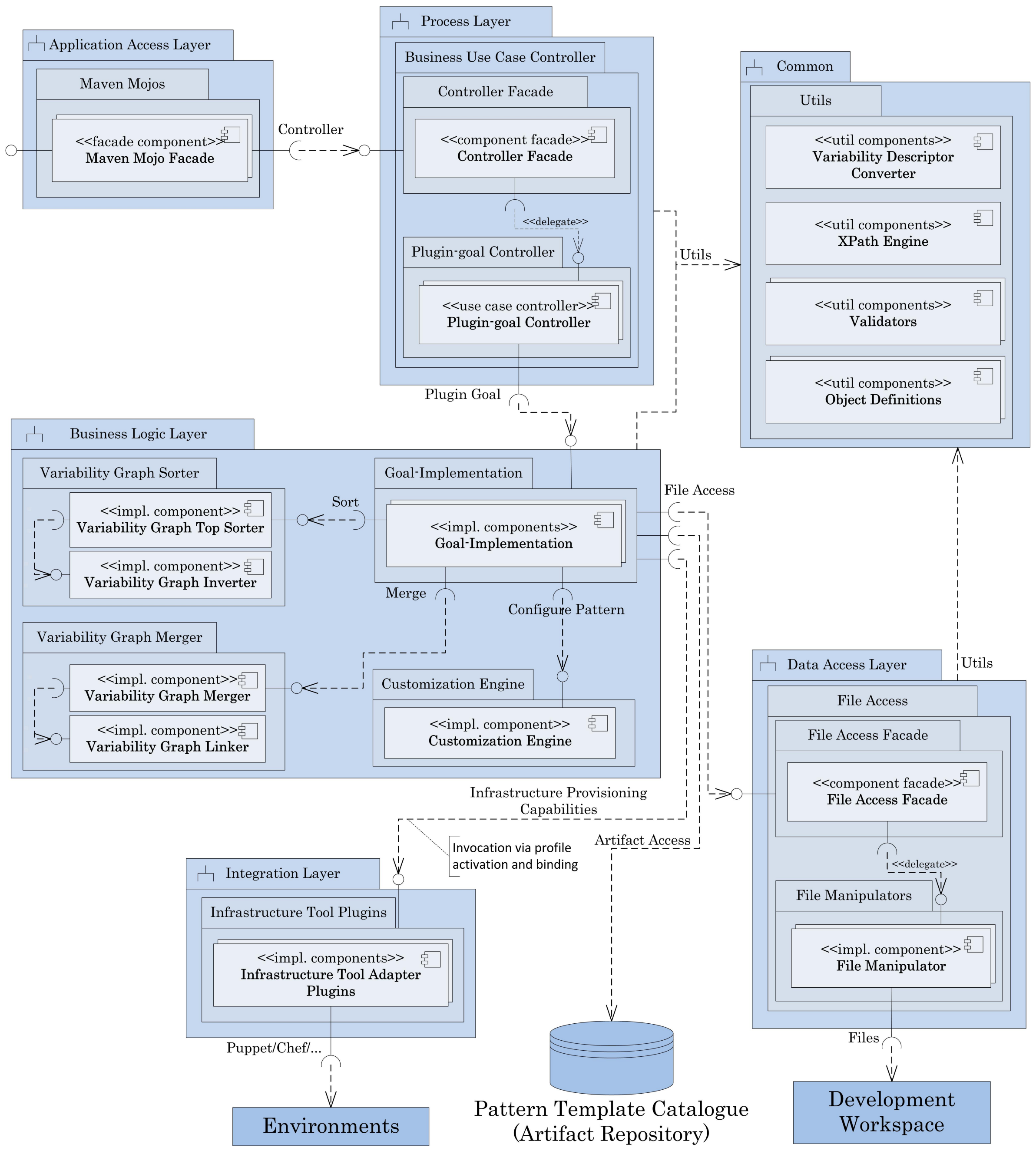

Application Access Layer
Maven Mojos
<<facade component>> Maven Mojo Facade
Controller
Process Layer
Business Use Case Controller
Controller Facade
<<component facade>> Controller Facade
<<delegate>>
Plugin-goal Controller
<<use case controller>> Plugin-goal Controller
Plugin Goal
Utils
Common
Utils
<<util components>> Variability Descriptor Converter
<<util components>> XPath Engine
<<util components>> Validators
<<util components>> Object Definitions
Business Logic Layer
Variability Graph Sorter
<<impl. component>> Variability Graph Top Sorter
<<impl. component>> Variability Graph Inverter
Sort
Goal-Implementation
<<impl. components>> Goal-Implementation
File Access
Merge
Configure Pattern
Variability Graph Merger
<<impl. component>> Variability Graph Merger
<<impl. component>> Variability Graph Linker
Customization Engine
<<impl. component>> Customization Engine
Data Access Layer
Utils
File Access
File Access Facade
<<component facade>> File Access Facade
<<delegate>>
File Manipulators
<<impl. component>> File Manipulator
Files
Development Workspace
Infrastructure Provisioning Capabilities
Invocation via profile activation and binding
Artifact Access
Integration Layer
Infrastructure Tool Plugins
<<impl. components>> Infrastructure Tool Adapter Plugins
Puppet/Chef/...
Environments
Pattern Template Catalogue (Artifact Repository)

Figure 5.2: Component Diagram - Technical Components of the Maven Workspace Plugin.

5.2.2 The Pattern Goal Invocation Flow

This subsection discusses the pattern goal invocation flow, which is executed each time a workspace plugin goal is invoked on a Maven-based pattern project. The invocation flow then propagates the goal execution within the workspace. This invocation is implemented by extending the maven goal propagation flow and is used to propagate the invocation of goals, presented in Section 4.9. Thus, the following paragraphs discuss the maven propagation flow in combination with the implemented workspace plugin.

The Project Invocation Propagation Mechanism
When a workspace plugin goal is executed on an aggregated, Maven-based pattern, each sub project of the pattern has to execute the goal as well, in order to guarantee the invocation semantic. For this reason, the pre-implemented maven default propagation mechanism is used, to propagate plugin goal invocations to sub projects within the workspace. The standard maven propagation mechanism propagates the execution of the current goal to all reachable child projects within the workspace, this propagation mechanism is leveraged, to invoke actions on the patterns, hosted within the developer workspace.

Figure 5.3 illustrates the direct invocation of the workspace plugin. The plugin triggers the default maven build life-cycle and activates an assigned, custom maven profile [5]. Maven then propagates this goal to all child projects, by enabling the corresponding profile in each project. Each project, that has defined such a profile handler then executes the bound actions.

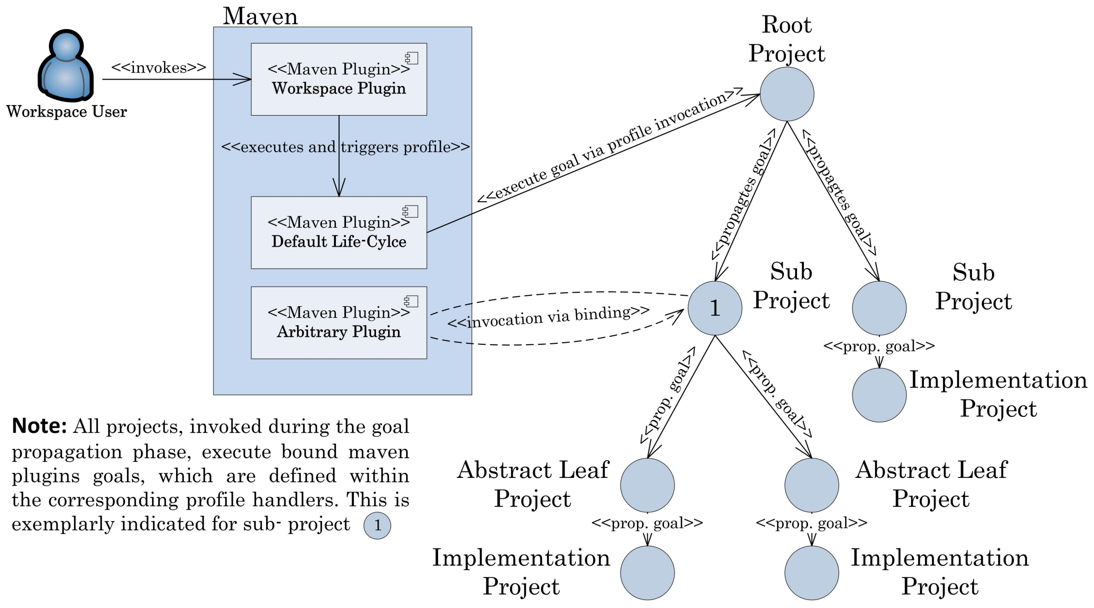

Maven
Workspace User
<<invokes>>
<<Maven Plugin>> Workspace Plugin
<<executes and triggers profile>>
<<Maven Plugin>> Default Life-Cylce
<<Maven Plugin>> Arbitrary Plugin
<<execute goal via profile invocation>>
<<invocation via binding>>
Root Project
<<propagtes goal>>
<<propagtes goal>>
1
Sub Project
Sub Project
<<prop. goal>>
Implementation Project
<<prop. goal>>
<<prop. goal>>
Abstract Leaf Project
Abstract Leaf Project
<<prop. goal>>
<<prop. goal>>
Implementation Project
Implementation Project
Note: All projects, invoked during the goal propagation phase, execute bound maven plugins goals, which are defined within the corresponding profile handlers. This is exemplarly indicated for sub- project 1

Figure 5.3: Overview - The Project Invocation Propagation Mechanism.

[5]Introduced in Related Works Section 3.3.2 and 3.3.5.

5.2.3 Pattern Defined Action Invocation

Subsection 5.2.2 presented the abstract *pattern goal invocation flow*. This flow is refined by each pattern, by reacting on propagated goals and performing bound actions. This pattern defined action invocation is discussed in the following.

The execution of the workspace plugin goals *configure*, *provision*, *deploy* and *undeploy* are bound, in each implementation pattern, to standard maven phases. This defines a custom event handling on received goals. Bound actions are encapsulated within maven plugins and are invoked when the pattern receives the corresponding goal, via the introduced *pattern goal invocation flow*. Therefore, each goal handler, of an implementation pattern, is defined in a separate maven profile, nested within the pattern's *pom.xml*. When the implementation pattern receives the request to execute such a goal, it sequentially executes the bound plugins. This binding process allows to integrate arbitrary systems with the presented approach, as arbitrary code can be wrapped within a maven plugin. Thus, patterns can address already defined provisioning services, offered by PaaS provisioning services, like *AWS Elastic Beanstalk*, or they can build their own provisioning service and invoke it via the bound maven plugin. E.g., by reading provisioning scripts, stored in the pattern project and passing them to a *Puppet* instance, running within an IaaS Cloud. This decouples the implementation of a pattern from its provisioning actions and, thus, allows to switch the provisioning provider, by simply redefining the bindings within the pattern's *pom.xml*. This approach can even be more decoupled, if common *cloud integration services* are integrated by this methods, like *Apache DeltaCloud* [Del]. This allows to shield the current cloud provider behind the integration service. Therefore, the cloud provider can be exchanged without affecting the pattern implementation, respectively the binding. Figure 5.1 exemplary illustrates such a binding, leveraging the standard *maven pom syntax*.

```
<profile>
    <id>deploy</id> <!-- Handles the deploy goal. -->
    <build>
        <plugins>
          <plugin> <!-- Definition of the bound plugin -->
            <groupId>...</groupId>
            <artifactId>...</artifactId>
            <version>....</version>
            <configuration> <!-- General plugin configuration values -->
              ...
            </configuration>
            <executions> <!-- Definition of the execution order -->
              <execution>
                <id>...</id>
                <phase>...</phase>
                <goals>
                  <goal>...</goal> <!-- Definition of the invoked plugin goals -->
                </goals>
              </execution>
            </executions>
          </plugin>
        </plugins>
    </build>
</profile>
```

Listing 5.1: Outline - Binding of a Maven Plugin Goal to the Workspace Plugin *Deploy* Goal.

5.2.4 Maven Workspace Plugin Goal Invocation Syntax

This section provides detailed information about the invocation syntax of the implemented *Maven workspace plugin goals.* The following paragraphs briefly present the corresponding goal invocation syntax, along with the required parameters and the possible projects, the goals can be invoked on.

Note: The implemented workspace plugin is structured according to the technical architecture, presented in Subsection 5.2.1, thus, each goal is offered via the *workspace-plugin-facade*.

The following goals have been implemented [6]:

- **generate-archetype**

  Invocation Location:
  An arbitrary pattern project in the developer workspace.
  Command:
  mvn workspace-plugin:generate-archetype

- **persist-archetype**

  Invocation Location:
  A pattern project of a previously created archetype.
  Command:
  mvn workspace-plugin:persist-archetype

- **initialize-archetype**

  Invocation Location:
  An arbitrary pattern project in the developer workspace.
  Parameters:
  {x, y, z} = artifact information about the archetype to be instantiated.
  {a, b, c} = artifact information about the new, cloned artifact.
  Command:
  mvn edu.diplom.thesis.maven.plugins:workspace-plugin-facade:1.0:
  initialize-archetype -DarchetypeGroupId=x -DarchetypeArtifactId=y
  -DarchetypeVersion=z -DgroupId=a -DartifactId=b -Dversion=c
  Note:
  The *initialize-archetype* goal is the only goal that requires explicit parameters and fully specified plugin information, since no meta-information is stored, in any *pom.xml*, before this step.

[6] Each goal implements a single use case, defined in Use Case Appendix A

- **initialize-implementation-archetype**

  Invocation Location:
  Base Project → Goal is executed in all reachable projects.
  Single Abstract Project → Only executed in the current *abstract project.*
  Parameters:
  Automatically read from the provided *vd.xml* file.
  Command:
  mvn workspace-plugin:initialize-implementation-archetype

- **configure**

  Invocation Location:
  Base Project → Goal is executed in all reachable projects.
  Single Project → Only executed in the current project.
  Parameters:
  Automatically read from the provided *vd.xml* files.
  Command:
  mvn workspace-plugin:configure

- **provision**, **deploy**, **undeploy**, **decommission**

  Invocation Location:
  Base Project → Goal is executed in all reachable projects.
  Single Project → Only executed in the current project.
  Parameters:
  Automatically read from the provided *vd.xml* and *pom.xml* files.
  Command:
  mvn workspace-plugin:{*provision*, *deploy*, *undeploy*, *decommission*}
  Note:
  These goals trigger a build process and activate a corresponding build profile. Actions, that are bound within the patterns' *pom.xml* to these goals, are then invoked.

Note: Not each of the above presented goal invocations, requires to specify the full meta-information about the implemented workspace plugin, as this information is typically stored within the root projects *pom.xml* file via the Maven-specific *goalPrefix* element.

5.2.5 XML Representation of the Functional Variability Meta-Model

The implemented workspace plugin requires a serialized variability model, to interpret bound variability points and their interrelations. This variability model is serialized using XML [XML] and is stored in each pattern project, within the so-called *vd.xml*. Hence, a xml schema has been designed, serving as model of these variability models that implements the functional variability meta-model, presented in *Concept and Specification Section* 4.6. The following paragraphs provide graphically extracts of the implemented xml schema [7].

Figure 5.4 illustrates the xml root element of each *vd.xml*; the *variability model* element. A *variability model* consists of a name, a description and a *variability points* element, which contains several *variability points*.

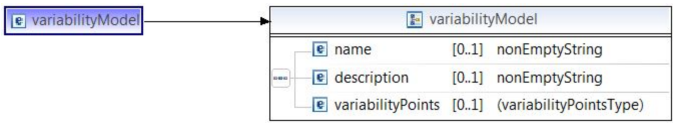

variabilityModel
variabilityModel
name [0..1] nonEmptyString
description [0..1] nonEmptyString
variabilityPoints [0..1] (variabilityPointsType)

Figure 5.4: XML Schema Illustration - The Variability Model Element.

Figure 5.5 illustrates the xml implementation of a *variability point*. Each variability point is addressable via an id attribute. The so established reference forms a soft reference, as the xml schema does not validate the existence of the addressed variability point, within the current variability descriptor. Thus, this soft references are resolved in each goal invocation, of the implemented *workspace plugin*. The referenced variability points then have to be present, within variability descriptors that are reachable in the workspace. The status attribute indicates, whether the variability point had been *untouched*, *enabled* or *disabled*. The requires provisioning flag, indicates whether the variability points requires provisioning actions during the configuration process, which may affect other variability points that reference received configuration details. These actions are then triggered via the provisioning goal, of the implemented workspace plugin, which then invokes the corresponding maven profiles, defined within the pattern's pom.xml [8]. Such a provisioning flag, e.g, is required in case of *Amazon's Simple Queueing Service* (AWS SQS), which returns the address of a recently created queue, after the queue has been created. Patterns that require this address then need to wait until the queue has been provisioned. Hence, such provisioning actions are invoked during the configuration flow, when the *configure* goal encounters a variability point that is marked with such a provisioning flag. Remaining relations and attributes are modeled according to the variability meta-model, presented in Section 4.6.

[7]The complete xml schema can be found in Appendix C.1.
[8]Subsection 5.2.3 discusses the invocation of concrete, pattern-defined actions in detail.

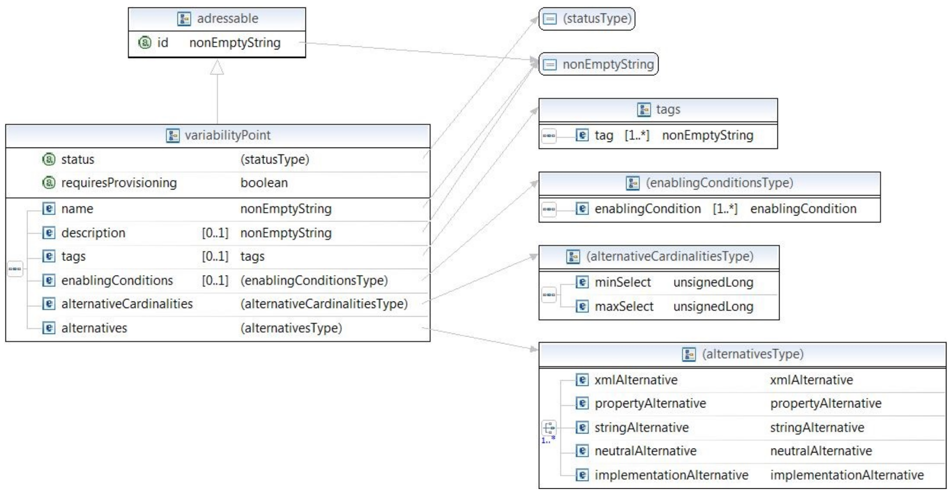

adressable
id nonEmptyString
variabilityPoint
status (statusType)
requiresProvisioning boolean
name nonEmptyString
description [0..1] nonEmptyString
tags [0..1] tags
enablingConditions [0..1] (enablingConditionsType)
alternativeCardinalities (alternativeCardinalitiesType)
alternatives (alternativesType)
(statusType)
nonEmptyString
tags
tag [1..*] nonEmptyString
(enablingConditionsType)
enablingCondition [1..*] enablingCondition
(alternativeCardinalitiesType)
minSelect unsignedLong
maxSelect unsignedLong
(alternativesType)
xmlAlternative xmlAlternative
propertyAlternative propertyAlternative
stringAlternative stringAlternative
neutralAlternative neutralAlternative
implementationAlternative implementationAlternative

Figure 5.5: XML Schema Illustration - The Variability Point Element.

Figure 5.6 illustrates the implemented alternative types and their related attributes. *Xml alternatives*, *property alternatives* and *string alternatives* store values, which are later on pushed into referenced variability locations, during the configuration process. *Neutral alternatives* and *implementation alternatives* do not reference variability locations. Thus, *implementation alternatives* are simply selected, respectively deselected.

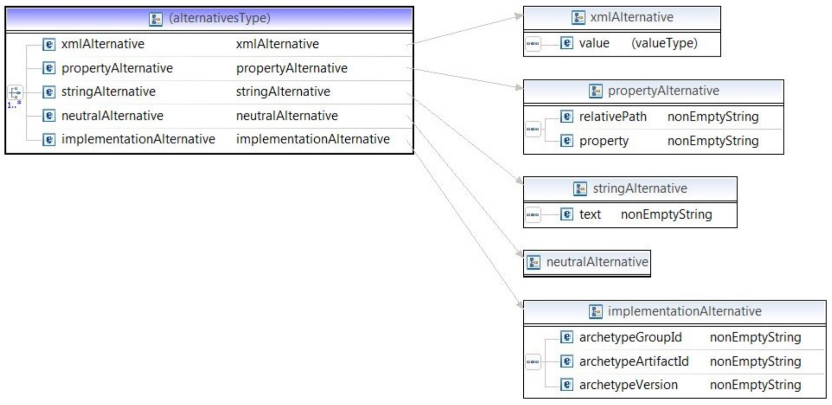

(alternativesType)
xmlAlternative xmlAlternative
propertyAlternative propertyAlternative
stringAlternative stringAlternative
neutralAlternative neutralAlternative
implementationAlternative implementationAlternative
xmlAlternative
value (valueType)
propertyAlternative
relativePath nonEmptyString
property nonEmptyString
stringAlternative
text nonEmptyString
neutralAlternative
implementationAlternative
archetypeGroupId nonEmptyString
archetypeArtifactId nonEmptyString
archetypeVersion nonEmptyString

Figure 5.6: XML Schema Illustration - Implemented Alternatives.

Figure 5.7 illustrates the implemented *alternative extension hierarchy*, as well as the locators that are used by the alternatives. Each alternative extends the *alternative base* element, which provides a status attribute that indicates whether the alternative has been *untouched*, *selected* or *deselected*. Furthermore, it provides a mandatory attribute that indicates whether the alternative has to be selected during the configuration process. Each *alternative* may contain a list of dependencies that reference a variability point or an other alternative. *Locator* *alternatives* extend the base alternative element, by referencing variability locations, via *locators*. The implemented Locator Alternative Types are: The *string alternative*, which defines a string value; the *xml alternative*, which stores an arbitrary xml element; and the *properties alternatives* that reads the alternative value from a referenced properties file.

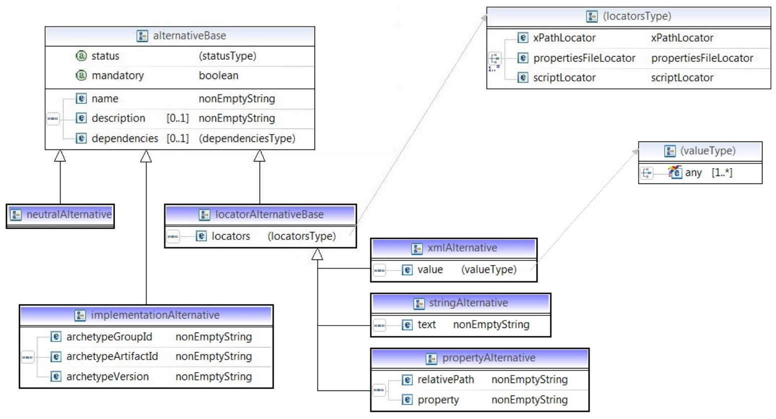

alternativeBase
status (statusType)
mandatory boolean
name nonEmptyString
description [0..1] nonEmptyString
dependencies [0..1] (dependenciesType)
neutralAlternative
locatorAlternativeBase
locators (locatorsType)
(locatorsType)
xPathLocator xPathLocator
propertiesFileLocator propertiesFileLocator
scriptLocator scriptLocator
(valueType)
any [1..*]
xmlAlternative
value (valueType)
stringAlternative
text nonEmptyString
propertyAlternative
relativePath nonEmptyString
property nonEmptyString
implementationAlternative
archetypeGroupId nonEmptyString
archetypeArtifactId nonEmptyString
archetypeVersion nonEmptyString

Figure 5.7: XML Schema Illustration - Alternative Extension Hierarchy and Locators.

Figure 5.8 provides a detailed illustration of the implemented *locator types*: The *xPath locator*, which references a set of xml elements, within a xml document; The *properties file locator*, which references a property, within a properties file; and the *script locator*, which references a variability location that is marked with a pre-defined symbol, located within an arbitrary script file. These locators are used, by the implemented alternatives and point to variability locations. The values, defined by the corresponding alternatives, are pushed, during the configuration process, into the referenced locations. The combination, of the implemented alternatives and locators, allows to model arbitrary configuration situations, where configurations can be pre-defined, by the workspace user, or can be automatically read, during the configuration process, which then pushes the alternative values into the addressed variability locations.

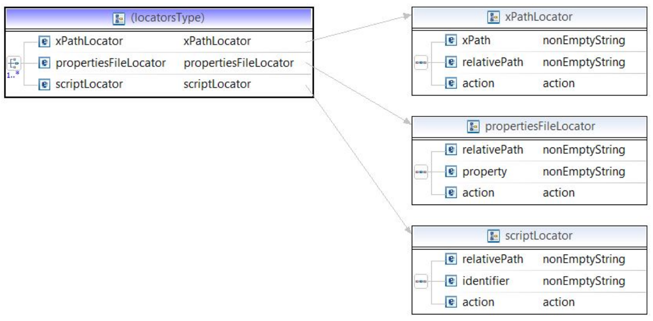

(locatorsType)
xPathLocator xPathLocator
propertiesFileLocator propertiesFileLocator
scriptLocator scriptLocator
xPathLocator
xPath nonEmptyString
relativePath nonEmptyString
action action
propertiesFileLocator
relativePath nonEmptyString
property nonEmptyString
action action
scriptLocator
relativePath nonEmptyString
identifier nonEmptyString
action action

Figure 5.8: XML Schema Illustration - Variability Locator Types.

5.3 Used Technologies and Tools

This section briefly provides an overview of technologies and tools, which have been used during the development of the concept, presented in Chapter 4. These technologies and tools have been used, to implement the corresponding *maven workspace plugin* and reference *cloud computing patterns*, i.e., the *Three-Tier Cloud Application Pattern* and one of its sub-patterns, the *Message-Oriented Middleware* pattern.

- *Java* - *Java* [Java] and the related *JavaEE* [Javb] Standard have been used to implement patterns and their pattern logic. Furthermore, the implemented maven workspace plugin was implemented using Java.
- *Apache Maven* - *Maven* [Mava] has been used to: create, store and load pattern archetypes into the *pattern template catalogue* - an Artifactory repository [Art]; execute *workspace plugin goals* [9], on Maven-based pattern projects; and to execute pattern-defined actions.
- *PuppetLabs' Puppet* - *Puppet* [Pup] has been used, to install required middleware on provisioned IaaS computation nodes.
- *Eclipse* - *Eclipse* [Ecl], along with the Eclipse Maven Plugin *m2Eclipse* [M2E] and the Cloudsmith Inc. Eclipse Puppet Plugin *Gepetto* [Gep], has been used to implement Maven-based pattern projects and puppet modules.
- *Amazon EC2* - *Amazon's EC2 Web Service* (AWS EC2) [EC2] has been used as IaaS environment provider, to create new computation nodes. Required middleware stacks are placed on top, to provide the required run time environments, respectively middleware services.
- *Amazon Elastic Beanstalk* - *Amazon's Elastic Beanstalk Service* [Ela] has been used as PaaS environment provider, to create a managed web container, where pattern-based web projects can be deployed to.
- *Amazon Simple Queuing Service* - *Amazon's Simple Queuing Service* (AWS SQS) has been used as *Commerical Off-The-Shelf Provider*, to implement the *Message-Oriented-Middleware* pattern. *Apache ActiveMQ* [Act] has been used as implementation alternative to SQS, which can be placed on an IaaS node.
- *Apache Deltacloud* - *Deltacloud* [Del] has been used, by the implemented *Deltacloud-Maven-Plugin*, to decouple dependencies between IaaS Provider APIs and patterns.

The following, pre-build Maven plugins have been used: Apache Tomcat's *Maven-Tomcat-Plugin* [Tomb], to deploy web project binaries to on-premise Tomcat instances; Ingenieux Labs' *Maven Elastic Beanstalk Plugin*, to deploy web project binaries into managed Tomcat environments; and Apache Maven's *Maven-Wagon-Plugin*[Mavb], to deploy puppet modules via *SSH* to provisioned *IaaS* computation nodes.

[9]These goals are defined within Chapter 4 and in Use Case Appendix A.

5.4 Implemented Reference Pattern(s)

This section provides an overview of implemented Java-based *cloud computing reference patterns*, which serve as base artifacts for the implemented *software product line*. The implemented patterns and code components are introduced in detail, in the following subsections, to provide a proof of concept, for the concept, presented in Section 4. Furthermore, the implemented reference patterns outline the *instantiation*, *customization* and *provisioning* capabilities, offered by the maven *workspace plugin*. The plugin allows to construct aggregated solutions, by instantiating previously stored pattern archetypes, customizing them and invoke various actions on them. Arbitrary parts, of so implemented, pattern-based applications, can be stored within the *pattern template catalogue*, by creating and persisting the so created archetypes. Other developers then can instantiate these archetypes, customize them and integrate them into their applications. These possible scenarios are outlined in the following, by providing implementation details, about the implemented patterns.

First, Subsection 5.4.1, introduces the implemented *Three-Tier Web Shop*, which implements the *Three-Tier Cloud Application Pattern*, identified by Fehling et al. [FLR+12]. Implemented components, maven projects and the underlying variability descriptors, modeling the variability points of the web shop, are presented in detail, as well as the corresponding deployment scenarios. The *Three-Tier Web Shop* outlines the possibilities to create an aggregated, pattern-based application. Subsection 5.4.2 then presents the implementation of the *Message-oriented Middleware Pattern* [FLR+12], to present the concept of commonly known interfaces and abstract patterns, introduced by the *functional project structure model*, presented in Section 4.5, which allows to offer *abstract application skeletons* and, then, instantiate concrete, selected pattern implementations.

5.4.1 Three-Tier Cloud Application

This subsection presents the *Three-Tier Web Shop*, which implements the *Three-Tier Cloud Application* pattern, identified by Fehling et.al. [FLR+12]. Its components, variability points and the possible deployment scenarios are presented in the following.

Figure 5.9 illustrates the implemented components that form the *Three-Tier Web Shop*. The web shop consists of three decoupled layers: the *presentation tier*, the *business logic tier* and the *data tier*.

Presentation Tier
The *presentation tier* consists of a *load balancer*, which receives incoming requests and dispatches them to the so-called *presentation application component*; related logic components are implemented by using the *stateless component* pattern. *Shop users* interact with the presentation application component by the *user interface component*. An *elastic load balancer* monitors the number of requests and provisions new instances of the presentation application component if required.

Business Logic Tier

The *business logic tier* consists of a *queue* (Message-Oriented Middleware), from which the so-called *business logic application component* receives processing requests and propagates processing results. The *business logic application component* also consists of *stateless components*, which implement the so-called *processing component*. This component implements the business use cases of the web shop. The *business logic tier* communicates with the *data tier* via a separate *queue*. Instances of the *business logic application component* are scaled by the *elastic queue* component, which measures the current number of messages in the request queue and creates, respectively deletes instances.

Data Tier

The *data tier* offers data access via a so-called *data access component*. This component receives incoming requests from a *queue* and queries the addressed information in the corresponding *storage offering*. Instances of the *data access component* are dynamically created and deleted, dependent on the current number of messages, stored within the corresponding queue.

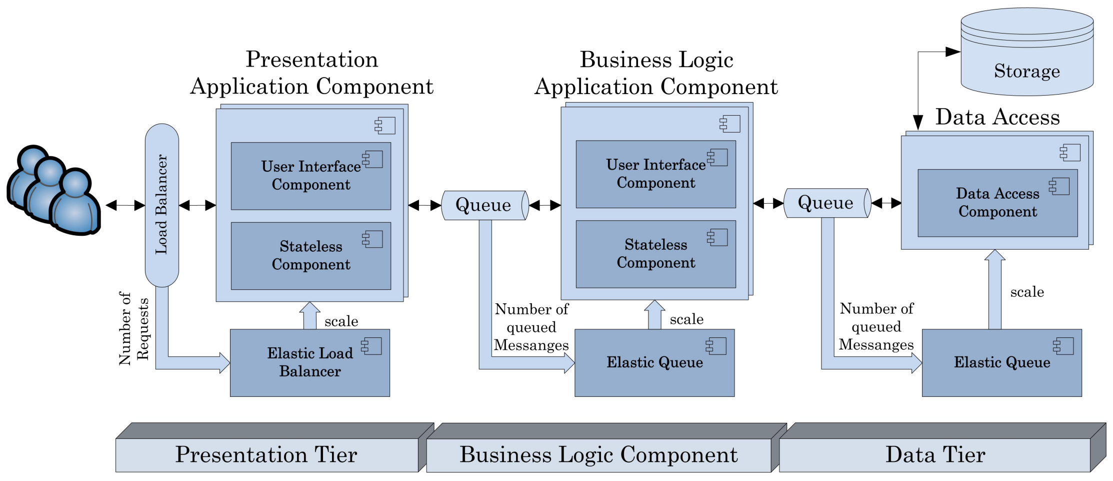
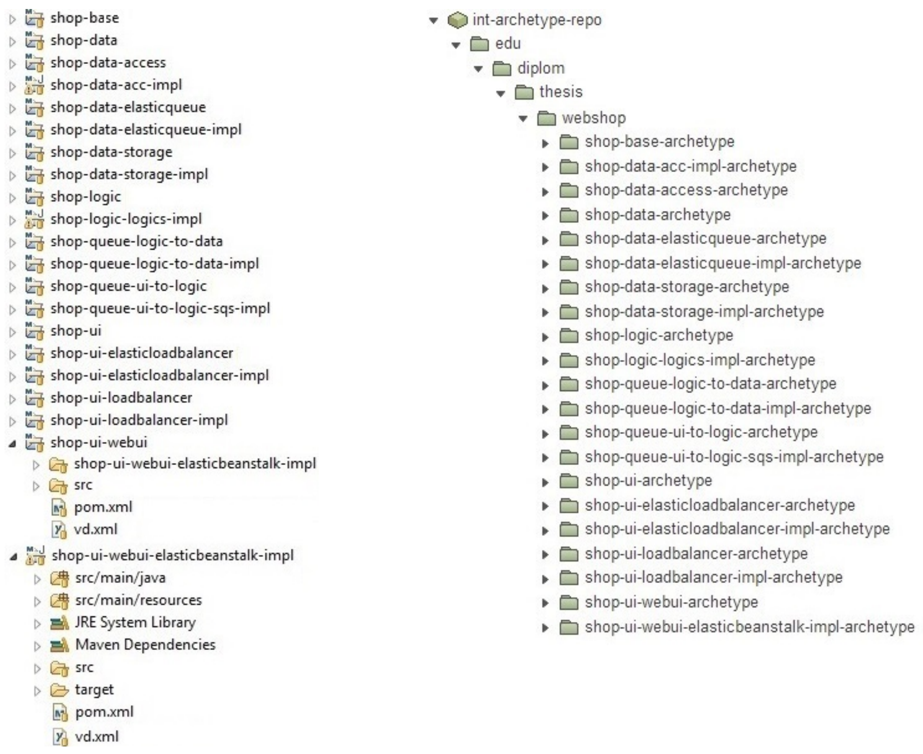


Figure 5.9: Overview - The Three-Tier Cloud Application Pattern. [FLR+12]

The hierarchical structure of the *Three-Tier Cloud Application Pattern*, illustrated in Figure 5.9, can directly be mapped to maven projects, using the *functional project structure model*, presented in Section 4.5, where each component is implemented in a single maven project. The *root project* includes all *modules* that are directly located under the overall pattern. Such a module may be: an *abstract project*, which allows to decouple required components from concrete implementations; or a concrete *implementation project*, which allows to leverage full benefits of the service provider's API, but does not allow to switch the implementation without code adaption.

Figure 5.10 illustrates the implemented maven projects (left), located at an *eclipse-based workspace*, as well as the corresponding, generated archetypes (right), stored within the *pattern template catalogue* - implemented by an artifactory repository.



```
▷ shop-base
▷ shop-data
▷ shop-data-access
▷ shop-data-acc-impl
▷ shop-data-elasticqueue
▷ shop-data-elasticqueue-impl
▷ shop-data-storage
▷ shop-data-storage-impl
▷ shop-logic
▷ shop-logic-logics-impl
▷ shop-queue-logic-to-data
▷ shop-queue-logic-to-data-impl
▷ shop-queue-ui-to-logic
▷ shop-queue-ui-to-logic-sqs-impl
▷ shop-ui
▷ shop-ui-elasticloadbalancer
▷ shop-ui-elasticloadbalancer-impl
▷ shop-ui-loadbalancer
▷ shop-ui-loadbalancer-impl
◢ shop-ui-webui
    ▷ shop-ui-webui-elasticbeanstalk-impl
    ▷ src
      pom.xml
      vd.xml
◢ shop-ui-webui-elasticbeanstalk-impl
    ▷ src/main/java
    ▷ src/main/resources
    ▷ JRE System Library
    ▷ Maven Dependencies
    ▷ src
    ▷ target
      pom.xml
      vd.xml
```

```
▾ int-archetype-repo
    ▾ edu
        ▾ diplom
            ▾ thesis
                ▾ webshop
                    ▸ shop-base-archetype
                    ▸ shop-data-acc-impl-archetype
                    ▸ shop-data-access-archetype
                    ▸ shop-data-archetype
                    ▸ shop-data-elasticqueue-archetype
                    ▸ shop-data-elasticqueue-impl-archetype
                    ▸ shop-data-storage-archetype
                    ▸ shop-data-storage-impl-archetype
                    ▸ shop-logic-archetype
                    ▸ shop-logic-logics-impl-archetype
                    ▸ shop-queue-logic-to-data-archetype
                    ▸ shop-queue-logic-to-data-impl-archetype
                    ▸ shop-queue-ui-to-logic-archetype
                    ▸ shop-queue-ui-to-logic-sqs-impl-archetype
                    ▸ shop-ui-archetype
                    ▸ shop-ui-elasticloadbalancer-archetype
                    ▸ shop-ui-elasticloadbalancer-impl-archetype
                    ▸ shop-ui-loadbalancer-archetype
                    ▸ shop-ui-loadbalancer-impl-archetype
                    ▸ shop-ui-webui-archetype
                    ▸ shop-ui-webui-elasticbeanstalk-impl-archetype
```

Figure 5.10: Three-Tier Web Shop - Maven Projects and Generated Archetypes.

Note: Eclipse illustrates imported maven projects in a flat view, meaning that all projects are presented at the same level, regardless of the defined dependency hierarchy.

Each shop project contains a maven *pom.xml*, which defines the hierarchy of the maven projects, by defining dependencies to other projects. Moreover, each project contains a *vd.xml*, which stores the *variability descriptor*, respectively the *variability points*, of the pattern project. The stored *variability points* and their relations are hierarchically structured, according to the structure of the related maven projects, meaning that each maven dependency can be mapped to an *alternative dependency*, stored within the variability descriptor of the related project.

The shop also consists of an *application skeleton*, which defines the logical structure of the shop. This is done by dependency definitions, pointing to *abstract projects*. Thus, the *root project* of the shop (*shop-base*) defines, in its *pom.xml*, that it requires an abstract *presentation tier* (*shop-ui*) and several other projects, respectively patterns. The *shop-ui* project defines that an abstract *web-ui* is required. The *web-ui* project (shop-ui-webui) provides several *implementation alternatives* that fit the common requirements of a web-ui. *Abstract projects* contain *neutral alternatives* that delegate to concrete variability points. This allows to plug in an arbitrary *vd.xml* file, nested within the corresponding *implementation project*, into the overall *variability graph*, as long as it contains a root variability point with a compatible id.

All projects within the workspace, except the selected *implementation projects*, form the *application skeleton*, as they only define the structure and dependencies, which are required to build a concrete application. *Variability descriptors*, of *abstract projects*, contain so-called *implementation alternatives*, presented in Section 4.6, which can be selected and instantiated by developers, during the *pattern development process*, presented in Section 4.4. Yet, additional projects that require code, offered by such an *implementation alternative*, need to add dependencies to the instantiated implementation pattern [10].

Since the *maven workspace plugin* allows to invoke goals on arbitrary projects, archetypes can be created at any scale of the project structure, meaning that an archetype can be created for the entire pattern or for any sub-patterns. When the entire pattern archetype is created, single archetypes, for all sub-patterns, are generated, which then can be stored in the *pattern template catalogue.* This allows to the invocation of the *create-archetype* goal once, and generate standalone, reusable patterns for each reachable sub-pattern. Figure 5.10 (right) illustrates this behavior, as each pattern project has its own archetype. Aggregated archetypes, like the *shop-base-archetype*, simply point at their module archetypes. During instantiation, these archetypes are fetched by Maven.

Customization Points
The presented *Three-Tier Web Shop* offers various variability points to *workspace users*, which can be bound to customize instantiated patterns. The web shops' *application skeleton* projects reference variability points, stored in *abstract projects*, from where the *workspace user* selects implementations. Within such an *abstract project*, the selected *implementation alternative* references a root variability point, defined by the selected *implementation project.* The referenced id-attribute, defined within the root variability point of all compatible implementation projects, are identical. This allows to switch implementation projects, without adapting variability descriptors, since concrete variability points are stored within the projects, where the corresponding variability locations occur. The so defined soft-references are resolved, during workspace plugin goal executions, to concrete variability points, stored within the workspace [11].

To outline the implemented concept and to provide an overview of customization possibilities, the following paragraphs cover customization points of the web shops' *application presentation component* (shop-ui-webui). The application presentation component is implemented by a so-called *web-project*, meaning that each implementation project creates a *.war* file that can be deployed into a *JavaEE* conform web container. Such a web container can be running in an *on-premise* environment, an *IaaS* Cloud or in a *PaaS* Cloud. Thus, the *shop-ui-webui's vd.xml* provides several implementation alternatives that cover these deployment scenarios. A workspace user selects an implementation alternative, by switching the *status* attribute to *selected.* Then he invokes the workspace plugin goal *instantiate-implementation*, which loads the corresponding archetype, from the *pattern template catalogue*, into the workspace.

[10]Subsection 5.4.2 discusses this approach in detail, by presenting implementations of the *Message-Oriented Middleware* pattern, along with the related, abstract interface definition.
[11]This concept is based on the *Naming* concept, commonly used within distributed systems.

Listing 5.2 illustrates the particular implementation alternative definitions of the abstract *shop-ui-webui* project. Each implementation alternative addresses a stored implementation archetype, by its artifact meta-information. This meta-information is read, when the user invokes the *instantiate-implementation* goal of the workspace plugin.

```
<alternatives>
  <!-- AWS - Elastic Beanstalk -->
  <implementationAlternative status="selected" mandatory="false" id="#ElasticBeanstalk-Impl">
    <name>Shop-WebUI-ElasticBeanstalk-Impl-Alt</name>
    <dependencies>
      <dependency type="requires">
          <target>#Shop-WebUI-Implementation-VP</target>
      </dependency>
    </dependencies>
    <archetypeGroupId>edu.diplom.thesis.webshop</archetypeGroupId>
    <archetypeArtifactId>shop-ui-webui-elasticbeanstalk-impl-archetype</archetypeArtifactId>
    <archetypeVersion>0.0.1-SNAPSHOT</archetypeVersion>
  </implementationAlternative>
  <!-- OnPremise - Tomcat -->
  <implementationAlternative status="deselected" mandatory="false" id="#OnPremiseTomcat-Impl">
    <name>Shop-WebUI-OnPremiseTomcat-Impl-Alt</name>
    <dependencies>
      <dependency type="requires">
        <target>#Shop-WebUI-Implementation-VP</target>
      </dependency>
    </dependencies>
    <archetypeGroupId>edu.diplom.thesis.webshop</archetypeGroupId>
    <archetypeArtifactId>shop-ui-webui-onpremisetomcat-impl-archetype</archetypeArtifactId>
    <archetypeVersion>0.0.1-SNAPSHOT</archetypeVersion>
  </implementationAlternative>
  <!-- EC2 - Tomcat -->
  <implementationAlternative status="deselected" mandatory="false" id="#Ec2Tomcat-Impl">
    <name>Shop-WebUI-Ec2Tomcat-Impl-Alt</name>
    <dependencies>
      <dependency type="requires">
        <target>#Shop-WebUI-Implementation-VP</target>
      </dependency>
    </dependencies>
    <archetypeGroupId>edu.diplom.thesis.webshop</archetypeGroupId>
    <archetypeArtifactId>shop-ui-webui-ec2tomcat-impl-archetype</archetypeArtifactId>
    <archetypeVersion>0.0.1-SNAPSHOT</archetypeVersion>
  </implementationAlternative>
</alternatives>
```

Listing 5.2: Shop-WebUI Variability Descriptor - Implementation Alternatives.

Listing 5.3 illustrates the implemented variability descriptor for the *elastic beanstalk implementation* of the *shop-ui-webui*. The root variability point, *#Shop-WebUI-Implementation-VP*, points to an elastic beanstalk configuration variability point. This variability point defines four mandatory run time configurations, which reference variability locations, located in the pom.xml [12] of the elastic beanstalk implementation project.

[12]Referenced properties are illustrated in Figure 5.5.

```
    ...
    <variabilityPoint status="enabled"
        id="#Shop−WebUI−ElasticBeanstalk−Implementation:RunTimeConfig−VP">
        <name>WebshopUI − Elastic Beanstalk Implementation RunTime Configuration</name>
        <alternativeCardinalities> <!−− Cardinalities −−>
            <minSelect>4</minSelect>
            <maxSelect>4</maxSelect>
        </alternativeCardinalities>
        <alternatives> <!−− Runtime−Config −−>
            <stringAlternative status="selected" mandatory="true"
                    id="#Shop−WebUI−ElasticBeanstalk−Implementation:Application−Name">
                <name>AWS−Beanstalk−Application−Name</name>
                <locators>
                    <xPathLocator>
                        <xPath>/project/properties/aws.beanstalk.application.name</xPath>
                        <relativePath>pom.xml</relativePath>
                        <action>replace</action>
                    </xPathLocator>
                </locators>
                <text>thesis−three−tier−webshop</text>
            </stringAlternative>
            <stringAlternative status="selected" mandatory="true"
                id="#Shop−WebUI−ElasticBeanstalk−Implementation:S3−Bucket−Name">
                <name>S3−Bucket−Namer</name>
                <locators>
                    <xPathLocator>
                        <xPath>/project/properties/aws.beanstalk.s3.bucket.name</xPath>
                        <relativePath>pom.xml</relativePath>
                        <action>replace</action>
                    </xPathLocator>
                </locators>
                <text>thesis−webshop−bucket</text>
            </stringAlternative>
            <stringAlternative status="selected" mandatory="true"
                id="#Shop−WebUI−ElasticBeanstalk−Implementation:Beanstalk−Region">
                <name>Beanstalk−Region</name>
                <locators>
                    <xPathLocator>
                        <xPath>/project/properties/aws.beanstalk.region</xPath>
                        <relativePath>pom.xml</relativePath>
                        <action>replace</action>
                    </xPathLocator>
                </locators>
                <text>us−east−1</text>
            </stringAlternative>
            <stringAlternative status="selected" mandatory="true"
               id="#Shop−WebUI−ElasticBeanstalk−Implementation:S3−Key">
                <name>AWS−S3−Key</name>
                <locators>
                    <xPathLocator>
                        <xPath>/project/properties/aws.beanstalk.s3.key</xPath>
                        <relativePath>pom.xml</relativePath>
                        <action>replace</action>
                    </xPathLocator>
                </locators>
                <text>shop−ui−webui−impl</text>
            </stringAlternative>
        </alternatives>
    </variabilityPoint>
    ...
```

Listing 5.3: Elastic Beanstalk Implementation - Variability Point Definition.

Each *implementation project* defines concrete handlers, by defining profiles located at its *pom.xml* [13]. This is done for the following, abstract workspace plugin goals: *provision*, *deploy*, *undeploy* and *decommission*.

Listing 5.4 illustrates the implemented binding of *elastic beanstalk provisioning actions* to the abstract workspace-plugin goal *provision*. Goals of the Ingenieux Labs' *beanstalk-maven-plugin* [bea], to provision the required *tomcat environment*, are bound to the corresponding *provision* profile. The defined goals are executed sequentially and trigger the provisioning actions, within Amazon's PaaS Cloud, by invoking the corresponding web services. Required parameters, respectively *variability locations*, are stored on top of the *pom.xml*, within the so-called *properties* element, illustrated in Figure 5.5. During the configuration process, of the *workspace plugin*, alternative values, that reference these parameters, are read and pushed into the corresponding variability location. These parameters are then referenced within the plugin invocation definition (binding), via ${parameter.name} and read during the plugin invocation.

```
...
<!-- Binding Definitions -->
<profile>
        <id>provision</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>br.com.ingenieux</groupId>
                    <artifactId>beanstalk-maven-plugin</artifactId>
                    <version>0.2.8</version>
                    <configuration>
                        <server>aws.amazon.com</server>
                        <region>${aws.beanstalk.region}</region>
                        <applicationName>${aws.beanstalk.application.name}</applicationName>
                        <s3Bucket>${aws.beanstalk.s3.bucket.name}</s3Bucket>
                        <s3Key>${aws.beanstalk.s3.key}/${project.build.finalName}-${maven.build.timestamp}.war</s3Key>
                        <environmentName>shop-ui-webui</environmentName>
                    </configuration>
                    <executions>
                        <execution>
                            <id>deploy-to-elastic-beanstalk</id>
                            <phase>pre-integration-test</phase>
                            <goals>
                                <goal>upload-source-bundle</goal>
                                <goal>create-application-version</goal>
                                <goal>create-environment</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
...
```

Listing 5.4: Provisioning Profile Binding - Definition of Plugins and Goal Executions.

[13]Discussed in Subsection 5.2.3 - Pattern Defined Action Invocation.

```
...
<properties>
        <!-- Variability Locations -->
        <aws.access.key>...</aws.access.key>
        <aws.secret.key>...</aws.secret.key>
        <aws.beanstalk.application.name>thesis-three-tier-webshop</aws.beanstalk.application.name>
        <aws.beanstalk.s3.bucket.name>thesis-webshop-bucket</aws.beanstalk.s3.bucket.name>
        <aws.beanstalk.region>us-east-1</aws.beanstalk.region>
        <aws.beanstalk.s3.key>shop-ui-webui-impl</aws.beanstalk.s3.key>
</properties>
...
```

Listing 5.5: Elastic Beanstalk Implementation - Variability Locations in the POM File.

This binding mechanism allows to integrate arbitrary actions, encapsulated within Java-based maven plugins, to *provision/decommission* environments and to *deploy/undeploy* generated pattern binaries. A possible integration scenario, e.g, would be to use the implemented *Deltacloud-Maven-Plugin*, which interacts with a Deltacloud server, pointing at an *OpenStack* based IaaS Cloud [Ope]. The Deltacloud server then provisions a computation node, which runs a pre-installed *TOSCA container* [TOS]. After the computation node has been provisioned and the TOSCA container is up and running, the required middleware definition scripts are deployed into the TOSCA container, which then installs the required middleware components.

Note: A similar integration scenario, using *Amazon's EC2 Cloud* and *Puppet*, is presented in the subsequent Section 5.4.2, to provision an *ActiveMQ* server on top of an EC2 instance.

5.4.2 Message-Oriented Middleware

This subsection presents the implemented *Message-Oriented Middleware* pattern, identified by Fehling et.al. [FLR+12]. The identified pattern is implemented by an *abstract project*, which defines selectable, compatible implementation alternatives. Selected implementation alternatives are then instantiated and plugged in the existing application skeleton project structure. This concept allows to provide a map of implementations, defined within the abstract projects *vd.xml*, which are compatible to the abstract pattern. The used interface eliminates vendor dependencies and increases reusability. Yet, the common interface definition reduces offered functionalities, provided by single vendors, as it defines functionalities that has to be provided by all known implementations.

Figure 5.6 illustrates the implemented, compatible queue constellations, offered in the *vd.xml* of the abstract queue project. Each implementation alternative points to a root variability point, which is defined by each compatible implementation project; the *AbstractQueue-Impl-VP*. This allows to switch implementation projects, without adapting the *vd.xml* files of the corresponding projects.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<variabilityModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="#AbstractQueue-VM">
    <name>AbstractQueue - Variability Model</name>
    <variabilityPoints>
        <!-- AbstractQueue - Variability Point -->
        <variabilityPoint status="enabled" id="#AbstractQueue-VP">
            <name>AbstractQueue-VP</name>
            <alternativeCardinalities><!-- Cardinalities -->
                <minSelect>1</minSelect>
                <maxSelect>1</maxSelect>
            </alternativeCardinalities>
            <alternatives>
                <!-- AWS - SQS -->
                <implementationAlternative status="selected" id="#SQS-Impl-Alt">
                    <name>SQS-Impl-Alt</name>
                    <dependencies>
                        <dependency type="requires">
                            <target>#AbstractQueue-Impl-VP</target>
                        </dependency>
                    </dependencies>
                    <archetypeGroupId>edu.diplom.thesis.single.patterns</archetypeGroupId>
                    <archetypeArtifactId>sqs-impl-archetype</archetypeArtifactId>
                    <archetypeVersion>0.0.1-SNAPSHOT</archetypeVersion>
                </implementationAlternative>
                <!-- EC2 - Active-MQ -->
                <implementationAlternative status="deselected" id="#EC2-ActiveMQ-Alt">
                    <name>EC2-Active-MQ-Impl-Alt</name>
                    <dependencies>
                        <dependency type="requires">
                            <target>#AbstractQueue-Impl-VP</target>
                        </dependency>
                    </dependencies>
                    <archetypeGroupId>edu.diplom.thesis.single.patterns</archetypeGroupId>
                    <archetypeArtifactId>ec2-activemq-impl-archetype</archetypeArtifactId>
                    <archetypeVersion>0.0.1-SNAPSHOT</archetypeVersion>
                </implementationAlternative>
            </alternatives>
        </variabilityPoint>
    </variabilityPoints>
</variabilityModel>
```

Listing 5.6: Abstract Queue Variability Descriptor - Defined Implementation Alternatives.

Each compatible implementation offers a *factory object* that returns a *queue proxy object*, which implements the commonly known queue interface. Corresponding configuration parameters are defined within the *vd.xml* of the instantiated pattern. After the configuration process, parameters, required for the queue connection, are stored within a properties file, from where the factory object reads them and creates a *queue adapter object*. This adapter object is then used by the offered queue proxy object to send and receive messages [14]. An application that uses an instantiated, compatible queue implementation only need to add a maven dependency, to the newly instantiated pattern. Then, after compiling the application, the communication with the queue is handled by the plugged in queue proxy object. This allows to switch the implementation without adapting previously written code. This concept correlates to the *Provider Adapter Pattern*, identified by Fehling et.al. [FLR+12].

[14]The *Factory Pattern*, the *Proxy Pattern* and the *Adapter Pattern* are a well known patterns in the Java Design Community and had been identified by the so-called *Gang Of Four* [GHJV95].

Figure 5.7 illustrates the commonly known, abstract queue interface definition, which is supported by each queue implementation.

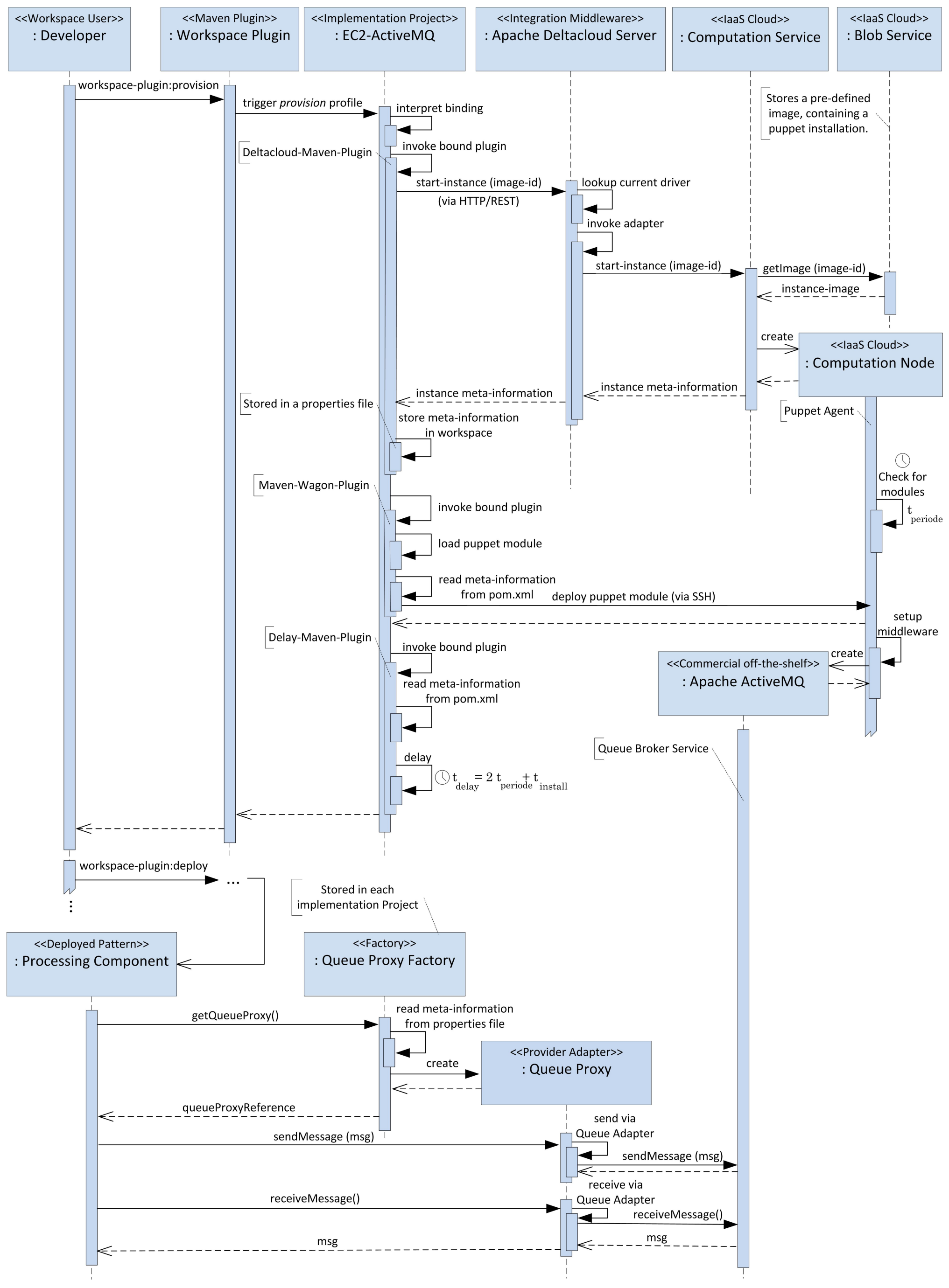
```java
public interface QueueProxy {
    /**
     * Receives a message from an Abstract Queue.
     *
     * @return An arbitrary message, stored within the addressed queue.
     * @throws QueueProxyReceiveException If no message can be received or the communication fails.
     */
    public String receiveMessage() throws QueueProxyReceiveException;

    /**
     * Sends a message to an Abstract Queue.
     *
     * @param message The message to be delivered.
     * @throws QueueProxySendException If the message can not be delivered.
     */
    public void sendMessage(String message) throws QueueProxySendException;
}
```

Listing 5.7: Abstract Queue - Common Interface Definition

Figure 5.11 illustrates the executed EC2-ActiveMQ provisioning steps, which are performed when a *developer* has invoked the workspace plugin goal *provision*. First the workspace plugin triggers a build process, which compiles the project sources, and activates the build profile *provision*. Then, within the EC2-ActiveMQ implementation project, the plugins, bound to the *provision* profile, are interpreted. These plugins are then executed sequentially. First, the *Deltacloud-Maven-Plugin* is executed. The corresponding *deltacloud server* [15] is addressed via its HTTP/REST interface. It receives the incoming request, which defines an *image* (by its image-id) that contains a *Puppet* installation and is stored within a *blob storage service*. Deltacloud then looks up the current *cloud provider* (driver), and dispatches the request to the corresponding provider adapter, which then starts the requested instance in the currently addressed IaaS Cloud. The corresponding meta-information is returned and stored in a properties file that is located in the implementation project folder. Then, the *Maven-Wagon-Plugin* [Mavb] is executed to deploy the ActiveMQ Puppet module via *SSH* into the *puppet agent*, which is running on the recently created computation node. The puppet agent periodically monitors the modules, stored within its module folder, and installs defined middleware, if the module is present. Hence, after the Maven-Wagon-Plugin has terminated, the implemented *Delay-Maven-Plugin* is executed, which delays the execution of possible subsequent plugins. After this, the developer invokes the workspace plugin goal *deploy*, which triggers the deployment of a *processing component*. The processing component then calls the factory, stored within the EC2-ActiveMQ implementation project, to create a new *queue proxy* object that implements the interface illustrated in Figure 5.7. The returned queue proxy object establishes the communication with the required queue, by instantiating a *queue adapter object* with the previously stored meta-information. Afterwards, the communication transparently

[15]Corresponding location information is stored within the *vd.xml* of the implementation pattern.

takes place via the provided queue proxy object, meaning that queue proxy consuming objects do not need to be aware of the concrete queue location.



<<Workspace User>> : Developer
<<Maven Plugin>> : Workspace Plugin
<<Implementation Project>> : EC2-ActiveMQ
<<Integration Middleware>> : Apache Deltacloud Server
<<IaaS Cloud>> : Computation Service
<<IaaS Cloud>> : Blob Service
workspace-plugin:provision
trigger provision profile
interpret binding
Stores a pre-defined image, containing a puppet installation.
invoke bound plugin
Deltacloud-Maven-Plugin
start-instance (image-id)
(via HTTP/REST)
lookup current driver
invoke adapter
start-instance (image-id)
getImage (image-id)
instance-image
create
<<IaaS Cloud>> : Computation Node
instance meta-information
instance meta-information
Stored in a properties file
store meta-information in workspace
Puppet Agent
Check for modules
t periode
Maven-Wagon-Plugin
invoke bound plugin
load puppet module
read meta-information from pom.xml
deploy puppet module (via SSH)
setup middleware
Delay-Maven-Plugin
invoke bound plugin
create
<<Commercial off-the-shelf>> : Apache ActiveMQ
read meta-information from pom.xml
Queue Broker Service
delay
t delay = 2 t periode + t install
workspace-plugin:deploy
...
Stored in each implementation Project
<<Deployed Pattern>> : Processing Component
<<Factory>> : Queue Proxy Factory
getQueueProxy()
read meta-information from properties file
create
<<Provider Adapter>> : Queue Proxy
queueProxyReference
send via Queue Adapter
sendMessage (msg)
sendMessage (msg)
receive via Queue Adapter
receiveMessage()
receiveMessage()
msg
msg

Figure 5.11: Sequence Diagram - EC2-ActiveMQ Provisioning and Queue Communication.

Summary and Outlook

6.1 Summary

This thesis outlined and implemented major components of a possible *software product line for cloud computing patterns*, to create customizable pattern-based artifacts, for recently identified cloud computing patterns, as well as an approach to provision these artifacts into different cloud environments. These challenges were solved by combining approaches of software product line engineering with open source tools, i.e., *Apache Maven* and *PuppetLabs' Puppet*. Therefore, *Chapter 1* provided a motivation for this thesis and defined its focus. *Chapter 2* discussed relevant, scientific fundamentals about *cloud computing*, *software product line engineering* and *patterns*. *Chapter 3* presented related work, including: *cloud computing patterns*, which form the foundation of implemented reference patterns; the *Cafe platform* and its concept of variability descriptors; *Apache Maven*, a build management tool, offering the possibility to manage Java-based projects; and *PuppetLabs' Puppet*, an infrastructure management tool, that allows to setup middleware stacks. These approaches have been combined within the concept presented in *Chapter 4*. The outlined approach provides major parts of the introduced *software product line*, which forms the superordinate context of this thesis. Moreover, the functional and non-functional requirements for implemented components have been presented. Afterwards, the abstract, functional architecture of the implemented approach has been introduced, in combination with its *project structure model* and its *variability meta-model*. The presented project structure model allows to decouple dependencies between applications and used pattern implementations, by intermediate *abstract projects*. These projects define a common interface, which is implemented by compatible pattern implementations and thus, allows to switch them without code adaption. Customization points of aggregated, pattern-based applications have been addressed with an adapted version of the presented *Cafe* variability meta-model, which is distributedly stored within the variability descriptors of the corresponding pattern implementations. Afterwards, functional components of the presented concept have been specified, in combination with their corresponding use cases. *Chapter 5* refined these components, by providing detailed design decisions, about the implemented maven workspace plugin and its components. Furthermore, technical details about the realized *pattern goal invocation flow* have been presented, which is based on Maven's internal goal propagation mechanism, in combination with goal dependent handler definitions. This allows *pattern developers* to integrate arbitrary provisioning actions. Finally, implemented reference patterns and used technologies have been presented to provide proof of concept.

6.2 Outlook

During this thesis, major parts of the superordinate software product line for cloud computing patterns, presented in Section 4.1, have been implemented. Yet, not all of the corresponding components have been fully implemented, i.e., the *pattern catalogue console* and the *pattern factory*. Further work, may focus on implementing the *pattern catalogue console*, which graphically illustrates cloud computing patterns, stored within the cloud computing pattern template catalogue, and allows to model aggregated pattern-based applications, by graphical means. The *pattern catalogue console* stores generated meta-information, about the aggregated application, within a file container. This file container is interpreted by the pattern factory, which triggers required goals, provided by the implemented *maven workspace plugin*, to setup the workspace of the defined application. The implementation of these components could even more reduce human interaction, when creating new pattern-based applications.

Furthermore, the implemented reference patterns provide support for integrating *Apache Deltacloud* and *PuppetLabs' Puppet*. These integration functionalities may be extended as part of further works, e.g., by *TOSCA* or *Chef* provisioning functionalities. Also, further maven plugins may be developed in the future, using JClouds [JCl] to overcome different IaaS Cloud APIs. This would enable the possibility to directly communicate with the corresponding IaaS Cloud, without a separate server installation. Moreover, further pattern implementations may focus on different scaling strategies for the created applications, which can be handled by the application itself or may be shifted to tools like *Puppet*, *Chef* or *Scalr* [Sca].

Moreover, the implemented reference pattern artifacts lack semantic descriptions, which can be used to offer proposals for aggregation applications, fitting the needs of developers. These descriptions are currently - at the time of this thesis - stored within a semantic wiki, i.e., DataWiki [Dat]. Combining the benefits of this semantic wiki with the benefits of the implemented approach of this thesis could be part of further research.

Use Case Appendix

	Description
Name	*Create Archetype from Existing Pattern*
Goal	The workspace user wants to create an archetype from an existing maven-based project.
Actor	Workspace User
Pre-Condition(s)	A maven-based project exists. All required files and configurations are present. The workspace plugin is installed and present.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The user executes the adapted maven-archetype plugin. 2. The plugin validates the pattern project(s). 2.1. The transitive shell of all projects is computed. 2.2. Each project within the shell is traversed. 2.2.1. The plugin checks if a variability descriptor is present. 2.2.2. The plugin checks if all required points are bound. 2.2.3. The plugin checks if all optional points are unbound. 2.3. The plugin globally checks for cyclic dependencies. 3. The plugin locally creates an archetype of the project. 4. A message about the successful processing status is logged. 5. The plugin terminates.
Special Cases	1.a. The custom plugin is missing. a.) Maven raises an exception, that the plugin is unknown. b.) Maven logs the reason and terminates. 2. Validation fails. a.) The plugin logs the reason and terminates. 1-4. An unchecked exception occurs. a.) The plugin logs the exception and terminates.
Note	The creation of the archetype-project is pre-implemented by Maven, yet the validation of the project structure had been added.

Table A.1: Use Case - Create Archetype from Existing Pattern.

	Description
Name	*Store Existing Archetype into the Pattern Catalogue*
Goal	The user wants to store a previously created archetype into the cloud computing pattern catalogue.
Actor	Workspace User
Pre-Condition(s)	A previously, locally created archetype-project exists. The required catalogue address is passed during the invocation of the plugin goal. The workspace plugin is installed and present.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The user starts the plugin goal to execute the persistance steps. 1.1. The plugin generates the archetype binaries. 1.2. The plugin stores the binaries into the pattern catalogue. 1.3. The stored binaries become visible for external usage. 2. A message about the successful processing status is logged. 3. The plugin terminates.
Special Cases	1.a. The plugin is missing. a.) Maven raises an exception, that the plugin is unknown. 1.b. The build life-cycle fails. a.) The archetype binaries are not created. b.) The archetype binaries are not stored. 1.c. The catalogue artifact repository is not reachable. a.) The plugin builds the binaries. b.) The plugin logs the reason and terminates. 1.d. An unchecked exception occurs. a.) The plugin logs the exception and terminates.
Note	This use case is partially pre-implemented by the standard functionality of Maven, it is used as part of the overall method to create reference pattern implementations.

Table A.2: Use Case - Store Existing Archetype into the Pattern Catalogue.

	Description
Name	*Instantiate an Archetype from the Pattern Catalogue*
Goal	The user wants to instantiate a previously stored pattern into his local workspace.
Actor	Workspace User
Pre-Condition(s)	The catalogue address is present in any *settings.xml* file. The pattern catalogue is up and running. The archetype is present in the catalogue. The maven plugin is present.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The workspace user queries the possible archetypes. 2. Maven lists all archetypes, defined within its known repositories. 3. The workspace user selects an archetype. 4. Maven creates the project structure within the local workspace. 5. A message about the successful processing status is logged. 6. The plugin terminates.
Special Cases	1.a. The standard plugin is missing. a.) Maven raises an exception, that the plugin is unknown. 2.a. The catalogue artifact repository is not reachable. a.) The archetype is not listed within the archetype list. 4.a. The creation of the requested projects fails. a.) The project structure is not created. b.) Maven logs an exception about the reason. c.) The standard plugin terminates. 1-4. An unchecked exception occurs. a.) The plugin logs the exception and terminates.
Note	This use case is pre-implemented by the standard functionality of Maven, it is used as part of the overall method to create reference pattern implementations.

Table A.3: Use Case - Instantiate an Archetype from the Pattern Catalogue.

	Description
Name	*Instantiate an Implementation Archetype from the Cloud Computing Pattern Catalogue*
Goal	The user wants to instantiate a selected implementation project into his local workspace.
Actor	Workspace User
Pre-Condition(s)	The catalogue address is present in any *settings.xml* file. The pattern catalogue is up and running. The archetype is present in the catalogue. The user has selected an implementation alternative within the project variability descriptor. The maven plugin is present.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The workspace user starts the plugin to initialize the implementation archetype, selected within the variability descriptor. 2. The plugin reads the stored archetype information from the variability descriptor. 3. The plugin initializes the implementation project. 4. A message about the successful processing status is logged. 5. The plugin terminates.
Special Cases	1.a. The plugin is missing. a.) Maven raises an exception, that the plugin is unknown. 3.a. The creation of the requested projects fails. a.) The project structure is not created. b.) Maven logs an exception about the reason. c.) The plugin terminates. 1-3. An unchecked exception occurs. a.) The plugin logs the exception and terminates.
Note	This use case is partially pre-implemented by the standard functionality of Maven. This functionality had been extended to support the instantiation of selected implementation alternatives from the corresponing varibility descriptor.

Table A.4: Use Case - Instantiate an Implementation Archetype from the Pattern Catalogue.

	Description
Name	*Configure Pattern*
Goal	The user wants to set the selected alternative values.
Actor	Workspace User
Pre-Condition(s)	The pattern is present in the workspace. All required variability models are present. The maven plugin is present.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The user starts the *configure project* goal. 2. The plugin checks if all required variab. descriptors are present. 3. The *effective variability descriptor* is computed. 3.1. The root variability descriptor is loaded. 3.2. All reachable nodes are *loaded*, *validated* and *aggregated*. 3.2.1 For each loaded variability descriptor: 3.2.1.1. The plugin validates all variability points. 3.2.1.2. The plugin evaluates all enabling conditions. 3.2.1.3. All child elements are merged into the result. 4. The *effective variability descriptor* is validated globally. 4.1. Check if cyclic dependencies exist. 4.2. Check if contradictory relations exist. 5. The configuration execution sequence is computed. 6. The configuration execution sequence is executed. 6.1. For each enabled variability point: 6.1.1. The selected alternative is extracted. 6.1.2. The extracted value is written into the target. 7. The plugin logs the successful configuration. 8. The plugin terminates.
Special Cases	1.a. The plugin is missing. a.) Maven raises an exception, that the plugin is unknown. 2.a. At least one project is missing its variability descriptor. a.) The plugin logs the absence and terminates. 3-4. Validation fails. a.) The plugin logs the violation and terminates. 5.a. Alternatives can not be set. a.) Set alternatives are revoked. b.) The plugin logs the reason and terminates.

Table A.5: Use Case - Configure Pattern.

	Description
Name	*Provision Nodes and Middleware*
Goal	The user wants to start the required nodes and provision the required middleware.
Actor	Workspace User
Pre-Condition(s)	The pattern is present in the workspace. The required provisioning scripts are present. The maven plugin is present.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The user starts the *Prov. Nodes and Middleware* goal. 2. Check if all required variability descriptors are configured. 3. Check if the referenced provisioning scripts are present. 4. The plugin traverses the pattern projects. 4.1. For each project: 4.1.1. Execute bound node starting plugins. 4.1.2. Execute bound provisioning plugins. 5. A message about the successful processing status is logged. 6. The plugin terminates.
Special Cases	1.a. The plugin is missing. a.) Maven raises an exception, that the plugin is unknown. 2.a. At least a variability descriptor is missing. a.) The plugin logs the missing descriptor. b.) The plugin terminates. 2.b. At least a variability descriptor is not configured. a.) The plugin logs the unconfigured descriptor. b.) The plugin terminates. 3. At least one required provisioning script is missing. a.) The missing provisioning script is logged. b.) The plugin terminates. 4. The provisioning process fails. a.) The plugin logs the reason and terminates. 1-4. An unchecked exception occurs. a.) The plugin logs the exception and terminates.

Table A.6: Use Case - Provision Nodes and Middleware.

	Description
Name	*Deploy Pattern*
Goal	The user wants to deploy the pattern related binaries into the specified cloud environments.
Actor	Workspace User
Pre-Condition(s)	The pattern is present in the workspace. The required maven deployment plugins are present. The maven deployment plugins are bound in their *pom.xml* files.
Post-Condition	The workspace plugin is ready for further actions.
Normal Case	1. The user executes the standard build life-cycle. 2. Maven builds the required binaries. 2.1 Maven traverses the projects 2.1.1 Each bound deployment plugin is executed. 2.1.1.1 The plugins authenticate themselves. 2.1.1.2 The plugins transfer the binaries. 4. A message about the successful processing status is logged. 5. The standard build life-cycle plugin terminates.
Special Cases	2.a. The maven build fails. a.) The deployment is not executed. b.) The plugin logs the build failure and terminates. 3.a. The authentication fails. a.) The deployment process is not executed. b.) The plugin logs the authentication failure and terminates. 3.b. The transfer of the binaries fails. b.) The plugin logs the transfer failure and terminates. 1-3. An unchecked exception occurs. a.) The plugin logs the exception and terminates.

Table A.7: Use Case - Deploy Pattern.

Note: The undeployment of a pattern, as well as the decommissioning of nodes and middleware, is done analogous to the use cases A.6 and A.7, by executing the corresponding undeployment actions, respectively decommissioning actions.

Variability Graph Appendix

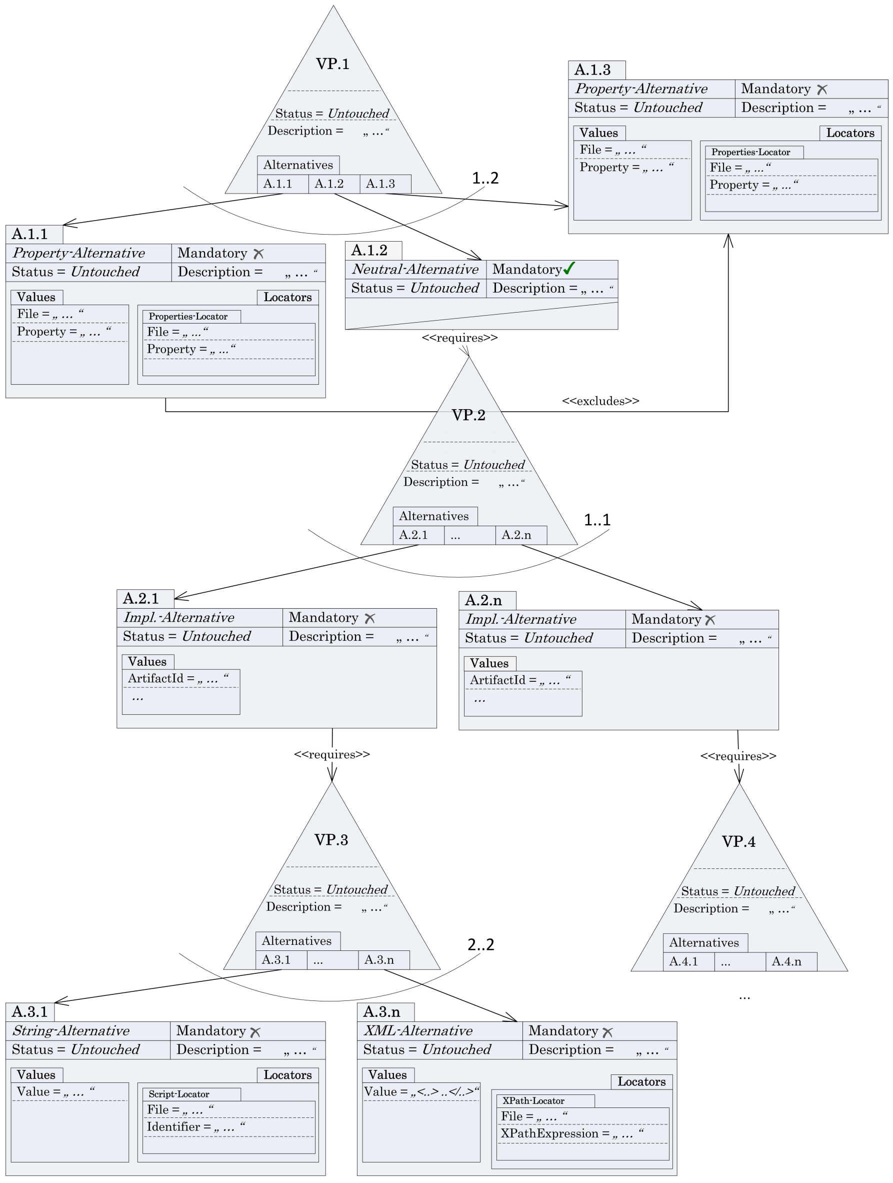


Figure B.1: Extended OVM Notation - Example Application Variability Model.

XML Schema Appendix

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>Represents a variability model, consisting of
      locators, alternatives and dependencies.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType name="adressable">
    <xs:attribute name="id" type="nonEmptyString" use="required" />
  </xs:complexType>
  <xs:complexType name="alternativeBase">
    <xs:complexContent>
      <xs:extension base="adressable">
        <xs:sequence>
          <xs:element name="name" type="nonEmptyString" />
          <xs:element minOccurs="0" name="description" type="nonEmptyString" />
          <xs:element minOccurs="0" name="dependencies">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="1"
                  name="dependency" type="dependency" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="status" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="untouched" />
              <xs:enumeration value="selected" />
              <xs:enumeration value="deselected" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="mandatory" type="xs:boolean" use="optional" default="false" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="locatorAlternativeBase">
    <xs:complexContent>
      <xs:extension base="alternativeBase">
        <xs:sequence>
          <xs:element name="locators">
            <xs:complexType>
              <xs:choice maxOccurs="unbounded">
                <xs:element name="xPathLocator" type="xPathLocator" />
                <xs:element name="propertiesFileLocator" type="propertiesFileLocator" />
                <xs:element name="scriptLocator" type="scriptLocator" />
              </xs:choice>
            </xs:complexType>
```

```
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="variabilityModel" type="variabilityModel" />
<xs:complexType name="variabilityModel">
    <xs:complexContent>
        <xs:extension base="adressable">
            <xs:sequence>
                <xs:element minOccurs="0" name="name" type="nonEmptyString" />
                <xs:element minOccurs="0" name="description" type="nonEmptyString" />
                <xs:element minOccurs="0" name="variabilityPoints">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element maxOccurs="unbounded" minOccurs="1"
                                name="variabilityPoint" type="variabilityPoint" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="variabilityPoint">
    <xs:complexContent>
        <xs:extension base="adressable">
            <xs:sequence>
                <xs:element name="name" type="nonEmptyString" />
                <xs:element minOccurs="0" name="description" type="nonEmptyString" />
                <xs:element minOccurs="0" name="tags" type="tags" />
                <xs:element minOccurs="0" name="enablingConditions">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element maxOccurs="unbounded" minOccurs="1"
                                name="enablingCondition" type="enablingCondition" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="alternativeCardinalities">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="minSelect" type="xs:unsignedLong" />
                            <xs:element name="maxSelect" type="xs:unsignedLong" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="alternatives">
                    <xs:complexType>
                        <xs:choice maxOccurs="unbounded" minOccurs="1">
                            <xs:element name="xmlAlternative" type="xmlAlternative" />
                            <xs:element name="propertyAlternative" type="propertyAlternative" />
                            <xs:element name="stringAlternative" type="stringAlternative" />
                            <xs:element name="neutralAlternative" type="neutralAlternative" />
                            <xs:element name="implementationAlternative" type="implementationAlternative" />
                        </xs:choice>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="status" use="required">
                <xs:simpleType>
                    < xs:restriction  base="xs:string">
                        <xs:enumeration value="untouched" />
                        <xs:enumeration value="enabled" />
                        <xs:enumeration value="disabled" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="requiresProvisioning" use="optional" type="xs:boolean" default="false"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="tags">
    <xs:sequence>
        <xs:element maxOccurs="unbounded" name="tag" type="nonEmptyString" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="enablingCondition">
    <xs:sequence>
        <xs:element name="condition" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="xmlAlternative">
    <xs:complexContent>
        <xs:extension base="locatorAlternativeBase">
            <xs:sequence>
                <xs:element name="value">
                    <xs:complexType>
                        <xs:choice>
                            <xs:any maxOccurs="unbounded" processContents="skip" />
                        </xs:choice>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="propertyAlternative">
    <xs:complexContent>
        <xs:extension base="locatorAlternativeBase">
            <xs:sequence>
                <xs:element name="relativePath" type="nonEmptyString" />
                <xs:element name="property" type="nonEmptyString" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="stringAlternative">
    <xs:complexContent>
        <xs:extension base="locatorAlternativeBase">
            <xs:sequence>
                <xs:element name="text" type="nonEmptyString" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="implementationAlternative">
    <xs:complexContent>
        <xs:extension base="alternativeBase">
            <xs:sequence>
                <xs:element name="archetypeGroupId" type="nonEmptyString" />
                <xs:element name="archetypeArtifactId" type="nonEmptyString" />
                <xs:element name="archetypeVersion" type="nonEmptyString" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="neutralAlternative">
        <xs:complexContent>
            <xs:extension base="alternativeBase" />
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="dependency">
        <xs:sequence>
            <xs:element name="target" type="nonEmptyString" />
            <xs:element minOccurs="0" name="enablingConditions">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element maxOccurs="unbounded" minOccurs="1"
                            name="enablingCondition" type="enablingCondition" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="type" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="requires" />
                    <xs:enumeration value="excludes" />
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:complexType name="xPathLocator">
        <xs:sequence>
            <xs:element name="xPath" type="nonEmptyString" />
            <xs:element name="relativePath" type="nonEmptyString" />
            <xs:element name="action" type="action" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="propertiesFileLocator">
        <xs:sequence>
            <xs:element name="relativePath" type="nonEmptyString" />
            <xs:element name="property" type="nonEmptyString" />
            <xs:element name="action" type="action" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="scriptLocator">
        <xs:sequence>
            <xs:element name="relativePath" type="nonEmptyString" />
            <xs:element name="identifier" type="nonEmptyString" />
            <xs:element name="action" type="action" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="action">
        <xs:restriction base="xs:string">
            <xs:enumeration value="replace" />
            <xs:enumeration value="before" />
            <xs:enumeration value="after" />
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="nonEmptyString">
        <xs:restriction base="xs:string">
            <xs:minLength value="1" />
            <xs:pattern value=".*[^\s].*" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Listing C.1: XML Schema - Representation of the Functional Variability Meta-Model.

Bibliography

[Act] Apache Software Foundation, Apache Active MQ. Apache Licence 2.0. URL http://activemq.apache.org/. (Cited on pages 15 and 78)

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977. URL http://www.amazon.fr/exec/obidos/ASIN/0195019199/citeulike04-21. (Cited on pages 23 and 24)

[Ale78] C. Alexander. *The Oregon Experiment*. Oxford University Press, New York, NY, 1978. URL http://www.amazon.fr/exec/obidos/ASIN/0195018249/citeulike04-21. (Cited on page 24)

[Ale79] C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979. URL http://www.amazon.fr/exec/obidos/ASIN/0195024028/citeulike04-21. (Cited on page 23)

[Ant] Apache Software Foundation, Apache Ant. Apache Licence 2.0. URL http://ant.apache.org/. (Cited on page 40)

[App] Google, Goole App Engine. URL http://www.cloud.google.com/appengine. (Cited on page 14)

[Art] JFrog, Artifactory. GNU Lesser General Public License. URL http://www.jfrog.com/. (Cited on page 78)

[bea] Ingenieux Labs, Maven Elastic Beanstalk Plugin. URL http://beanstalker.ingenieux.com.br/beanstalk-maven-plugin/. (Cited on page 85)

[BPM] Object Management Group, Business Process Model and Notation. URL http://www.bpmn.org/. (Cited on page 51)

[Che] Opscode, Chef. Apache Licence 2.0. URL http://www.opscode.com/chef/. (Cited on page 44)

[Clo] MuleSoft, CloudHub. URL http://www.mulesoft.com/cloudhub. (Cited on page 15)

[CVS] The CVS Team, Concurrent Versions System (CVS). URL http://cvs.nongnu.org/. (Cited on page 37)

[Dat] DIQA, DataWiki. URL http://diqa-pm.com/de/DataWiki. (Cited on page 92)

[Del] Apache Software Foundation, Apache Deltacloud. Apache Licence 2.0. URL http://deltacloud.apache.org/. (Cited on pages 72 and 78)

[EC2] Amazon Web Services, Amazon Elastic Compute Cloud (EC2). URL http://aws.amazon.com/ec2/. (Cited on pages 14, 15 and 78)

[Ecl] Eclipse Foundation, Eclipse. Eclipse Public License. URL http://www.eclipse.org/. (Cited on page 78)

[EKN+12] G. Engels, M. Kremer, T. Noetzold, T. Wolf, K. Prott, J. Hohwiller, A. Hofmann, A. Seidl, D. Schlegel, O. F. Nandico. *Quasar 3.0 - A Situational Approach to Software Engineering.* Capgemini sd&m Research, 2012. (Cited on pages 7 and 26)

[Ela] Amazon Web Services, Amazon Elastic Beanstalk. URL http://aws.amazon.com/elasticbeanstalk/. (Cited on pages 14, 15 and 78)

[Feh09] C. Fehling. *Provisioning of Software as a Service Applications in the Cloud.* Master's thesis, University of Stuttgart, 2009. (Cited on pages 7 and 15)

[FEL+12] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rutschlin, D. Schumm. Capturing Cloud Computing Knowledge and Experience in Patterns. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 726 –733. 2012. doi:10.1109/CLOUD.2012.124. (Cited on page 27)

[FLMS11] C. Fehling, F. Leymann, R. Mietzner, W. Schupeck. A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-based Application Architectures. Technical report, Institute of Architecture of Application Systems - University of Stuttgart and Daimler AG, 2011. (Cited on pages 3, 9, 10, 27 and 29)

[FLR+11] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck. An Architectural Pattern Language of Cloud-based Applications, 2011. (Cited on page 27)

[FLR+12] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*, volume 1. Springer, 2012. (Cited on pages 3, 7, 9, 10, 17, 27, 29, 79, 80, 86 and 87)

[FLRS12] C. Fehling, F. Leymann, J. Ruetschlin, D. Schumm. Pattern-Based Development and Management of Cloud Applications. *Future Internet*, 4(1):110–141, 2012. doi:10.3390/fi4010110. URL http://www.mdpi.com/1999-5903/4/1/110. (Cited on pages 27 and 46)

[For] SalesForce, Force. URL http://www.force.com/. (Cited on page 14)

[Fus] FuseSource, Fuse Fabric. URL http://fuse.fusesource.org/. (Cited on page 15)

[GBS01] J. van Gurp, J. Bosch, M. Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pp. 45 –54. 2001. doi:10.1109/WICSA.2001.948406. (Cited on page 20)

[Gep] Cloudsmith Inc., Gepetto. Apache end EPL. URL http://cloudsmith.github.io/geppetto. (Cited on page 78)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 edition, 1995. 37. Reprint (2009). (Cited on pages 25 and 87)

[Git] GitHub Inc., GitHub. URL http://github.com. (Cited on page 15)

[Goo] Google, Google Docs. URL http://docs.google.com. (Cited on page 15)

[Gra] Gradleware, Gradle. Apache Licence 2.0. URL http://www.gradle.org/. (Cited on page 40)

[Gri11] P. Grimm. *Metamodell und Plattform fuer Mustersprachen und Musterkataloge*. Master's thesis, University of Stuttgart, 2011. Language: German. (Cited on page 23)

[Her] SalesForce, Heroku. URL http://www.heroku.com/. (Cited on page 14)

[HT] J. Hamano, L. Torvalds. Git. URL http://git-scm.com/. (Cited on page 37)

[HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Cited on page 25)

[Ivy] Apache Software Foundation, Apache Ivy. Apache Licence 2.0. URL http://ant.apache.org/ivy/. (Cited on page 40)

[Java] Oracle, Java. GNU General Public License. URL http://www.java.com/. (Cited on pages 25 and 78)

[Javb] Oracle, JavaEE. GNU General Public License. URL http://www.oracle.com/technetwork/java/javaee. (Cited on page 78)

[JCl] JClouds Inc., JClouds. Apache Licence 2.0. URL http://www.jclouds.org/. (Cited on page 92)

[KM08] T. Kwok, A. Mohindra. Resource Calculations with Constraints, and Placement of Tenants and Instances for Multi-tenant SaaS Applications. In A. Bouguettaya, I. Krueger, T. Margaria, editors, *Service-Oriented Computing - ICSOC 2008*, volume 5364 of *Lecture Notes in Computer Science*, pp. 633–648. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-89652-4_57. URL http://dx.doi.org/10.1007/978-3-540-89652-4_57. (Cited on page 14)

[Knu98] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. (Cited on page 60)

[LSR07] F. J. v. d. Linden, K. Schmid, E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. (Cited on pages 7 and 22)

[M2E] Eclipse Foundation, m2eclipse. Eclipse Public License. URL http://eclipse.org/m2e/. (Cited on page 78)

[Mava] Apache Software Foundation, Apache Maven. Apache Licence 2.0. URL http://maven.apache.org/. (Cited on pages 3, 10, 33, 38, 39 and 78)

[Mavb] Apache Software Foundation, Apache Maven Wagon Plugin. Apache Licence 2.0. URL http://maven.apache.org/wagon/. (Cited on pages 78 and 88)

[Mie] R. Mietzner. Cafe Website. URL http://www.iaas.uni-stuttgart.de/forschung/projects/cafe/. (Cited on page 32)

[Mie08] R. Mietzner. Using Variability Descriptors to Describe Customizable SaaS Application Templates. Technical report, University of Stuttgart - Institute of Architecture of Application Systems (IAAS), 2008. (Cited on pages 7, 31, 32, 55 and 57)

[Mie10] R. Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. Ph.D. thesis, Universitaet Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2010. URL http://elib.uni-stuttgart.de/opus/volltexte/2010/5614. (Cited on pages 31, 32 and 61)

[MPH$^+$ct] A. Metzger, K. Pohl, P. Heymans, P. Schobbens, G. Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pp. 243–253. Oct. doi:10.1109/RE.2007.61. (Cited on pages 55 and 58)

[MUTL09] R. Mietzner, T. Unger, R. Titze, F. Leymann. Combining Different Multi-tenancy Patterns in Service-Oriented Applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, pp. 131 –140. 2009. doi:10.1109/EDOC.2009.13. (Cited on page 14)

[NIS] The NIST Definition of Cloud Computing. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. (Cited on pages 9, 13 and 29)

[Ope] OpenStack LLC, OpenStack. Apache Licence 2.0. URL http://www.openstack.org/. (Cited on page 86)

[OSI] OSI Reference Model. URL http://www.iso.org/iso/. (Cited on page 16)

[Oxf] Oxford English Dictionary, Archetype Definition. URL http://oxforddictionaries.com/. (Cited on page 18)

[PBL05] K. Pohl, G. Boeckle, F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (Cited on pages 55 and 58)

[Pez11] M. Pezzini. Integration Platform as a Service: Moving Integration to the Cloud. Technical report, Gartner, 2011. URL http://www.gartner.com/id=1575414. (Cited on page 15)

[Pup] Puppetlabs, Puppet. Apache Licence 2.0. URL https://puppetlabs.com/. (Cited on pages 3, 10, 42 and 78)

[Rac] Rackspace, Rackspace Cloud Services. URL http://www.rackspace.com/. (Cited on page 14)

[RH06] R. Reussner, W. Hasselbring. *Handbook of Software Architecture*, volume 2. Dpunkt Verlag, 2006. Language: German. (Cited on page 21)

[Sal] SalesForce, SalesForce. URL http://www.salesforce.com. (Cited on page 15)

[Sca] Scalr Inc., Scalr. Apache Licence 2.0. URL http://www.scalr.com/. (Cited on page 92)

[Ser] Apache Software Foundation, Apache ServiceMix. Apache Licence 2.0. URL http://servicemix.apache.org/. (Cited on page 15)

[Smi11] D. M. Smith. Hype Cycle for Cloud Computing. *Gartner Research*, 2011. (Cited on page 9)

[Spl] Carnegie Mellon Software Engineering Institute (SEI), Software Product Line Definition. URL http://www.sei.cmu.edu/productlines/. (Cited on page 18)

[SVN] Apache Software Foundation, Apache Subversion. URL http://subversion.apache.org/. (Cited on page 37)

[Toma] Apache Software Foundation, Apache Tomcat. Apache Licence 2.0. URL http://tomcat.apache.org/. (Cited on page 50)

[Tomb] Apache Software Foundation, Tomcat Maven Plugin. Apache Licence 2.0. URL http://tomcat.apache.org/maven-plugin.html. (Cited on page 78)

[TOS] Organization for the Advancement of Structured Information Standards (OASIS), Topology and Orchestration Specification for Cloud Applications (TOSCA). URL https://www.oasis-open.org/committees/tosca/. (Cited on page 86)

[VCl] VMWare, vCloud. URL http://www.vmware.com. (Cited on page 14)

[VVE10] T. Velte, A. Velte, R. Elsenpeter. *Cloud Computing, A Practical Approach*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010. (Cited on page 9)

[XML] World Wide Web Consortium (W3C), Extensible Markup Language (XML). URL http://www.w3.org/XML/. (Cited on page 75)

All links were last followed on July 31, 2013.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Ort, Datum, Unterschift