Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis No. 38

A Pattern Language for Modeling the Provisioning of Applications

Christian Endres

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dipl.-Inf. Uwe Breitenbücher M.Sc. Michael Falkenthal
Commenced:	March 05, 2015
Completed:	November 04, 2015
CR-Classification:	D.2.11

Abstract

In cloud computing, there are various technologies that automate the provisioning of cloud applications by employing different domain-specific languages and modeling techniques. These domain-specific languages and modeling techniques encompass different extents of functionality that the user have to know before the decision for or against a technology can be made. This master thesis contributes by introducing the Application Provisioning Modeling Pattern Language that enables the user to understand the underlying principles of the considered technologies, choose one technology according the requirements, and model the provisioning of the desired cloud application. The introduced Application Provisioning Modeling Pattern Language fosters the understanding of cloud application provisioning and works out the differences of the considered technologies as well as shows how the concepts can be combined. The Application Provisioning Modeling Pattern Language is validated by documenting systematically the occurrences of the concepts, their requirements and implications and by providing a statistical basis that quantitatively proves the repeatedly occurrence of the found principles.

Contents

1 Introduction 7
- 1.1 Problem Domain and Motivation 7
- 1.2 Research Issus and Contributions 8
- 1.3 Research Method 9
- 1.4 Structure of the Document 9

2 Fundamentals and Related Work 11
- 2.1 Patterns 11
  - 2.1.1 On Patterns and Pattern Languages 12
  - 2.1.2 A Process for Pattern Identification, Authoring, and Application 13
  - 2.1.3 Formulating Patterns 13
- 2.2 Patterns in Computer Science and Information Technologies 14
- 2.3 State of the Art Management Technologies 15
  - 2.3.1 Bluemix 16
  - 2.3.2 Chef 16
  - 2.3.3 Juju 18
  - 2.3.4 TOSCA and OpenTOSCA 18
  - 2.3.5 General-purpose Infrastructure, Platform and Cloud Provider Technologies and APIs 19

3 Research Design 21
- 3.1 Pattern Identification 21
- 3.2 Authoring Pattern 22
- 3.3 Pattern Application 24
- 3.4 Summary of the Adapted Process 25

4 Analyzed Artefacts 27
- 4.1 Chef Cookbooks 27
- 4.2 Juju Charms 28

5 Design of the Application Provisioning Modeling Pattern Language 31
- 5.1 Domain Definition and Constraints 31
  - 5.1.1 Domain Definition 31
  - 5.1.2 Domain Characteristic Problems 32

5.2 Information Format Design 33
5.2.1 Sequence Diagram 33
5.2.2 Graphical Notation of Icons and Sketches 33
5.3 Pattern Primitives Definition 35
5.4 Pattern and Pattern Language Design 38

6 Application Provisioning Modeling Pattern Language 41
6.1 Imperative Provisioning 42
6.2 Declarative Provisioning 45
6.3 Parametrized Imperative Provisioning 47
6.4 Local Management Operation Execution 50
6.5 Component Lifecycle Interface 52
6.6 Container Component Interface 54
6.7 Explicit Dependency Model 57
6.8 Implicit Dependency Model 59
6.9 External Instance Data Access 61
6.10 Overview of the Pattern Language 63

7 Discussion 65
7.1 Overview of the Known Uses of the Patterns and Pattern Candidates 65
7.2 Maturity Evaluation of the Patterns and Pattern Candidates 65
7.3 Threats to Validity 66
7.4 Limits of the Thesis 67

8 Literature and Other Resources 69
8.1 Literature 69
8.2 Online Resources 72

9 Appendix 79
9.1 List of Figures 79
9.2 List of Tables 80

1 Introduction

In this chapter, the background of this master thesis is introduced. First, in Section 1.1, the domain of cloud computing, application provisioning and their relevance are discussed as well as why there is need for further research. Subsequent, in Section 1.2, the research issues in the problem domain and the contributions are outlined. In Section 1.3, the research method is described briefly. Also, the structure of the document is outlined in Section 1.4 to provide an overview of this document.

1.1 Problem Domain and Motivation

Cloud computing is currently one of the major topics for companies that rely on or use information technology. Figure 1.1 depicts a current statistic about the usage of cloud computing in German companies. Between 403 and 458 executives of companies that have more than 20 employees took part in the surveys between 2011 and 2014. [KPM]

According to this statistic, cloud computing becomes more and more important. Contrary to this trend, it is not realistic to assume that all companies have dedicated experts responsible for the used cloud technology in combination with their custom business software. Cloud computing promises reduced costs and flexible usage that is fitted to the dynamically changing demand [Ley09]. Fulfilling these promises introduces new challenges. Also, cloud computing offers to book computing power and storage in a remote and public, on premise and private, or hybrid environment that is operated by oneself or a chosen provider. But booking computing power and storage does not address the issue of whom is responsible for provisioning and operating the custom business software, nor the issue of handling the hidden infrastructure stack behind an arbitrary cloud computing interface and the usage of that interface.

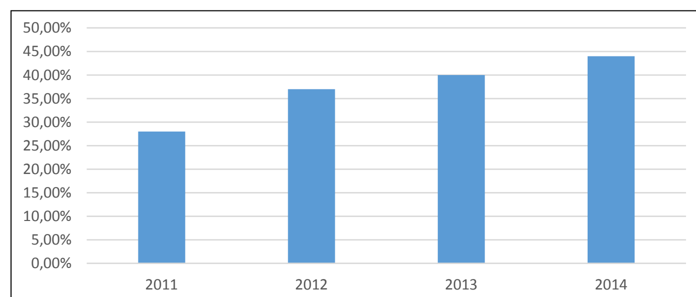


Figure 1.1. Statistic about the usage of cloud computing in German companies between 2011 and 2014 [KPM]

To tackle the issues of provisioning and operating software in the virtualized context of the cloud, there are different vendors and approaches. Famous and wide-spread representatives are, for example, Chef [Cheh] and Juju [Canm]. Both provide a means of more or less easy provisioning of applications in the cloud. New in this domain are the Bluemix called solution of IBM [IBMg] and the OASIS standard TOSCA [OASb] with the open-source ecosystem OpenTOSCA [Unia]. All these approaches tackle the issue of supporting users in their task of provisioning and operating custom business software. Therefore, these approaches aim for a high degree of automation of the provisioning and management of the applications, reusability of the used artifacts, and a usage as easy as possible.

But the aforementioned standard and technologies differ in their modeling and execution capabilities. The user has to model the application to be provisioned in the particular domain-specific language that the standard or technology enforces. Consequently, the domain-specific language are executed with different runtimes. Therefore, the users have to find out themselves the right technology for their particular job according the technology's functionalities and mechanisms. The master thesis aims for supporting the users in their understanding of the mechanisms, principles, and their combination. Consequently, the master thesis provides means to select the right technology regarding abstract requirements, a design method to model characteristics of the provisioning and the application to be provisioned in this technology, and a deeper understanding of the underlying concepts.

1.2 Research Issus and Contributions

The provisioning of custom applications in the cloud encompasses many aspects, for example, the provisioning of custom applications using the current technologies Bluemix, Chef, or Juju or the standard TOSCA in combination with OpenTOSCA. The research issues of this master thesis are the identification of often used mechanisms and the underlying principles regarding the provisioning of arbitrary applications using the aforementioned standard and technologies, the identification of the resulting requirements and implications, and the commonalities and structured combination of the identified principles. To document the findings, a pattern language has been formulated.

This master thesis contributes to the current research by providing a design method for modeling reusable cloud application components regarding their provisioning in the cloud using the aforementioned standard and technologies. The design method is provided in the form of the Application Provisioning Modeling Pattern Language that documents the identified principles, their combined usage and the supporting technologies. With this pattern language, the readers are enabled to model their desired application and the provisioning of this application, and to choose a standard or technology capable of realizing their model.

1.3 Research Method

The aim of the master thesis is to formulate a pattern language. The patterns, pattern candidate, and the pattern language follow the rules introduced and summed up by Buschmann et al. [BHS07]. To formulate the pattern language, the iterative pattern identification and writing process of Fehling et al. [FLRS+14] is used in combination with the writing process of Wellhausen et al. [WF12]. The details and rules of the methodology are introduced and defined in the Section 2.1 and Chapter 3.

To ensure the correctness and existence of the introduced patterns, they are evaluated by explicitly stating their occurrence in the *known uses* Paragraph of each pattern in Chapter 6. With this approach, each pattern fulfills the rule of three occurrences for being a pattern [CA96]. If patterns occurred less than three times, they are marked as pattern candidates. Counting occurrences are those in the aforementioned standard TOSCA and the technologies Bluemix, Chef, Juju and OpenTOSCA and in addition other technologies, for example, the Apache Tomcat. Scientific work may contribute with additional evidences, but do not count because they are no dedicated solutions for provisioning cloud applications or are widely used in that context.

1.4 Structure of the Document

The master thesis is structured as follows: In the first chapter, domain and research question are stated. In Chapter 2 the fundamentals and related work are introduced. First the related work covering patterns and pattern languages is introduced, followed by motivating examples of patterns in the domain of computer technology and computer science, and an overview of the considered technologies is outlined. Subsequent, in Chapter 3, the research design is defined, followed by the analysis of core artifacts in Chapter 4. The utilized research process produces besides the pattern language itself a set of definitions of the analyzed domain, the documentation format, pattern primitives, and the pattern and pattern language design. These results are documented in Chapter 5. The Chapter 6 documents the pattern language with its patterns and pattern candidates. The results, threats to validity, and the limits of the master thesis are discussed in Chapter 7, followed by the bibliography in Chapter 8.

2 Fundamentals and Related Work

This chapter introduces the fundamentals and existent works on which this master thesis bases. First, this introduction outlines the pattern topic in general. Subsequent, some pattern works in the information technology context are presented to provide motivating examples for this thesis. Finally, this chapter sketches key management technologies for cloud applications in which, as well as in scientific work, patterns may be found.

2.1 Patterns

Many pattern authors point to Christopher Alexander for being the founder of the pattern movement, for example, Coplien et al. [CA96]. In 1977, Christopher Alexander et al. laid the foundations for patterns and pattern languages by publishing the two books covering patterns about "architecture, building and, planning" [AIS77; Ale79] and pattern thinking. Alexander points out that thinking in patterns is a process that considers the living of and in the things that are described by patterns, that patterns are precise but not directly applicable, so simple that it seems to be childish to point out, and structured but always different like languages. In the context of architecture, there are many patterns. On the first sight, some are more concrete, for example, the "beer hall" [AIS77] and some more abstract, for example, the "tapestry of light and dark" [AIS77], but all describe abstract principles of mechanisms that can be used to achieve a certain effect. [AIS77]

The impact of this kind of thinking is also concise for disciplines that have nothing to do with architecture, for example, the computer science. There are even conferences about patterns and exploiting patterns in the field of computer science, for example, the PLoP [Theb] or in Europe the EuroPLoP [Thea]. The "most relevant" [BHS07] pattern publications about the pattern concepts in computer science are the works of Buschmann et al. [BMRS+96], Coplien et al. [CA96], and Gamma et al. [GHJV94]. The book *A system of patterns: Pattern-oriented software architecture* [BMRS+96] of Buschmann et al. is the first book of the so-called POSA series that covers knowledge of patterns and concrete patterns in the field of computer science. The *Design patterns: elements of reusable object-oriented software* [GHJV94] of Gamma et al. could be the most famous book about software engineering patterns. At least these patterns are taught at the University of Stuttgart as foundation for good software engineering. Also, Coplien et al. contributed to the pattern movement by stating that "a good pattern should have

three examples that show three insightfully different implementations" [CA96]. This rule of thumb describes the obstacle between pattern candidates and patterns in this master thesis.

In Section 2.1.1, the lessons learned of the book *Pattern-oriented Software Architecture: On Patterns and Pattern Language* [BHS07] are outlined: Buschmann et al. documented their years of experience with writing, authoring, and shepherding patterns. This book provides comprehensive knowledge about the pattern concept, patterns and pattern languages, besides other forms of pattern collections. The paper "A Process for Pattern Identification, Authoring, and Application" [FBBL15] introduces a process for writing patterns in a structured, iterative manner. Also, the authors point out to analyze software artifacts, for example, source code or documentation, in addition to solely interview experts to find evidences for patterns. The Section 2.1.2 and Chapter 3 describe the details and how the process is adapted for this master thesis. The paper "How to Write a Pattern?: A Rough Guide for First-time Pattern Authors" [WF12] proposes an approach for formulating a pattern that is described in Section 2.1.3 more detailed.

2.1.1 On Patterns and Pattern Languages

The book *Pattern-oriented Software Architecture: On Patterns and Pattern Language* [BHS07] focuses on patterns, what they should be and what not, and how they form pattern languages. Towards well-built patterns, Buschmann et al. define patterns as follows:

Existing experience and best practice to solve certain problems can be documented as Patterns. The formulated patterns should be independent of distinct project details, constraints, languages or paradigms. Patterns are abstract and never on the level of concrete objects, classes or components. Patterns may contribute as a vocabulary to the transfer of knowledge about design concepts. Patterns can contribute to document software architectures as well as to design a software for distinct characteristics. [BHS07]

Contrary, the authors identified misapplication of or misconceptions about patterns: Not all artifacts or solution attempts of software developers are patterns. Also, not all smart designs or single design decisions are patterns. Patterns are not static pieces nor guidelines and are not necessarily easy to understand, thus not easily to apply. Also, patterns may not contribute to solve a given problem, but may be chosen nonetheless. Patterns cannot contribute to new domains. Patterns are not components and cannot be applied directly. Patterns are not buzzwords. [BHS07]

In addition to the patterns that are standing for themselves, pattern languages form a concept of "systematic application of patterns" [BHS07]. A pattern language consists of "tightly integrated patterns" [BHS07] and should answer to the following questions: Which domain is addressed by the pattern language and what are the driving forces? Which problem areas are tackled by the pattern language? How is the structure of the pattern language and are there paths

between the single patterns which the reader can or should follow? What semantic have the links between patterns that form the paths? [BHS07]

2.1.2 A Process for Pattern Identification, Authoring, and Application

In [FBBL15], a process is introduced that guides authors to identify commonalities, formulate them to pattern, and to work the patterns up for better usability. Figure 2.1 depicts the three main phases that the paper defines. The phases are iterative, the second depends on the results of the first phase. In contrast, the third phase is independent as soon as patterns are produced. However, the third phase is included in the iterative circle.

The first main phase serves to identify commonalities that could be patterns. The results of the phase are a definition of the target domain and the constraints limiting it, how the analyzed data has to be prepared for better processing, a vocabulary of specific language elements of the domain and the collected information in which the pattern can be found.

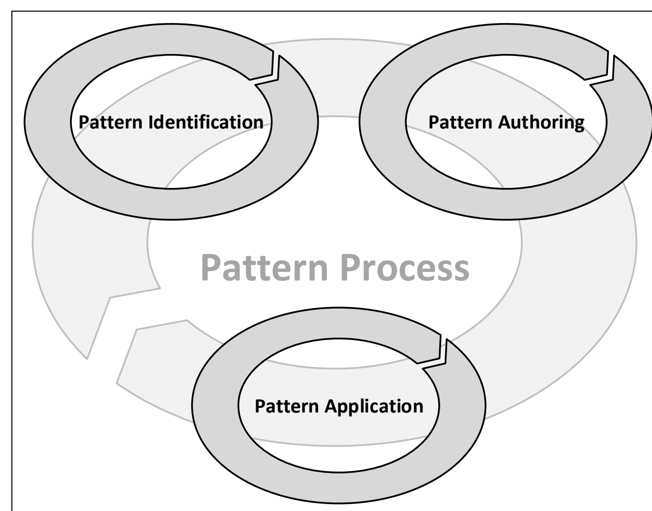


Figure 2.1. Sketch of the Pattern Identification, Authoring, and Application Process [FBBL15]

In the second phase, the found commonalities are authored to patterns. This results into the definition of the documentation structure, the revision of the already existing vocabulary, the definition of how this vocabulary can be composed to patterns, the actual patterns and how they are interrelated.

The third phase helps to make the found pattern applicable for pattern users. The produced results are a better searchable summary of the patterns, reference implementations or references to known implementations, guidelines to apply the patterns in the specific problem domain, and techniques and tooling for reducing the manual effort of the pattern users.

2.1.3 Formulating Patterns

In their paper “How to Write a Pattern?: A Rough Guide for First-time Pattern Authors” [WF12], the authors Wellhausen et al. propose an approach of how to write patterns. In contrast to Fehling et al., the approach is not an overall process, but the detailed steps of writing the pattern.

A pattern often starts with a statement about the problem that should be solved or the context in which the pattern can be found and ends with the results of applying the pattern. Contrary to this reading direction, the authors advise to write the pattern in another succession. The approach is depicted in Figure 2.2.

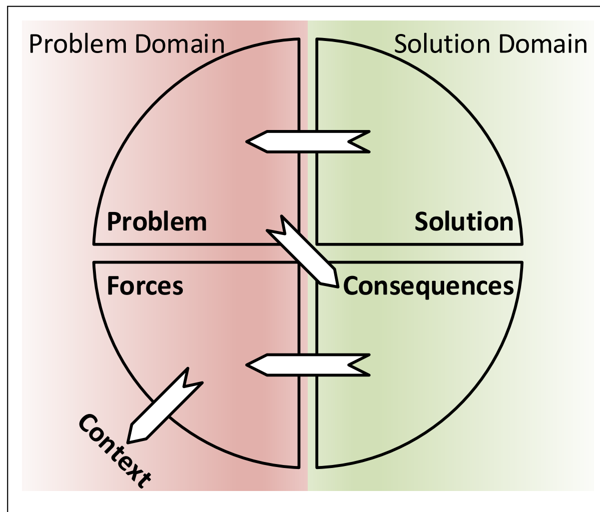


Figure 2.2. Sketch of the pattern writing approach [WF12]

In search of patterns the pattern authors do not find the problem or context first. Instead, repeating occurrences of slightly different implementations are found. These implementations should be documented on an abstract layer, forming the solution part of the pattern. Based on that solution the problem statement can be formulated by answering the question of what the solution solves or why it should be considered. But the authors should avoid trivial problem-solution pairs, for example, "*How to do* x*?*" and "*Do* x*!*". Knowing the solution and the problem, the authors can aim for enumerating the consequences of the pattern application. This encompasses the benefits as well as the drawbacks. Further, to state the *forces* Paragraph completes the depicted circle in Figure 2.2. The forces are sometimes called motivation and answer the questions of why the solution is difficult to apply and why other solutions are not suitable. Encompassing the problem statement and forces the context can be formulated. It should state the pattern's prerequisites without the problem cannot be found. Last but not least, the pattern needs a name that should be short and easy to remember as well as fitting to describe the proposition of the pattern. [WF12]

2.2 Patterns in Computer Science and Information Technologies

The Section 2.1 outlined patterns in general and approaches of how to acquire and formulate them. In this section, some motivating examples of pattern works in the field of computer science and information technologies are described.

The first pattern work to mention is the pattern catalog of the so-called *Gang of Four*, published in the book *Design patterns: elements of reusable object-oriented software* by Gamma et al. [GHJV94]. Gamma et al. introduce their patterns in three classifications: creational, structural, and behavioral. The creational patterns describe how to hide the creation process and free the rest of the application from the dependency of how the described component has to be created. One basic pattern of this class is the *Singleton* pattern that describes exactly one,

globally accessible object. The structural patterns describe compositions of components to a greater application. One representative of this class is the *Adapter* pattern that describes how to enable the collaboration of two components that originally are not able to because of their interface design. The third pattern class describes behavior patterns that describe how components and their interaction are designed. One famous behavior pattern is the *Observer* pattern that describes a one-to-many relation of one observed component to other observer components. This relation depicts a mechanism of propagating a state change of the observed component to the other observer components and enables the observer components react on the state change. These three patterns and the others described in Gamma et al. may be the basic literature for each adept of software engineering. [GHJV94]

With their book *A system of patterns: Pattern-oriented software architecture* [BMRS+96], the authors Buschmann et al. describe a catalog of patterns that enables the users to design software applications. Similar to the pattern book of Gamma et al., the authors divided their patterns into three classifications: the architectural patterns, the design patterns, and idioms. The architectural patterns describe possible structures of applications on the abstraction level of the software architecture. Well-known examples are the *Model-View-Controller* pattern or the *Pipes and Filters* pattern. The design patterns describe the interplay of components and are more fine-grained as architectural patterns. The *Proxy*, for example, is a design pattern which decouples the client from the actual component that should be not accessible directly. These patterns are on a high abstraction layer and can be applied to different object-oriented program languages, for example, Java. Contrary, there are idioms that describe principles on a low level according their abstraction. They are specific to a program language and describe characteristics of a component or the interplay between components. Buschmann et al. state as examples for idioms the two different *Singleton* implementations in *C++* and *Smalltalk*. [BMRS+96]

The book *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications* [FLRS+14] describes architectural patterns for cloud computing. The superior question is how to design a cloud application architecture according common goals of cloud computing strategies. Therefore, Fehling et al. describe categories of patterns, for example, the decomposition of components according the distribution of functionality or the component characteristics according the faced workload. The patterns of this pattern language can be used by the reader as a toolbox for designing cloud application architectures to face common challenges in the cloud environment. [FLRS+14]

2.3 State of the Art Management Technologies

This section introduces provisioning and management technologies for cloud applications. This introduction serves to get a better understanding of the technologies, but does not provide all information contained in the respective documentations. The details of the technologies,

with which the patterns were found, are documented in combination with this section and the *known uses* Paragraph in each pattern section in Chapter 6.

2.3.1 Bluemix

Bluemix is a cloud platform service of IBM with which users can host their application in the cloud. Bluemix offers multiple platforms for different types of applications, for example, Java, PHP, or Go as well as container solutions or virtual machines. Also, Bluemix offers enterprise solutions and services, for example, to integrate applications with iOS, Internet of Things, context mining, or hybrid cloud applications. Bluemix is able to host various offerings from Bluemix itself, third parties and the community. [IBMd; IBMh; IBMj]

In Bluemix an application is called App and is developed and provided by the user. An App can be provisioned into a runtime which is provided by Bluemix. Bluemix offers so-called services to enable the App using other functionalities, for example, a database or caching, which the user has not to provide. Boilerplates encompass multiple components, for example, a predefined App, a respective runtime, and services for a distinct domain. A buildpack defines dependencies of the App to, for example, other services. [IBMd]

There are two ways of managing utilized services of Bluemix: On the one hand, the users can manage their applications using the graphical user interface [IBMj]. On the other hand, users can exploit the *cf command line interface* of Cloud Foundry on which Bluemix bases [IBMc]. Additionally, Bluemix supports DevOps approaches by offering dedicated DevOps services, for example, a delivery pipeline for better software development [IBMl].

2.3.2 Chef

Chef is a software that enables the automated provisioning and management of cloud applications [Cheh]. The cloud applications are described with cookbooks and recipes. The machines can be any physical or virtual machines that runs the chef-client. Therefore, with Chef almost any bootstrapped machine can be utilized. If not cited otherwise, the information for this section can be found at [Chej].

Chef uses a central Chef server which is responsible for the management of Chef resources, for example, cookbooks, the metadata about the application, and to respond to requests of the chef-client for configuration details [Chei]. The chef-client is the agent which runs in the application environment. The chef-client requests all needed configuration information from the Chef server, compiles the specific sequence of management tasks to apply in the application environment and processes them. This specific sequence of management tasks is called run-list and is built anew for each chef-client run.

The cookbooks are the basic resources to describe configuration characteristics of components. The cookbooks also contain the resources needed for the configuration characteristics, for example, recipes, policies, attributes, and files [Cheb]. Attributes describe details about applications, for example, the current IP address. The attributes are managed by the Chef server. The chef-client collects additional resources during its run from the Chef server or ad-hoc values from Ohai. A cookbook can be used by another cookbook which needs functionality from the aforementioned cookbook. Therefore, a cookbook can expose its functional capabilities via an API like definition called *Resources*. These Resources are implemented by *Providers* which represent distinct management operations to realize the desired functionality represented by the Resource, for example, on different operating systems.

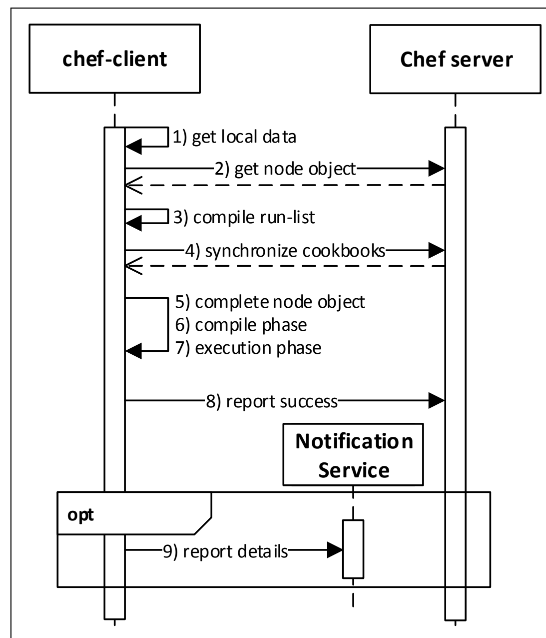


Figure 2.3. Chef-client configuration process of an application [Chej]

The configuration steps taken by the chef-client are depicted in Figure 2.3 and described by the following enumeration:

1. All locally existing configuration information like the node name are collected. Sources are the client.rb file and Ohai.
2. If available, the remote configuration information are downloaded from the Chef server and rebuild. This configuration information collection is called node object and contains also the previously processed run-list or a default run-list.
3. The list of roles and recipes is compiled in their exact order to apply. The sequence is stored in the run-list of the node object.
4. The cookbooks and resources required to apply the run-list are loaded.
5. The local node object is completed by finalizing the attribute data stored in it. The previous attributes are reset and updated with the current values
6. The resource collection is compiled and loaded.
7. The chef-client invokes actions to alter the bootstrapped machine according the node object.

8. If the execution phase is successful, the node object of the Chef server is replaced with the local node object.
9. Report details to a notification service like mail or logging.

2.3.3 Juju

Juju in version 1.24 promises to "[m]odel, build and scale your environments on any cloud" [Canm]. Similar to Chef, the user is able to provision applications on physical machines and virtual ones in the cloud on the service level and not on the machine level. In addition to its own capabilities Juju also supports Ansible, Chef, Docker, and Puppet. If not cited otherwise, the information in this section can be found at [Cana; Canb].

With Juju one can model an abstract service landscape. *Services* interacting with each other are modeled with *relations* between them. Due these relations the services can react on *events* with which is signaled, that the application has been altered. This is used from the very start on, also during the provisioning of the application. If an instantiation event is triggered, a so-called *hook* is executing the represented code to deal with the new situation. Such service landscapes can be modeled with so-called Juju bundles. [Cana; Canc; Cani; Cank]

The service landscape encompasses modeled *services*. With Juju these services are defined in *charms* which contain the "application-specific knowledge" [Cana] like integration options, dependencies, or events. The services can be connected with each other using relations. If an event happens, for example, triggered through such a relation, hooks are activated. Juju uses an event-driven approach for structuring the execution sequence of management tasks as well as enabling reusability. If two services are connected with each other and one has a change, for example, regarding its lifecycle, the other service is able to react by letting the *unit agent* execute the according hook. [Canc]

2.3.4 TOSCA and OpenTOSCA

The "Topology and Orchestration Specification for Cloud Applications" [OASb] (TOSCA) is available in version 1.0 since November 2013. If not cited otherwise the information in this section can be found at [OASa; OASb]. As the naming suggests there are two main focuses aimed for in this specification: the definition of cloud application topologies and the orchestration of those.

The definition of a cloud application structure is graph-based: The *topology template* consists of *node templates* and *relationship templates*. Node templates represent a component of the application, for example, an operating system or a web application. These node templates can be connected with relationship templates. Relationship templates describe a logical connection between two Node Templates, for example, a web application *is hosted on* an

Apache Tomcat or a web application *connects to* a MySQL database server. Both node templates and relationship templates are typed to enable reusability. The composition of the node templates and relationship templates is called a service template. The service template is the blueprint for a specific, reusable service.

Node types and relationship types enables to define reusable characteristics. The node type defines characteristics for a node template, requirements, capabilities, and its interfaces. The relationship type defines the characteristics of a relationship template, which source and target interfaces are available, and which source and target nodes are valid. Nodes and relations are completed by the respective *node type implementations* and *relationship type implementations* that contain the executables like code, scripts, and files.

In addition to the declarative definition of the topology by the topology template, the service template is able to implement its management operations imperatively. Therefore, process models are exploited, for example, BPEL 2.0 or BPMN 2.0. These management operations are exposed in the *boundary definitions* of the service template to enable users or agents to manage the described service.

The academic prototype OpenTOSCA implements TOSCA. OpenTOSCA is developed at University of Stuttgart since 2012. OpenTOSCA is an ecosystem for TOSCA cloud services packaged in so-called *CSARs*. The *OpenTOSCA container* is the runtime environment with which the CSARs can be deployed. With the *Winery*, the user can graphically model TOSCA applications. The self-service portal for TOSCA applications is called *Vinothek*. [BBHK+13; Unia]

2.3.5 General-purpose Infrastructure, Platform and Cloud Provider Technologies and APIs

In this section, other technologies that provide evidences for found patterns are briefly outlined.

Infrastructure Provider: Everything computed runs eventually on physical machines, the so-called bare metal. Besides other advantages, this hardware layer can be virtualized to better distribute workload. One provider is VMware with the products ESX, ESXi and vSphere. With these technologies, virtual machines can be hosted. The virtual machines behave like common, physical machines. Only the desired operating system has to be able to work on the virtualized hardware. [VMw]

Platform Provider: In the cloud, there are multiple offerings for platforms that enable users to provision and host easily their applications. One example is Docker that follows the concept of containers: Every application and its dependencies are placed inside a container. Instead of a general-purpose hypervisor layer like with VMWare, every container is placed on the

Docker Engine. Therefore, the portability and efficiency can be enhanced because there are no different, underlying infrastructure or hypervisor layers to consider. [Docg]

JBoss and Tomcat follow a more targeted approach. While with Docker the contained and hosted application has only to fit into a container, JBoss and Tomcat focuses on distinct application types, for example, WAR applications. [Apab; Reda; Redb]

Cloud Technologies: The aforementioned technologies have in common that they host other applications, for example, located in the data center of a company. The same principle can be found in cloud technologies: A virtualized layer hosts other applications. One example is Bluemix that was introduced in Section 2.3.1. Bluemix does not only enable to host Java web applications like with Tomcat. Also, Bluemix enables to host multiple applications or services that are in interplay in the same environment. Therefore, the users are able to develop their applications with a high degree of homogeneity despite the heterogeneity of the used components.

Workflow Engines: Workflow engines are able to process workflows that describe, for example, sequences of activities or tasks. Activiti is an example for processing BPMN 2.0 processes and Apache ODE processes BPEL 2.0 processes. Both BPEL and BPMN are imperative workflows that describe in detail what has to be done in which order. [Alfa; Apaa]

3 Research Design

This chapter describes the research design of the master thesis. The research design bases on the process introduced by Christoph Fehling et al. [FBBL15] and is shortly described in Section 2.1.2. The Sections 3.1 to 3.4 describe how the steps are performed. The iteratively produced definitions and designs are documented in Chapter 5: In Section 5.1 the domain of the pattern language is defined as well as the constraints limiting the domain to the specific topics considered by this master thesis. In Section 5.2, the format is introduced that is used to document the collected information. Part of the *information format design* as well as the *pattern language design* is a vocabulary of *pattern primitives* that are defined in Section 5.3. These pattern primitives are parts that each describe one specific characteristic or element in the target domain. In Section 5.4, the design of the patterns and the pattern language is defined.

3.1 Pattern Identification

The first main phase "pattern identification" [FBBL15] describes the goal to find reoccurring solutions that have the potential for being patterns. Figure 3.1 depicts the phase with its steps, each described in the next five paragraphs.

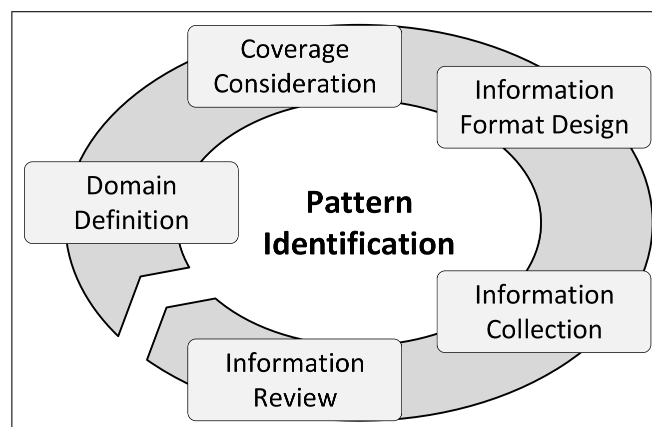


Figure 3.1. Sketch of the phase *Pattern Identification* [FBBL15]

Domain Definition: First, the targeted domain has to be defined and documented. This serves to create a common knowledge about the target domain and enables persons to use a common taxonomy and terminology to describe domain characteristics. The results are refined in each iteration and are documented together with the results of the next step in the Section 5.1.

Coverage Consideration: To cover a whole domain may not be feasible depending on its size and complexity. Thus, the amount of information can be reduced by focusing on specific and relevant topics or by formulating constraints to the domain. A possible result can be a domain structure and characteristic problems of the domain. The pattern then should provide a solution to the characteristic problems. These considerations are documented in Section 5.1.

Information Format Design: Depending on the domain, there can be prevalent language elements and concepts. To process and express information homogeneously a format design is defined in this step. This serves to enable multiple people to express their individual findings in a unified way. For the reader the perception is eased. [Zdu07] These defined elements are referred to as pattern primitives and are documented and iteratively refined in Section 5.3.

Information Collection: After defining the terminology of the found information, the information can be gathered and documented. One way to identify and elaborate patterns is to gather experts of the domain as information source and let them discuss the best solutions. On the one hand, it may be difficult to gather these experts and, on the other hand, the knowledge about the domain may be persisted also in existing artifacts produced by these experts. Thus, as proposed by Fehling et al. [FBBL15], an alternative approach to gather information as basis for the pattern identification is to document existing solutions and artifacts in the previously defined information format. The introduced process relies on this approach for the information collection, also to meet the limitations of a master thesis. The gathered information are documented in Section 2.3 and Chapter 4.

Information Review: At this point of the process the information collection may encompass many solutions to characteristic problems of the domain. The amount of information may be not feasible to be considered completely. Thus, the domain structure is to be refined towards smaller, manageable sets of solutions. Also, similar or duplicate solutions can be identified and grouped or pruned.

3.2 Authoring Pattern

After the information basis is collected and refined, patterns can be extracted out of similarities of existing solutions. Figure 3.2 depicts the phase and its steps.

Pattern Language Design: Although there are similar documentation formats following a structure like "intent", "forces", "driving question", and "context", a format fitting all domains generally does not exist [AIS77; GHJV94; Han12; Zdu07]. Thus, the custom design is basing on well-established pattern formats and has to be adjusted to specific needs of the targeted

domain. The pattern language design that is conducted in this step and used in this master thesis is introduced in Section 5.4.

Primitive Definition: In addition to the pattern primitives defined in the information format design step of the main phase *pattern identification*, there may be the need of new pattern primitives for the pattern description. Thus, the catalog of pattern primitives is revised and extended. Also, the results are documented in Section 5.3.

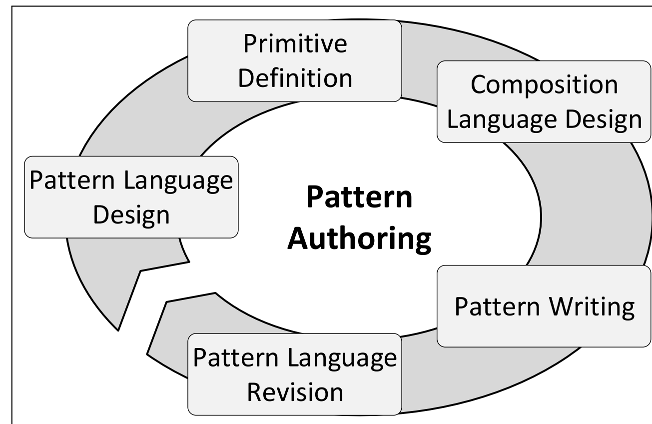


Figure 3.2. Sketch of the phase *Pattern Authoring* [FBBL15]

Composition Language Definition: In this step, the graphical elements and their composition are defined. The definition serves a common look and feel. The formalization of the composition enables a verification of correctness. The resulting definition is documented in Section 5.2.

Pattern Writing: After identifying different solutions and defining a common pattern format, the pattern writing begins. This step serves mainly the creation of a pattern. After this initial description, the pattern is discussed by other pattern authors and users. The pattern can be discussed and refined by a much broader community because to the high abstraction level of a pattern in contrast to the solutions it was extracted of. In later iterations, the existing patterns are revisited and compared to solutions which are found after the last refinement of the pattern. In this step, the collaboration with other authors and users is assumed [FBBL15]. The results were discussed with experts of the Institute of Architecture of Application Systems at the University of Stuttgart in form of the mentoring during the conduction of the master thesis. Unfortunately, it is not possible in this master thesis to conduct workshops, for example, at a pattern conference or with experts of multiple companies working in the domain. The documentation of the pattern can be found in Chapter 6.

Pattern language revision: The patterns found are not isolated. They are interconnected and form relations like *alternative to* or *composable with* [BHS07; FBBL15]. Due to the iterative nature of this approach, the amount of relations is growing with the amount of found patterns. In this step, the task is to revision existing relations and check for new relations between patterns as well as check if the relations apply unidirectional or bidirectional.

3.3 Pattern Application

Whilst the preceding main phases *pattern identification* and *pattern authoring* iteratively identify and document new pattern and their interrelations, the phase *pattern application* serves to improve the access and application of the pattern for pattern user. This can be done independently of the other two phases, as soon as there are patterns found. Figure 3.3 depicts the phase and its steps.

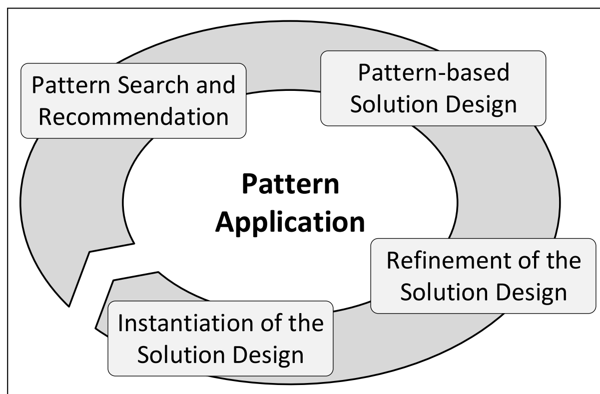


Figure 3.3. Sketch of the phase *Pattern Application* [FBBL15]

Pattern Search and Recommendation: The detailed description of patterns has to be summarized. This enables the user to review pattern information more quickly. After that, the detailed documents and relations between patterns can be read to get the full set of information. These summaries are the introducing descriptions of the Chapter 6. To simplify the process model, this step will be done with the *pattern writing* step of the *pattern authoring* phase.

Pattern-based Solution Design: Each pattern is an abstraction of multiple concrete solutions. Due this abstraction, a pattern is not applicable at once and has to be adapted to the concrete problem and environment at which the user aims. To support the user, "reference implementations" [FBBL15] or documentation about existing solutions can be provided. This is done in the *known uses* Paragraph of each pattern in Chapter 6. To simplify the process model, this step will be done with the *pattern writing* step of the *pattern authoring* phase.

Refinement of the Solution Design: The problem tackled by this step is the heterogeneity of the infrastructure stack in which the solutions are implemented. To reduce this cost driver the paper suggests to not only constraint the environment, but also the pattern. In the best case, this results into the developer being supported by "automated infrastructure management, deployment functionality and code templates" [FBBL15]. The aim of this master thesis is to identify and author new pattern. Therefore, this step is skipped.

Instantiation of the Solution Design: The patterns are introduced to techniques and tools for configuring, and deploying pattern refinements. This would reduce the amount of manual and redundant tasks for the pattern users. Therefore, the found pattern language is transferred

into the pattern tool *PatternPedia* [Unib]. This is done as a follow-up to the creation of this document, therefore, this step is not included into the process.

3.4 Summary of the Adapted Process

In the Sections 3.1 to 3.3, the three phases of the underlying process are described. The process enables to work collaboratively on the patterns, which is not the case for this master thesis. Also, two steps of the third phase are skipped. Therefore, the process can be simplified, as aforementioned. The Figure 3.4 depicts the adapted process. The *pattern identification* phase concentrates on finding commonalities for solving the same or similar problems. The *pattern authoring* phase puts the found data on an abstract level to improve the perception of pattern users. Both phases are realized in the previously described, iterative manner. Additional the master thesis encompasses to learn new technology. Therefore, the knowledge has to be obtained continuously which is depicted in the lower arrow in Figure 3.4.

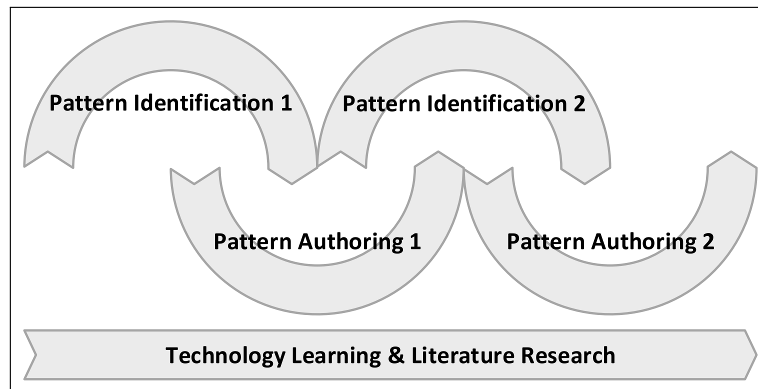


Figure 3.4. Sketch of the Adapted Process

The step *pattern search and recommendation* provides a summary of single patterns as well as pattern groups. The step *pattern-based solution design* provides one or multiple reference implementations of the pattern or at least references to already existing ones. The results of both steps are of great value for this master thesis, therefore, they are integrated into the *pattern authoring* phase. This results in the *adapted pattern authoring* phase which is depicted in Figure 3.5.

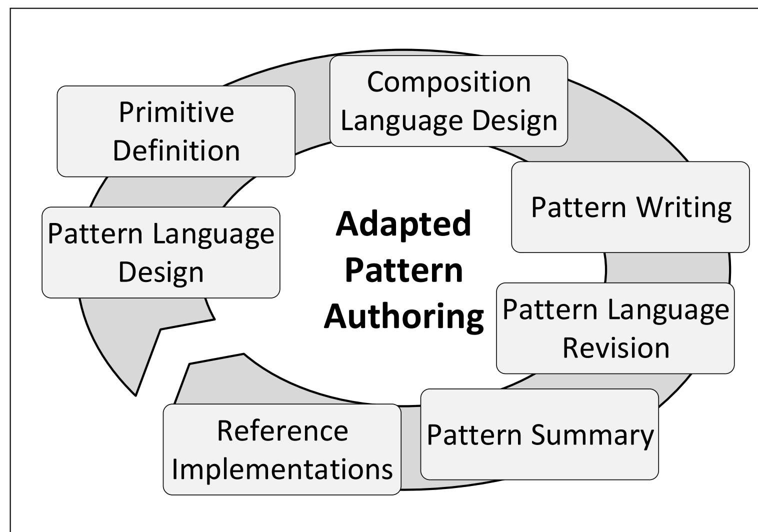


Figure 3.5. Sketch of the Adapted Pattern Authoring Phase

4 Analyzed Artefacts

In this section, the artifact analysis is introduced. To analyze artifacts with a preferably high degree of significance the artifacts are selected according their distribution. For Chef and Juju there are public marketplaces [Canl; Chel], which provide to sort the offered artifacts according their download or instantiation count. Bluemix offers many artifacts that comply with the functionality of the *cf command line interface* [IBMc]. OpenTOSCA is an academic prototype that has no marketplace yet. Therefore, for the latter two technologies no dedicated artifact analysis was conducted.

The next two subsequent sections describe the analyzed artifacts, how they were selected in detail, and which management capabilities they expose.

4.1 Chef Cookbooks

In this section, the ten most downloaded cookbooks of the Chef marketplace are analyzed [Chel]. Unfortunately, the sixth cookbook *bacon*, which is just a dummy application, is not suitable for this analysis [Var]. Thus, bacon is skipped in favor of the eleventh cookbook *artefact*. In Table 4.1, the ranking regarding the count of total downloads is shown. The analyzed sources can be downloaded from the respective pages cited in Table 4.1.

No.	Cookbook	Owner	Version	Downloads
1	Mysql [Cheo]	Chef Software, Inc.	6.0.22	105,353,280
2	Java [Agi]	Agile Orbit	1.30	63,296,631
3	Apache2 [vZoe]	van Zoest	3.1.0	62,748,068
4	Docker [Fla]	Flad	1.0.0	62,716,620
5	New Relic [Esc]	Escape Studios Development	2.12.2	59,254,308
6	Bacon [Var]	Vargo	11.0.13977...	59,094,009
7	Nginx [Fie]	Fiedler	2.7.6	53,446,238
8	Chef-client [Chek]	Chef Software, Inc.	4.3.0	52,729,508
9	Windows [McL]	McLellan	1.37.0	52,637,365
10	Apt [Cheg]	Chef Software, Inc.	2.7.0	50,034,998
11	Artifact [Win]	Winsor	1.11.3	49,093,711

Table 4.1. Chef marketplace - total downloads ranking [Chel]

Chef enables developers of cookbooks to expose management operations [Chea; Chec; Chee]. Table 4.2 summarizes the characteristics of the available APIs of the analyzed cookbooks and their components. The ✓symbol indicates an explicitly exposed management operation of the type indicated by the column heading. If a cell is empty, there is no such method available. The last column, *implicit configuration*, indicates that either the management operations of the first two columns accept configuration attributes or configuration attributes can be provided for the cookbooks without explicitly exposed operations.

Cookbook	Install / Remove Methods	Start / Stop Methods	Configuration	Implicit Configuration
Mysql	✓	✓		✓
Java	✓		✓	✓
Apache2				✓
Docker	✓	✓	✓	✓
New Relic	✓			✓
Nginx				✓
Chefclient	✓	✓		✓
Windows	✓	✓	✓	✓
Apt	✓			✓
Artifact	✓			✓

Table 4.2. Cookbook API characteristics

4.2 Juju Charms

In this section, the ten most deployed charms of the Juju store [Canl] are analyzed. Table 4.3 depicts the ranking of the charms according their deployment count. The deployment count information are taken from the store page of each particular charm and not from the overview page of the store, because the shown deployment counts are different. The deployment counts of the charm pages match to the order of the ranking on the overview page, thus, they are sound. Contrary, the deployment counts of the overview page do not match to the order of the ranking despite both is shown on the same page [chaa; chab; chac; chad; chae; Canl; Juj; Opea; Opeb; Opec; Oped]. The analyzed sources can be download from the respective pages cited in Table 4.3.

Juju defines hooks for reacting on events. A hook triggers, for example, a specific Python script in case of a distinct event. In Table 4.4, the characteristics of the hooks are summarized. A ✓symbol indicates an explicitly exposed hook of the type indicated by the column heading. If a cell is empty, there is no hook available. Juju defines *unit hooks* for the operations of installing, starting, stopping, reconfiguring, and upgrading the charm [Canc]. However, it is not required to implement them.

No.	Charm	Owner	Revision	Deploy Count
1	juju gui [Juj]	"Juju GUI Charmers" team	33	38442
2	rabbitmq server [chae]	"charmers" team	32	35380
3	mysql [chac]	"charmers" team	25	30390
4	keystone [Opeb]	"OpenStack Charmers" team	26	25498
5	postgresql [chad]	"charmers" team	23	18047
6	apache2 [chaa]	"charmers" team	14	17660
7	haproxy [chab]	"charmers" team	11	17129
8	glance [Opea]	"OpenStack Charmers" team	22	16900
9	nova cloud controller [Opec]	"OpenStack Charmers" team	58	16577
10	openstack dashboard [Oped]	"OpenStack Charmers" team	14	16176

Table 4.3. Juju store - total deployment ranking [Canl]

Charm	Install / Remove Methods	Start / Stop Methods	Configuration
jujugui	✓	✓	✓
rabbitmq	✓	✓	✓
mysql	✓	✓	✓
keystone	✓	✓	✓
postgresql	✓	✓	✓
apache2	✓	✓	✓
haproxy	✓	✓	✓
glance	✓	✓	✓
nova cloud controller	✓	✓	✓
openstack dashboard	✓	✓	✓

Table 4.4. Charm API characteristics

5 Design of the Application Provisioning Modeling Pattern Language

In this chapter, the design of the Application Provisioning Modeling Pattern Language is defined. This encompasses the definition of the domain in which the patterns can be found in Section 5.1, the information format design that is used to document the found information in Section 5.2, the pattern primitives in Section 5.3, and the definition of the patterns and pattern language design in Section 5.4.

5.1 Domain Definition and Constraints

In this section, the domain of the pattern language that is described in this master thesis is documented. Resulting, the characteristic problems are deviated.

5.1.1 Domain Definition

There are multiple approaches for hosting and managing applications. On the enterprise level, manual provisioning and management of applications is very costly and, thus, not appreciated. Thus, technologies like Bluemix, Chef, and Juju, and OpenTOSCA and standardizing approaches like TOSCA are developed. These approaches have in common to utilize small components to compose bigger applications. In the domain of cloud computing, these approaches aim for automated provisioning and management. From infrastructure up to services topologies and also the operation of all, each layer should be covered.

However, there are significant differences between the implemented approaches: Whilst Chef [Cheh] is a script-based product which clearly aims for DevOps, TOSCA [OASb] is a standard for modeling whole business applications on enterprise level, independently from the DevOps approach. All these approaches have in common to compose whole application topologies out of smaller components. But these components are implemented and glued together in different ways. Nevertheless, the common aim is to provision and manage enterprise applications in an automated fashion by using the aforementioned, reusable components.

This master thesis aims for finding commonalities of the different approaches regarding the modeling of components, their wiring, their combined usage, and, foremost, the modeling of

the provisioning of applications that consist of these components. The goal is to develop a pattern language that supports the modeling of new applications by systematically reusing proven knowledge that results from analyzing different and often used technologies.

5.1.2 Domain Characteristic Problems

In this section, common problems are introduced that are tackled by the technologies introduced in Section 2.3.

In the domain of cloud applications, there is a huge heterogeneity according the employed software solutions. There are few standards like TOSCA [OASb] and best practices, for example, published in the Chef and Juju manuals, are not always used and implemented. Therefore, there are not many obvious commonalities on which can be relied on when building a cloud application out of components, for example, obtained from marketplaces.

But these marketplaces and the there offered components are very important. To repeatedly develop and implement the same software solution component for the same or slightly different requirement would be too expensive. The reusability and supposedly easy usage of artifacts is one of the main concerns of the technologies examined in this master thesis.

But to use these offered artifacts does not solve all problems. As a house cannot be built solely by piling bricks and without cement, arbitrarily arraying software components does not create an application. An application has a structure which has to be considered as well as the interplay between the components. Therefore, the next challenge to mention is the wiring of components and the orchestration to applications of arbitrarily size and complexity.

Also, having a plan and the materials of the house does not automatically result in a built house. The question of how the house has to be built remains, for example, if the roof is finished before or after the basement. The examined technologies offer different approaches of how to construct the application. These approaches differ in the methodology, the tooling, and the degree of automation, although, all have a high degree of automation compared to installing and managing the whole application by hand.

But then, the automation opens new issues regarding the control and knowledge of the management and state of the application. The examined technologies are still programs that are solely capable of their functionality, but not of any intelligence compared to a software architect. Another tackled problem is the abstraction and implementation of management logic as well as how the management logic knows the current and desired state of the application and all the steps between.

Therefore, the superior question raised in this master thesis is, how the users of the examined technologies are enabled to model their imagination, so that the aforementioned technologies are able to fulfill the imagination according their functional capabilities to provision cloud applications.

5.2 Information Format Design

The master thesis encompasses and documents knowledge about different technologies, artifacts, scientific works, and concepts. The primarily used format to document this knowledge is the textual description of the findings. Text is not only used in text blocks, but also in combination with common enumerations and tables, which is not defined explicitly. In addition to the textual descriptions, this section defines the used graphical notations of diagrams, icons, and sketches in which the knowledge is documented. The introduced information format design applies for the Section 2.3 and Chapters 4 and 6.

5.2.1 Sequence Diagram

A sequence diagram describes the behavior of and interaction between components. In this master thesis, sequence diagrams follow the rules of the Unified Modeling Language [Objc] which are implemented as shapes in the program Visio 2013 [Mic]. Figure 5.1 shows an example that is additionally described by an enumeration:

1. Server receives a request.
2. Server prepares for management.
3. Server invokes the management.
4. Node returns the results.

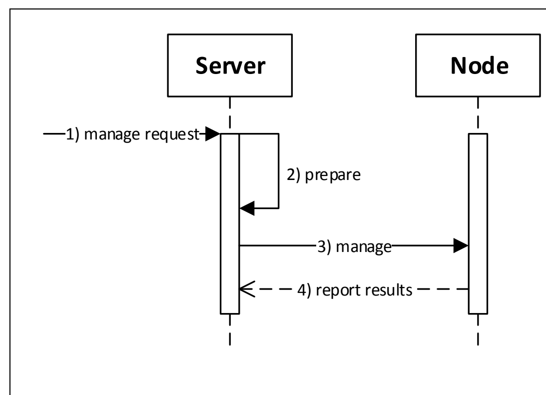


Figure 5.1. Example Sequence Diagram

5.2.2 Graphical Notation of Icons and Sketches

A sequence diagram is very specific in its notation and the type of information it describes. For describing the patterns a less rigorous notation is picked. Subsequent, the notation of the icons and sketches is defined.

Imperative Process Model: The imperative process model is described exemplarily with the BPMN notation of Visio 2013 [Mic]. The circle with the thin border depicts the start event with which the process model instance is started. The inner rectangle with the rounded corners depicts an activity that may, for example, implement a management task. The circle with the

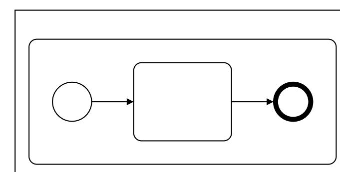


Figure 5.2. Example Imperative Process Model

thick border depicts the end event of the process model. The rectangle with the round corners encompassing the start event, activity, and end event depicts the process model borders.

All other elements used inside a process model, but not defined here, follow the notation of BPMN 2.0 [Obja; Objb]. Figure 5.2 shows an example process model.

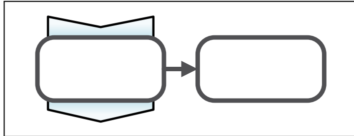


Figure 5.3. Example Topology

Components and Application Topology: Components are depicted as a rectangle with rounded corners and thick, dark borders. If components are connected with a dark arrow, they form a topology.
Components have requirements that are to be satisfied for correct operation. The existence of requirements is depicted with the pointy end of a chevron. Also, components may expose their functionality as capabilities. The graphical notation is the part of a chevron that the pointy end of a chevron can dock. Matching capabilities and requirements can be indicated with the same color. Figure 5.3 shows on the left side a component with exposed capabilities and requirements, on the right side a component without exposed capabilities and requirements, and in the middle an arrow depicting the relation between both components to form a topology.

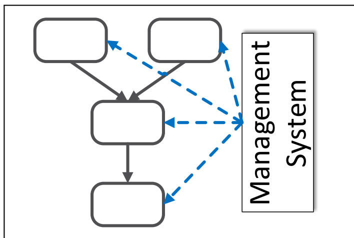


Figure 5.4. Example Management System

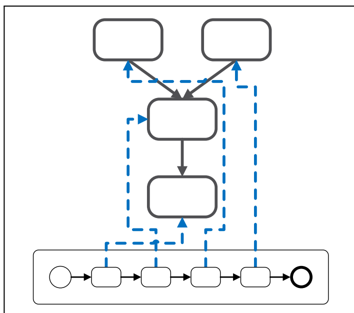


Figure 5.5. Example Management Access

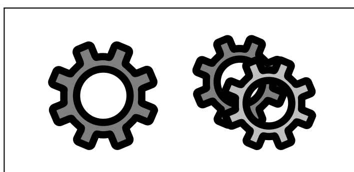


Figure 5.6. Example Technical Statements

Management System and Management Access: A management system is depicted as a rectangle that has a shadow and is named Management System. Declarative management systems access applications to manage, for example, to provision an application modeled by an application topology. This access is depicted with blue, directed arrows with dotted line. In this context, the difference between a modeled application topology and an instantiated application topology instance is neither depicted nor textual differentiated. In Figure 5.4, a management system is depicted that accesses four components of an application topology. In contrast to Figure 5.4, in Figure 5.5, an imperative process model is managing, for example provisioning, an application topology.

Technical Statements: The symbols for technical statements are two overlapping gears or one gear. Figure 5.6 shows both variations. The gear symbols may occur inside activities of imperative process models and components of application topologies. The occurrence location hints for the location of the execution of the technical statements as well as for a very fine-grained, low level implementation of management logic.

Instance Data, Data Access and Data Persistences: For the management of applications, for example, the provisioning, the imperative process model and the management system need instance data. The instance data as transferable object is depicted as a sheet of paper and the instance data persistence is depicted as a database, for example, a CMDB. The access to instance data is visualized with a green, dotted line. The line may be directed to emphasize the flow of instance data or not directed to symbolize general access to instance data. In Figure 5.7, an activity is connected to a database. The access line symbolizes the general access for the instance data d. In this example, the activity may read or write the instance data d.

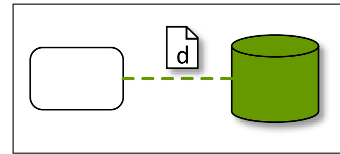


Figure 5.7. Example Data Access

5.3 Pattern Primitives Definition

Domains and subdomains consist of elements with specific names that are well-known to the experts in these fields. Similar elements may occur with different names in various subdomains and domains. Despite the different names, the elements describe the same thing with the same characteristics or purpose. To enable the experts of these various fields to communicate and discuss about common principles, first, a common vocabulary has to be established. Therefore, in this section, a catalog of pattern primitives is introduced. A pattern primitive describes such an element that may occur in various domains and subdomains with different names but the same characteristics or purpose. [FBBL15; Zdu07] The pattern primitives have a name with which they can be referred to and a semantically defined meaning. Table 5.1 lists the pattern primitives used in the Application Provisioning Modeling Pattern Language.

Name	Description
Application	An application satisfies the whole business functionality or a part of it. The application usually consists of multiple software components that may be installed on multiple operating systems and are working together to realize the encompassed business functionality.
Component	A component is one, reusable part of an application. Components can be divided into either application-specific components or general-purpose components.
Dependency	Component's dependencies describe functionalities that the component needs to run but does not implement by its own. So that an application operates correctly, all dependencies inside that application have to be satisfied.
Relation	A relation exists between two components and expresses their interplay, for example, a dependency of on to the other or the wiring between both.

Table 5.1. Definition of the pattern primitives of the Application Provisioning Modeling Pattern Language

Name	Description
Application-specific components	The LAMP stack describes besides common general-purpose components also an application-specific component: The PHP application that is dedicated to the business functionality of the user. Such application-specific components are not as common, heavily used in industry, well-maintained and, thus, easy to use as, for example, a MySQL server.
Container component	A container component is a component which is able to operate another component, typically an application-specific component. The operated component has to be of a specific type.
General-purpose components	The LAMP stack describes mostly general-purpose components: a Linux operating system, an application server, and a MySQL database server. Contrary to the application-specific components, the general-purpose components are not developed for a specific need of, for example, one or a few customers or companies. These components are heavily used in industry, well-maintained and, thus, easy to use.
Application topology	The application topology describes the structure of an application. An application topology is a graph consisting of nodes that represent application-specific and general-purpose components and relations that describe the interplay and wiring between exactly two components.
Application environment	The application environment describes the instance of the application topology and its surroundings, for example, the virtual machines and the installed operating systems. In the application environment, technical statements, scripts, and programs can be executed, for example, invoked by an imperative process model or a management system.
Configuration	The configuration describes the properties of an application as well as how a component has to fit into an application. Also, the configuration may be used for describing the current state of an application or component.
Instance data	All components have specific characteristics in the form of their state and configuration, for example, endpoint information, or credentials. These data are instance data.

Table 5.1. Definition of the pattern primitives of the Application Provisioning Modeling Pattern Language

Name	Description
Imperative provisioning	Imperative provisioning describes a way of how to provision an application. To provision in the imperative way means to model *what* has to be provisioned and *how* the provisioning has to be conducted in detail. The management system simply executes or invokes imperative provisioning logic, for example, a process model that defines the *how*. Contrary to the declarative provisioning, the management system does not derive provisioning logic from the modeled application. However, to provide specific information about the application to the process model, the *what* is modeled as well, but not in the extent of the declarative provisioning. [BBKK+14]
Declarative provisioning	Declarative provisioning describes a way of how to provision an application. To provision in the declarative way means to only model *what* has to be provisioned, without modeling *how* to conduct the provisioning. Everything else to do for provisioning is done by a management system which understands the declarative model. Therefore, management systems derive and conduct the provisioning logic from the declaratively modeled application. [BBKK+14]
Management system	A management system provides the functionality to automatically provision or manage applications, for example, install, configure, or terminate applications or application components. Management systems typically support pure declarative or imperative provisioning, or a hybrid of both.
Management interface	A component may expose distinct operations with which users or management systems can operate the component. These operations define clear semantics and are called a management interface.
Management operation	A management operation is part of a management interface and can be executed to partly or wholly provision a component or change a component instance. To adapt the execution of the management operation to the desired configuration, the management operation may accept parameters. Also, the management operation may report the result of the execution.
Management task	A management task is an abstract task of the management of applications, for example, the provisioning of a component may encompass the tasks *install*, *configure*, and *start*. A management task can be conducted by invoking and executing one or multiple management operations, for example, in the application environment.

Table 5.1. Definition of the pattern primitives of the Application Provisioning Modeling Pattern Language

Name	Description
Process model	An imperative process model is an imperative approach to describe in detail what has to be done. An imperative process model describes typically a sequence of multiple management tasks and, if run, executes or invokes management operations. Examples for process models are workflows, for example, BPEL plans, or scripts, for example, shell scripts.
Technical statement	A technical statement is one distinct command, for example, a command in a shell script. A reoccurring sequence of technical statements may be bundled as a management operation.

Table 5.1. Definition of the pattern primitives of the Application Provisioning Modeling Pattern Language

5.4 Pattern and Pattern Language Design

This section defines the design of the Application Provisioning Modeling Pattern Language and its patterns and pattern candidates that are used in the Chapter 6 to describe the findings. The format of a pattern or pattern candidate is similar to the format of other, aforementioned pattern works, for example, outlined in the Sections 2.1, 2.1.1 to 2.1.3 and 2.2 and adapted iteratively to the needs of the master thesis. The hereby introduced design covers patterns as well as pattern candidates. This is due to the mere fact that pattern candidates may be patterns. Therefore, the patterns and pattern candidates shall not be distinguished in their format. The decision of whether a pattern candidate is a pattern or not is made by applying the *rule of three* that is introduced by Coplien et al. [CA96] and depicted in Section 7.2.

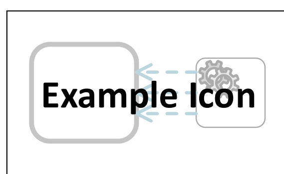


Figure 5.8. Example placeholder for an icon

Problem: The *problem* Paragraph states the difficulty or problem one can be confronted with while modeling, for example, a component to be provisioned. Additional to the problem statement, each pattern specifies an icon which is represented together with the problem statement to ease the visual recognition. Contrary to Figure 5.8, the icons in Chapter 6 have no captions. Also, this enables a visual representation of the pattern.

Context: The *context* paragraph states shortly the situation in which the problem may occur.

Forces: The *forces* Paragraph states the reasoning for the need of a pattern, why it may be difficult to solve, and other possible solutions and their advantages and drawbacks, for example, also not working solutions.

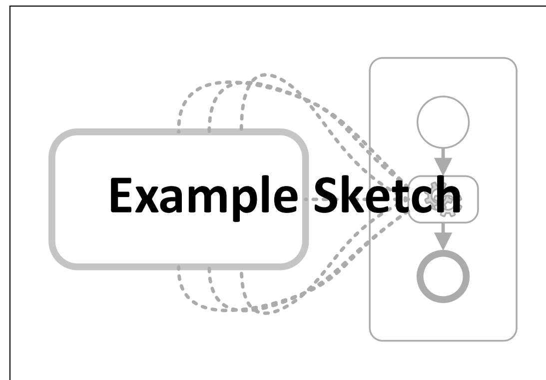


Figure 5.9. Example of a solution sketch

Solution: The *solution* Paragraph provides an abstract description of how to solve the described problem. Additionally, the textual description is enhanced with a solution sketch that visualizes the solution. For example, in computer science a sequence diagram could be chosen to describe the interaction between a process model and a component during the provisioning of a component. Figure 5.9 depicts such a solution sketch.

Results: The *results* Paragraph states what outcome the pattern has. Both advantages and drawbacks have to be documented to enable the reader to be prepared for desired and unwanted consequences.

Next: The *next* Paragraph states the relations of the proposed pattern to other patterns. This section improves the simple collection of patterns to a pattern language by defining in detail the relations between patterns, for example, *cannot be combined because of reasons*, *should be considered*, and so on.

Known Uses: The *known uses* Paragraph states the sources from which the pattern had been extracted or where it has been successfully applied. This substantiates the maturity evaluation of the pattern by providing links to provisioning technologies, artifacts, or scientific works where the pattern is documented or implemented.

6 Application Provisioning Modeling Pattern Language

In this chapter, the Application Provisioning Modeling Pattern Language, its patterns, and pattern candidates are presented that had been found during the analysis of existing technologies, their documentation, and artifacts. These patterns and pattern candidates follow the pattern language design introduced in Chapter 5. Contrary to patterns, the pattern candidates describe principles which are best of breed, but did not succeed yet according their distribution. To distinguish, the *rule of three* [CA96] is applied. If the pattern is realized in three different technologies, the pattern complies with the rule of three. If not, there is need for further research if the pattern candidate is truly a pattern. Nevertheless, the pattern candidates are worthy to be communicated.

The patterns and pattern candidates in this section are describing technical mechanisms of the technologies introduced in Section 2.3 and Chapter 4. The viewpoint for these patterns is a conceptual one and the superior question is how to exploit the conceptual techniques of the analyzed technologies for provisioning an application and how to model or build components for being provisioned in a composed application. The interested users may be developers who want to automate the provisioning of an application and the encompassed components. Therefore, they want to know the mechanisms of a potentially used technology to optimally benefit from the different strengths and possibilities as well as knowing the drawbacks which might be considered. In addition, the patterns guide users in selecting the right technology according their aims. Each pattern and pattern candidate describes in its *next* Paragraph which other patterns should or should not be considered as well as the causation for it.

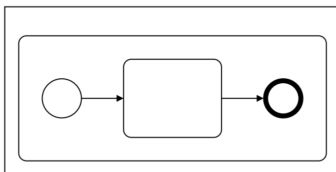
The Sections 6.1 to 6.8 describe the patterns of the Application Provisioning Modeling Pattern Language. The first pattern Imperative Provisioning describes the imperative mechanism with which an application provisioning can be modeled by defining in detail each step of the provisioning to be executed. Contrary, the Declarative Provisioning pattern describes how to provision an application by modeling only the structure and configuration of the application without explicitly modeling the steps of the provisioning to be executed. The Parametrized Imperative Provisioning pattern describes in addition to the Imperative Provisioning pattern how to adapt the imperative provisioning process model according similar scenarios. The Local Management Operation Execution pattern describes how to separate the high level provision logic from the low level technical statements. The pattern Component Lifecycle Interface describes how a component can expose its management behavior regarding its

lifecycle. The pattern Container Component Interface describes how a component can expose its management behavior regarding to host another component of a specific type. The pattern Explicit Dependency Model describes how an application structure can be modeled explicitly. The pattern Implicit Dependency Model describes how an application structure can be modeled incompletely.

In Section 6.9, the External Instance Data Access pattern candidate describes in addition to the Imperative Provisioning pattern how to let the imperative provisioning process model independently access and handle instance data.

The Section 6.10 depicts graphically the connections of the patterns and pattern candidates, which is described in each *next* Paragraph.

6.1 Imperative Provisioning



Problem: How to automate the provisioning of a big, complex application that requires application-specific customization?

Context: An application shall be provisioned automatically.

Forces: Complex business applications typically consist of many different software components that have to be provisioned, deployed, configured, and wired. These components often require individual configurations and cannot always be achieved automatically by analyzing structural models and desired configurations. A structural model specifies the components, their configurations, and their relations to other components, but not how to achieve this. Although, technologies according the Declarative Provisioning pattern exist that are able to provision applications based on structural models, for example, Bluemix, in general they cannot be customized arbitrarily for complex business applications.

Besides common general-purpose components, such as web servers or databases, business applications typically also consist of application-specific components that are not common. Examples are application components that provide a certain functionality which are developed for a specific customer need. Management systems basically not know these types of components and, thus, are not able to provision them automatically in general.

Therefore, a means is required that allows the automatically provisioning of an application by modeling in detail the required provisioning steps.

Solution: Create an executable process model, for example, a workflow or a script that imperatively describes in detail each management task that has to be executed for provisioning the application. This can be combined with various other patterns of other pattern languages, such as the *Parallel Split* workflow pattern [vDTKB03] to enable executing statements in parallel. Also, this approach can be supported by using the Component Lifecycle Interface and Container Component Interface patterns to model the components to be provisioned.

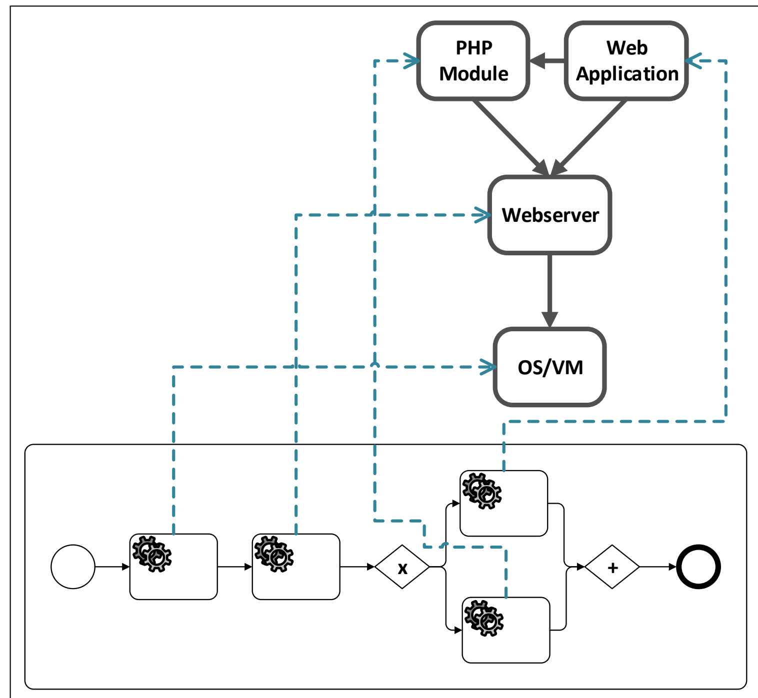


Figure 6.1. Solution Sketch: Imperative Provisioning pattern

Figure 6.1 shows a graphical sketch of the solution. A simple stack consisting of an operating system running in a virtual machine, a web server, its PHP module, and a web application shall be provisioned. The workflow below is provisioning each of these components. Each activity is caring for one component whilst the provisioning of the PHP module and the web application is processed in parallel.

Results: The process model can be executed automatically by a suited runtime, for example, the Apache ODE that is a workflow engine for running BPEL workflows. The process model imperatively prescribes all management operations to be invoked, which enables to customize arbitrarily the provisioning. Therefore, general-purpose components as well as individual, application-specific components can be provisioned arbitrarily by specifying all the management operations in detail, for example, all technical statements needed. Especially, the Imperative Provisioning pattern enables modeling the provisioning of complex applications that cannot be provisioned using a declarative model due to the potentially enormous complexity of individualization of components. Also, parameterizing the process model, as described by the Parametrized Imperative Provisioning pattern in Section 6.3, enables the customization of the same characteristics with different values and, therefore, the reusability for slightly different

provisioning scenarios. If the process model and runtime supports compensation mechanisms [LR98; KMO98], failed provisioning can be automatically compensated.

On the other hand, the Imperative Provisioning may be more complex for the user to model than the Declarative Provisioning pattern. Whilst with the Declarative Provisioning pattern the user only specifies the application's structure and configuration, the Imperative Provisioning pattern requires modeling an additional process model that explicitly describes each step to execute. However, to solve this manual modeling issue, there are approaches for automatically generating provisioning process models based on structural application models, for example, [BBKK+14; BBKL13; BBKL+13; EEKS11; EMEK+06; Mie10]. With the Imperative Provisioning pattern, the user has to specify also in detail how to provision or manage this application additionally.

Next: The Imperative Provisioning pattern can be combined with the Component Lifecycle Interface pattern and the Container Component Interface pattern to ease the modeling of the imperative provision logic. The Declarative Provisioning pattern is using this pattern eventually for provisioning and adapting components. If the instance data is not complex, this pattern can be combined with the Parametrized Imperative Provisioning pattern. Otherwise, the External Instance Data Access pattern candidate should be considered. The Local Management Operation Execution pattern describes how to imperatively execute the technical statements specified by the imperative process model in the application environment and not in the management system itself.

Known Uses: Bluemix exposes its interface via the *cf command line interface*. [IBMc] This command line interface can be used manually to control applications and services or exploited with scripts.

Within Chef, imperative provisioning process models can be found in form of the run-list [Chej]. Also, with the commands *run-list add*, *remove*, and *set* beneath others the run-list can be influenced to customize, if needed [Chef].

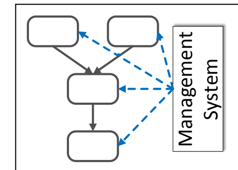
Juju internally uses imperative provisioning logic represented by hooks [Canc]. Also, the user may invoke Actions with parameters to execute provisioning logic [Cang].

TOSCA enables explicitly two ways of application provisioning: Declarative provisioning via the topology model and imperative provisioning by using plans that are executable process models [OASa; OASb].

For the OpenTOSCA runtime environment [BBHK+13; Unia] a generator component has been presented to generate provisioning plans automatically [BBKK+14], thus, easing applying the pattern as the generated plans can be customized individually for certain needs.

Also, there are other academic evidences that provide evidences for this pattern, for example, [BBKL14; BBKL^{+}13; FLRS12].

6.2 Declarative Provisioning



Problem: How to automate the provisioning of a simple application that consists of common components and which requires only little, individual customization?

Context: An application shall be provisioned automatically.

Forces: Applications typically encompass well-known, general-purpose components, for example, a virtual machine running a LTS Ubuntu, a Tomcat application server, and a MySQL server. As these components are heavily used in industry, they are well-maintained and, as a result, easy to use. One could copy prepared configuration files to the right place in the file system, use well-known APIs with which the configuration can be done by invoking with the right parameters, or copy and execute prepared scripts which do not need adaption. To provision applications that consists only of such components by applying the Imperative Provisioning pattern is not efficient and too costly, because the Imperative Provisioning pattern implies to create individual process models.

Therefore, a means is required that allows the automatically provisioning of an application by modeling the application and without defining the required provisioning steps.

Solution: To model the provisioning of such, aforementioned applications, create a detailed model of the application's structure, for example, the required components, their interplay, and their configuration. The modeling can be supported by applying the Explicit Dependency Model pattern to model the dependency relation between two modeled components. Alternatively, if, for example, only the requirements of one component are known, the Implicit Dependency Model pattern can be applied.

To model components that need the execution of specific operations for being provisioned, apply the Component Lifecycle Interface pattern, if the employed management system supports the pattern.

In Figure 6.2, the application is provisioned by a management system. For that the management system analyzes the application topology depicted on the left and, for example, derives the imperative provisioning logic from it.

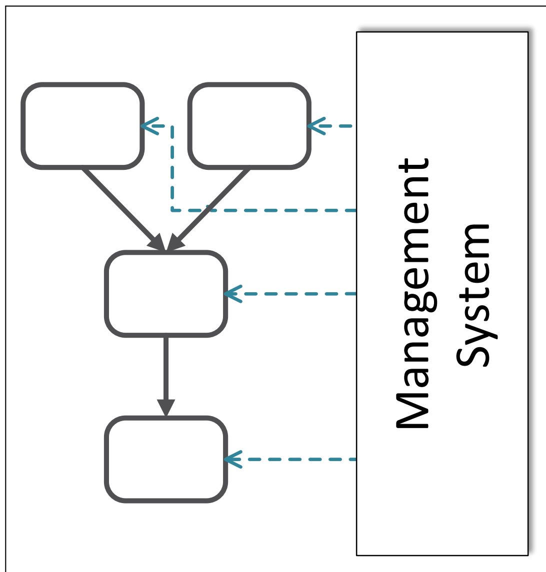


Figure 6.2. Solution Sketch: Declarative Provisioning pattern

Results: The application is modeled primarily with common, well-known components. These components can be prepared to be modeled for processing in the declarative way. The application consists mostly of general-purpose components that are reusable and can be selected from a catalog of available components. In addition, the application structure is modeled with relations that are understood by the employed management system.

The application topology can be processed automatically by a suited management system. Adaption of the characteristics of the components according the desired application topology is supported by the management system. Therefore, the management system can provision the modeled application topology or parts of it in an automated manner. Scenarios like horizontal scalability invoked by a watchdog benefit from the automated management.

Next: As already mentioned, the Declarative Provisioning pattern describes an approach contrary to the Imperative Provisioning pattern. However, the management system implementing the Declarative Provisioning pattern is eventually executing imperative provisioning logic. To support the modeling of the application structure the Explicit Dependency Model pattern and the Implicit Dependency Model pattern can be used. The Component Lifecycle Interface pattern and the Container Component Interface patter support the modeling of components. Because the declarative management system is caring for the provisioning logic the Parametrized Imperative Provisioning, External Instance Data Access, and Local Management Operation Execution may be only of interest for the developers of components, which shall be operated by a declarative management system.

Known Uses: In Bluemix an *App* can be described by a manifest in the *manifest.yml* file. The manifest contains information like the used buildpack, how often the App shall be instantiated or with which services the App shall be bound. Using the manifest the provisioning can be automated. [IBMa] Also, Bluemix supports boilerplates which are application containers. The underlying application model consists of the runtime environment and predefined services for a distinct domain, for example, for hosting Java applications [IBMe]. Boilerplates can be instantiated by the user without interacting with imperative management. The user has only to specify characteristics of the topology and the payment method. For example, there

are multiple options for the database size or monitoring capabilities. [IBMi] Additionally, application topologies can be combined with functionality like automatic scaling [IBMb]. Similarly, runtime environments need no input of how the underlying operating system or the hosting infrastructure are defined [IBMk].

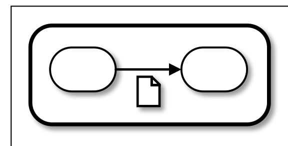
With Chef, the user has not to model imperatively the steps for provisioning. Instead, the user may write just a cookbook importing other cookbooks according the desired functionality. Then, before the chef-client executes the sequences of operations, various information, for example, cookbooks, recipes, files, or attributes, are collected. This information are considered by the chef-client to compile the imperative run-list to execute for provisioning the desired application. [Chej] Then the "chef-client configures the system based on the information that has been collected" [Ched].

Juju supports bundles that are describing services and their configuration. The application topology is represented by the relations linking the services. The user can provision such bundles without modeling imperative provisioning logic. [Cane]

TOSCA enables to model an application topology to enable according declarative approach [OASa; OASb]. Also, there are approaches to combine the declarative provisioning with the imperative provisioning [BBKK^{+}14].

Additional evidences can be found, for example, within the deployment management system Engage that enables to describe application topologies by metadata containing dependencies between components and configuration parameters. With this information Engage can provision the application automatically. [FME12]

6.3 Parametrized Imperative Provisioning



Problem: How to model an imperative provisioning process model in a way that it does not have to care about externally stored application instance data?

Context: The Imperative Provisioning pattern is applied to provision an application and the instance data about the application is handled by an external component, for example, a CMDB.

Forces: To provision a whole application, typically, different kinds of instance data of the involved components are required. For example, for provisioning a virtual machine the installing software requires the endpoint information of the virtual machine for executing the installation. But the instance data are unlikely to be unified for all possible applications which shall be provisioned. Credentials are not always just username and password, endpoints might

reference a certain driver, describing the connection mechanism, instead of only stating an IP. But these instance data can be handled if the application is not arbitrarily big and complex as Bluemix shows with its environment variable [IBMm]. This is important if the instance data are managed by the management system. Otherwise, the process model has to care for it which means an overhead: The management of the instance data are tasks which are not related distinctly to the provisioning.

Also, it is very important to control the instance data. On the one hand, it is necessary to provide the correct instance data to the imperative process model. On the other hand, it is required to update configuration and instance data created by the execution of the process model in the CMDB.

Therefore, a means is required that allows modeling the required parameters of an imperative process model.

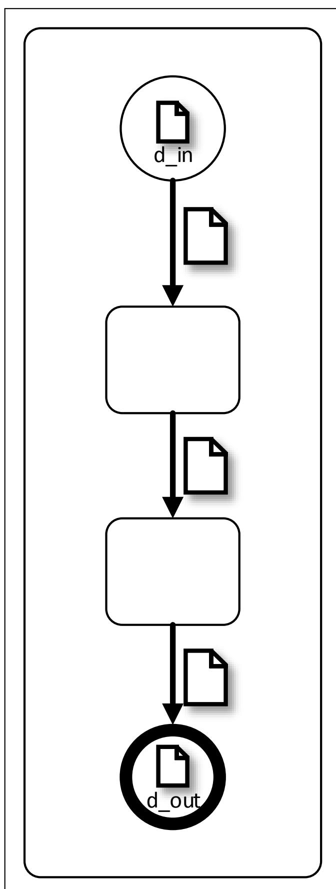


Figure 6.3. Solution Sketch: Parameterized Imperative Provisioning pattern

Solution: Model the imperative process model without hard-coded instance data and without coupling the process model to an external persistence, for example, a CMDB. Therefore, let the management system provide all required input data as parameters with the invocation mechanism and ensure that all further produced instance data are handled within the process model without calling an external storage, such as CMDBs. Also, the instance data can be altered locally. The instance data resulting of the imperative process model are returned to the invoker, who has to care for the persistence and management of the returned instance data.

In Figure 6.3, the start event contains a sheet of paper symbolizing instance data that is passed as parameter when invoking the process model. These instance data are passed, used, and modified accordingly through all the activities until the end activity which contains also a sheet of paper symbolizing the return of the resulting instance data. The accepted sheet of paper d_in alters according to the change of instance data to the sheet of paper d_out that is then returned to the invoker.

Results: The imperative process model exposes the information about which instance data are needed and returned. Therefore, the invoker, for example, a management system or an administrator, is enabled to analyze and handle the required and produced instance data. Also, the resulting instance data are reported back to the management system which is responsible

for the management and persistence of the instance data. Therefore, the imperative process model is not responsible for caring about the persistence of the instance data or other issues like security. Also, the performance of the imperative process model is not reduced by the overhead of retrieving instance data on its own.

On the other hand, the management system has to understand the invocation mechanism of the imperative process model and how to set the parameters for being able to provide the required instance data. This includes issues like security and data complexity. Also, the management system has to manage new instance data which are returned after the execution of the provisioning process model, as these instance data are required for further management activities on the application. Therefore, the management system has to understand completely the instance data for not only effectively store them, but also being able to identify them again later. Thus, this pattern is only applicable to applications with simple instance data consisting of common and well-known input and output parameters that can be handled generically, for example, independently of a certain application.

Next: The Parametrized Imperative Provisioning pattern describes characteristics of the Imperative Provisioning pattern. Therefore, this pattern may be combined with the Component Lifecycle Interface pattern, the Container Component Interface pattern, and the Local Management Operation Execution pattern. On the other hand, this pattern is opposite to the External Instance Data Access pattern candidate and is independent of the Explicit Dependency Model and Implicit Dependency Model patterns. But the Declarative Provisioning pattern may use the Imperative Provisioning pattern and, therefore, the Parametrized Imperative Provisioning pattern eventually, too.

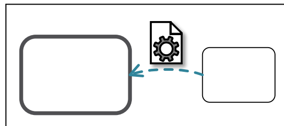
Known Uses: With Bluemix, the user is able to provide parameters to the commands of the *cf command line interface*. Therefore, the invoked command accepts parameters for reacting on, for example, different computing power requirements of the user. The resulting data is printed onto the console. [IBMc]

The chef-client injects instance data into prepared files, for example, configuration files, recipes, or scripts, during the "compile phase" [Chej]. Results produced may be reported to a logging service or sent by a mail. Thus, the used imperative process models are not forced to deal with getting and storing instance data. After the compilation phase, the system is configured by the chef-client running the run-list.

Juju provides a mechanism called Actions which accepts "complex, nested arguments" [Cang] provided by the user. Also, Juju commands accept parameters in form of key-value pairs, for example, to define machine constraints [Cand; Canj].

In TOSCA Definitions, operations and their parameters can be defined for example, in Interfaces of the Boundary Definitions of Service Templates [OASa; OASb]. OpenTOSCA uses these information to enable the invocation of BPEL workflows with the correct input data. [End13]

6.4 Local Management Operation Execution



Problem: How to model the provisioning of a component on a general-purpose infrastructure component, for example, an operating system, which requires multiple operations to be executed?

Context: The provision of an application shall be controlled centrally.

Forces: Usually components have to be adapted to the application. To provision a component on a general-purpose infrastructure component, multiple technical statements have to be executed, for example, copying files. If the component does not implement and expose these technical statements as operations, the operation cannot be invoked by one call. To invoke each single technical statement, for example, by executing each technical statement as shell command using SSH or another mechanism, would imply different drawbacks: The persistent communication over the network consumes resources and introduces a higher latency, thus, is not efficient, and introduces new issues, for example, what happens if the network connection breaks in the middle of a sequence of technical statements to invoke?

Also, the provision logic may not be able to encompass each detail of the management task. A process model may not be designed for implementing and executing technical statements, for example, there is no way to execute single technical statements in the application environment. Also, to model technical statements in the provision logic would clash with a clear separation between the high level provisioning logic implemented in, for example, the process model and low level provisioning logic represented, for example, by a sequence of technical statements and executed in the application environment.

Therefore, a means is required that allows the execution and, if not yet available, prior transfer of multiple operations or sequences of technical statements in a bundled manner in the application environment.

Solution: For all the component's provisioning operations create programs or scripts that are executable on the infrastructure component that shall host the component to be provisioned. These programs or scripts implement all steps needed for accomplishing the provisioning tasks of the component. Transfer these programs or scripts to the infrastructure component to enable the execution of the provisioning operations in a bundled manner.

In Figure 6.4, the Local Management Operation Execution pattern is combined with the Imperative Provisioning pattern. The process model on the right side encompasses a provisioning tasks, represented by the second activity, for a component on the left side. To accomplish the provisioning, the provisioning operation implementations are transferred onto the infrastructure component in the application environment.

Results: Required management operations are deployable in form of, for example, programs or scripts. Therefore, all technical statements are eventually located in the application environment and executed on the infrastructure component that shall host the component to be provisioned. Thus, to transfer and execute the management operations in a bundled manner minimizes the communication over the network and the resulting latency, compared to call each technical statement separately.

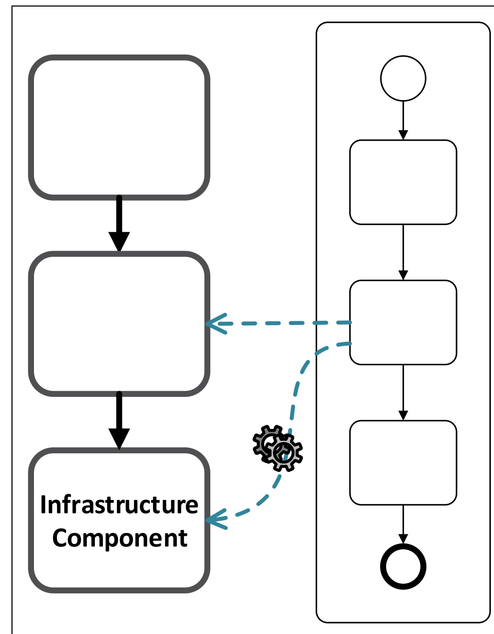


Figure 6.4. Solution Sketch: Local Management Operation Execution pattern

The provisioning logic describes all management tasks to be executed in the correct order to provision the component, for example, by modeling the provisioning logic using the Imperative Provisioning pattern. To achieve the provisioning of components the management operations are transferred and executed on the infrastructure components that shall host the component. Therefore, while modeling the provisioning logic the implementation of management operations has not to be modeled. Thus, modeling the provisioning logic to transfer and execute management operations in a bundled manner is easier than modeling each call of technical statements separately.

Next: The Local Management Operation Execution pattern describes characteristics of the Imperative Provisioning pattern. If a component exposes interfaces described by the Component Lifecycle Interface or Container Component Interface patterns they can be combined with this pattern to, for example, deploy and execute interface operations. The Declarative Provisioning pattern may be using the Local Management Operation Execution pattern in combination with the Imperative Provisioning pattern as a management system derives imperative provisioning logic, for example, from the application topology. Just in combination with the Declarative Provisioning pattern the Implicit Dependency Model pattern and the Explicit Dependency Model pattern should be considered. For retrieving instance data this pattern can be combined with the Parametrized Imperative Provisioning and External Instance Data Access.

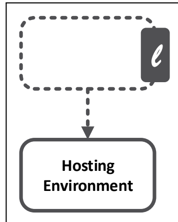
Known Uses: Applications in Bluemix are hosted in runtimes and are to be deployed, installed, and configured there. The *cf command line interface* enables the user to run scripts that are deployed with the application, for example, by executing the command *cf push AppName -c "bash ./configuration_script.sh"*. [IBMc; IBMm]

To provision an application with Chef the run-list is executed after the chef-client transferred every needed resource on the bootstrapped infrastructure component. The run-list is executed on the aforementioned infrastructure components. [Chej]

With Juju all charms are deployed onto the infrastructure component before the provisioning is executed. Direct actions on units are defined by hooks and are invoked on events on the infrastructure component. [Canc; Canf]

With TOSCA this pattern is explicitly compatible: "Deployment artifacts are the installables of the components" [OASa] that are deployed into the application environment. "The global management behaviour covering the complete lifecycle of a cloud application is defined by means of plans." [OASa]. OpenTOSCA is implementing this pattern by first deploying the application resources and then instantiating the application by running a build plan. [BBHK+13]

6.5 Component Lifecycle Interface



Problem: How to model a component to enable its automated operation on a general-purpose infrastructure or platform, for example, a virtual machine, which has not explicitly been built to run solely this component type?

Context: A component shall be hosted on another component.

Forces: To operate a component on a general infrastructure or platform component typically several management tasks are required to be performed: The component has to be installed, configured, started, and so on. If the semantics of these management operations are not defined clearly, invoking them in the correct order by a management system is not possible in general.

In addition, the modeled component maybe should be hosted on different infrastructure systems or platforms. To enable this portability, it is required to implement the management operations for each supported host. For example, to enable running the modeled component both on Linux and Windows implies distinct sequences of technical statements.

Therefore, a means is required that allows modeling a component to expose its capabilities to be managed regarding its lifecycle.

Solution: Let the component expose an interface that provides all management operations needed regarding the component's lifecycle. The exposed interface and its operations are clearly defined regarding their functionality. The functionality shall enable to accomplish the following management task types: installation, configuration, start, stop, and deletion. Further, define arbitrarily the required input and output parameters of the management operations as needed. The component needs to implement management operations capable of the aforementioned management task types for all the underlying environments you possibly want to employ. For example, employ Bash scripts for operating the component on Linux systems.

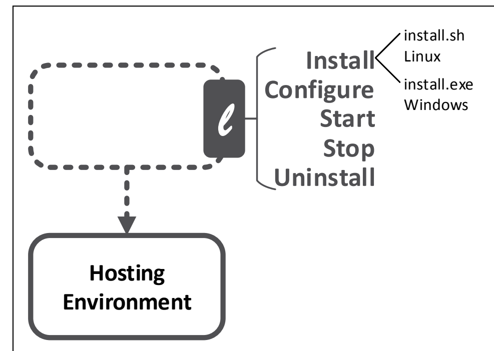


Figure 6.5. Solution Sketch: Component Lifecycle Interface pattern

In Figure 6.5, the lower component shall host the upper component with the dotted line. The upper component exposes a lifecycle interface with five operations according the five proposed management task types. The *Install* management operation has two different implementations: one for Linux and one for Windows.

Results: External management systems or process models are enabled to invoke the exposed management operations of the modeled component, as the semantics of the management task types are clearly defined. Therefore, if an external management system is able to understand the modeled management operations, it can invoke them in the correct order for accomplishing the desired management tasks. Thus, the management systems are not only enabled to provision a component automatically, but also to operate the component automatically.

On the other hand, the invocation mechanism may still differ between different implementations of this pattern. For example, there are differences between the handling of scripts and Java implementations. This is up to the invoking management system.

Furthermore, no knowledge is needed about the management operation implementation regarding the underlying component. Only the right management operation implementations has to be selected and invoked to accomplish the management task, for example, by exploiting the *Strategy* pattern [GHJV94] to automatically choose the right implementation regarding the underlying infrastructure component. The different implementations of how to accomplish the management task enables the easy operation of the component on various underlying components.

Next: The Component Lifecycle Interface pattern can be used by process models that are described by the Imperative Provisioning pattern. Therefore, the Parametrized Imperative Provisioning pattern and External Instance Data Access pattern candidate should be considered

to enable the imperative provision logic to provide needed parameters. Also, the Component Lifecycle Interface pattern is benefiting the Local Management Operation Execution pattern by describing local management operations. Furthermore, the Declarative Provisioning pattern is enabled, if the management system is able to understand the modeled management operations. Although it is not necessary, this pattern can be combined with the Container Component Interface pattern. This pattern is independent of the Explicit Dependency Model pattern and Implicit Dependency Model pattern.

Known Uses: Bluemix services can be managed with the *cf command line interface* that supports the operations of the Component Lifecycle Interface. For example, the command *cf push AppName* creates an application, *cf delete AppName* deletes the application instance, *cf start AppName* and *cf stop AppName* starts and stops application instance, and with *cf set-env AppName key value* the input data can be set in form of an environment variable to configure the application instance. The information can be found in the documentation [IBMc] and in the help-dialog of the *cf command line interface* itself.

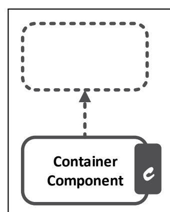
There are four Chef cookbooks among the analyzed ones which expose management operations that implement functionality for install, start, stop, and delete [Chek; Cheo; Fla; McL]. Also, there are three cookbooks among the analyzed ones which expose configuration operations [Agi; Fla; McL].

Juju charms implement so-called "unit-hooks" [Canc] which comply the functionality of installation, changed configuration, starting and stopping, and upgrading the component with a new charm version. All analyzed charms are exposing these hooks [chaa; chab; chac; chad; chae; Juj; Opea; Opeb; Opec; Oped].

OASIS is advising to expose a lifecycle related interface when using TOSCA [OASb]. Additionally, with OpenTOSCA it is possible to implement the *Strategy* pattern [GHJV94] to enable easy management on multiple underlying components or environments [HLNW14].

The management framework for cloud applications c-Eclipse picks up and implements the notion of the lifecycle interface proposed by TOSCA [SLTP+14].

6.6 Container Component Interface



Problem: How to model a container component for automated operation of components that has been built to be run by the container component?

Context: A component shall host another component.

Forces: To operate a component on a container component takes several management tasks to be accomplished: First, the container component needs all resources of the component. Before the component can be launched, it maybe needs to be prepared for operation. Also, the component maybe needs to be stopped and restarted or even decommissioned. These management tasks can also be bundled in management operations with clear semantics. Contrary, if the semantics are not defined clearly, the correct invocation is not possible in an automated way.

Modeling a container component to enable the operation of other components requires a definition of the operation capabilities of the container component. This encompasses the type or types of all possible components that can be operated.

Therefore, a means is required that allows modeling a component to expose its capabilities to operate other components of specific types.

Solution: Let the container component expose an interface that enables the operation of other components of one or multiple specific types. The exposed functionality shall enable to accomplish the following management task types: Deploy and undeploy a component, start and stop a deployed component. The container component needs to implement each of the management operations for all the types of components which shall be operated. For the deployment management task type, define all mechanisms with which the container component can be supplied with the component files to operate.

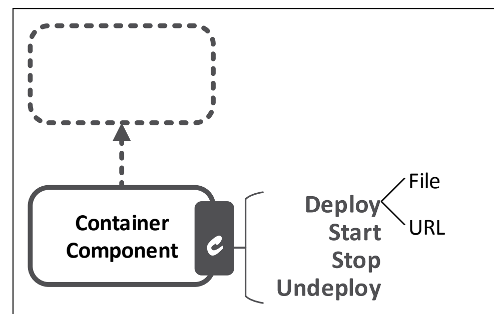


Figure 6.6. Solution Sketch: Container Component Interface pattern

In Figure 6.6, the upper component with the dotted line shall be hosted on the lower container component. The lower component exposes a container interface with management operations regarding the four aforementioned management task types. In the depicted example, the *deploy* provisioning operation has two different implementations: One for accepting a file and one for accepting an URL pointing to the file that contains the component resources to host.

Results: External management systems are enabled to invoke the exposed management operations of the modeled container component. The semantics of the four management task types are clearly defined. Therefore, if an external management system understands the semantics the management task types, they are usable correctly to accomplish the desired management task. Thus, the management systems are enabled to operate a component hosted by the container component automatically.

On the other hand, the invocation mechanism may still differ between different implementations of this pattern. Also, the transportation protocol to provide all needed component resources to the container component may differ. All this is up to the invoking management system.

Next: The Container Component Interface pattern can be used by an imperative process model described by the Imperative Provisioning pattern. Therefore, the Parametrized Imperative Provisioning pattern and External Instance Data Access pattern candidate should be considered to enable the imperative process model to provide needed parameters. Also, this pattern is benefiting the Local Management Operation Execution pattern by describing local management operations. Otherwise, to use the Container Component Interface pattern declaratively the Declarative Provisioning pattern should be considered. Although it is not necessary, this pattern can be combined with the Component Lifecycle Interface pattern. This pattern is independent of the Explicit Dependency Model pattern and Implicit Dependency Model pattern.

Known Uses: There are several general-purpose container components, for example, application servers. The pattern is supported, for example, by two application servers: The Apache Tomcat supports a HTTP interface with the operations *deploy*, *start*, *stop*, and *undeploy* [Apab]. Also, the application server JBoss Application Server 7.0 supports with its command line interface the operations *deploy* and *undeploy* [BK].

Other container components that implement the Container Component Interface pattern are workflow engines that are able to execute workflows. Activiti is such a workflow engine that implements this pattern within its classes *RepositoryService* and *RuntimeService*. The two operations *createDeployment* and *deleteDeployment* of the class *RepositoryService* match the *deploy* and *undeploy* management task types [Alfb]. The two operations *startProcessInstanceById* and *deleteProcessInstance* of the class *RuntimeService* match the management task types *start* and *stop* [Alfc]. Another example is the Apache ODE [Apaa] which is used in the *uniform BPEL management layer* to process BPEL workflows. The proposed *uniform BPEL management layer* exposes operations for deploying and undeploying BPEL plans in the *UniformProcessDeployment*. As the plan is deployed, it is also started. The implemented prototype is capable of using the versions 1.3.5 and 1.3.6. of the Apache ODE. [HLWL14]

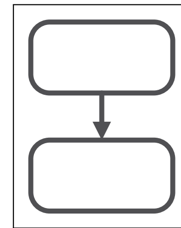
Also, offerings of platform-as-a-service support this pattern: the *cf command line interface* of Bluemix [IBMc] as well as Docker with its operations *pull*, *run*, *start*, *stop*, *rm*, and *rmi* [Doca; Docb; Docc; Docd; Doce; Docf].

There are two cookbooks among the analyzed cookbooks that contain container components and could support the pattern: The apache2 cookbook enables with its definitions *apache_site* and *web_app* to configure a new virtual host and install a web application [vZoe]. The Docker cookbook supports with its container and image provider to manage containers and the hosted

images with the operations *docker_image pull* and *remove* as well as *docker_container run* and *kill* [Fla].

Also there are ambitions of standardizing container interfaces in science with the COAPS API. [HKOH13; MSSM^{+}]

6.7 Explicit Dependency Model



Problem: How to model dependencies of a component that have to be fulfilled specifically by other components?

Context: A component to provision has dependencies to other components.

Forces: In software engineering, it is well-known that the monolithic approach does not work well. Component based software development is known since years [Sam97]. Therefore, the monolithic approach is unlikely to work for the provisioning of applications. Modularized applications cannot be provisioned as a whole without considering its several components. Thus, the components have to be provisioned separately and wired, according the application structure. To enable the automated provisioning of the components and their correct wiring, the application structure has to be known as well as be interpretable by the automated provisioning mechanism. But to decompose the application by using only components would result in a loss of the information about the structure of the application and the interplay between the components.

Therefore, a means is required that allows modeling the interplay between components, for example, the *hosted on* relation between an application server and an operating system. While modeling the components in a loosely coupled manner may be enough in some cases, often it is required to model explicitly how these dependencies of a component have to be fulfilled specifically.

Solution: Model explicitly all dependencies between components of the application as follows: For each component model's requirement model a dependency relation to the component that satisfies the requirement. Type the dependency relation according the requirement to define the semantics.

In Figure 6.7, the application topology is modeled with two elements: the components and arrows. The arrows connect one component having a need with another component satisfying the need.

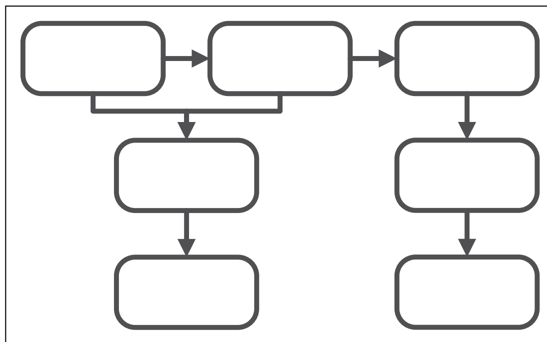


Figure 6.7. Solution Sketch: Explicit Dependency Model pattern

Results: By modeling explicitly the dependency relations of one component to others a management system is able to interpret the application structure. By defining the semantic of the dependency relations an automated provisioning mechanism is enabled to identify the order in which the components have to be provisioned. The explicitly modeling of dependency relations enables the declarative provisioning described by the Declarative Provisioning pattern. Also, in a visual notion additional ways of modeling are enabled, for example, by drawing an arrow to specify the dependency relation.

A management system that is capable of interpreting explicit dependency relations may identify parallel processable provisioning tasks. This boosts the speed of provisioning a new application. Also, the Imperative Provisioning pattern benefits with additional understanding of the modeler, even if the pattern does not need an explicit application topology. With modeling the application structure the author of the imperative provisioning logic can identify parallel processable tasks and improve the imperative provisioning logic accordingly [BBKK+14].

Next: The Explicit Dependency Model pattern serves to model explicitly an application topology, contrary to the Implicit Dependency Model pattern. But both can be combined, if the management system supports this. The Explicit Dependency Model pattern is needed for considering the Declarative Provisioning pattern. In general the Explicit Dependency Model pattern is helpful for the user to understand the application topology, but not necessary to apply the other patterns.

Known Uses: With Bluemix the dependencies between runtimes and services can be fulfilled by so-called bindings. Instead of binding the service instances according their dependency relations manually, the user can model relations explicitly in the manifest file by stating which other services have to be bound during the instantiation [IBMa].

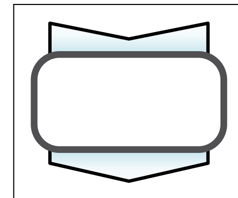
In the file *metadata.rb* in Chef cookbooks, the modeler can state explicitly which other cookbook satisfies dependencies of the own [Chen]. This can be visualized using the tool knife [Chem].

Juju enables the user to model the relation between components, for example, while defining a bundle. In the definition of the bundle, the user not only models encompassed services, but also how these services are to be connected to enable their interplay. [Cane] Another dependency relation called *implicit relation* enables services to observe other service's lifecycle events [Cani].

TOSCA defines so-called Relationship Templates with which components called Node Templates can be connected [OASa]. With the graph formed by these dependency relations imperative provisioning plans can be generated out of the declarative TOSCA models [BBKK+14].

Within the deployment management system Engage "intercomponent dependencies" [FME12] are subdivided into three types: The inside dependency describes a hosting relation, the environment dependency describes a requirement relation fulfilled on the same node of the graph, and the peer dependency describes a requirement dependency on any node of the modeled application.

6.8 Implicit Dependency Model



Problem: How to model the dependencies of a component without modeling explicitly how to fulfill the dependency?

Context: A component to provision has dependencies to other components or the target environment.

Forces: It is not always desired or possible to explicitly model how to satisfy a dependency. Users may not want to limit the relation to one explicit other component. Also, users may not want to or are not able to model the whole application topology from the application-specific layer down to the infrastructure. This would mean a modeling overhead because all characteristics on the infrastructure and platform layer have to be modeled, too. Therefore, the users may want to state solely which dependencies of their component the rest of the application topology has to satisfy.

Therefore, a means is required that allows modeling the interplay between components, for example, the dependency of an application server to be *hosted on* an operating system, without explicitly model how to satisfy the dependency.

Solution: For all components having dependencies that are not to be explicitly modeled, for example, by whom to be satisfied, model the requirements and capabilities of the component and attach it to the component model. Employ a management system which is able to automatically satisfy requirements that are not modeled how to be satisfied.

In Figure 6.8, the user models the *PHP App*. This component exposes its requirement which is an application server capable of running a PHP application. The management system cares for realizing the rest of the application topology. The user may know that an application server may need a *PHP module* to run the *PHP App*, but does not model it. Also, the user does not

know which application server the management system is using and therefore is not able to model the components depicted with a question mark in the lower part of the topology.

Results: It is possible to model incomplete application topologies containing only the components which the user can or want to define explicitly. The components expose their requirements as well as their capabilities. The relations between components are modeled in a loosely coupled manner instead of explicitly.

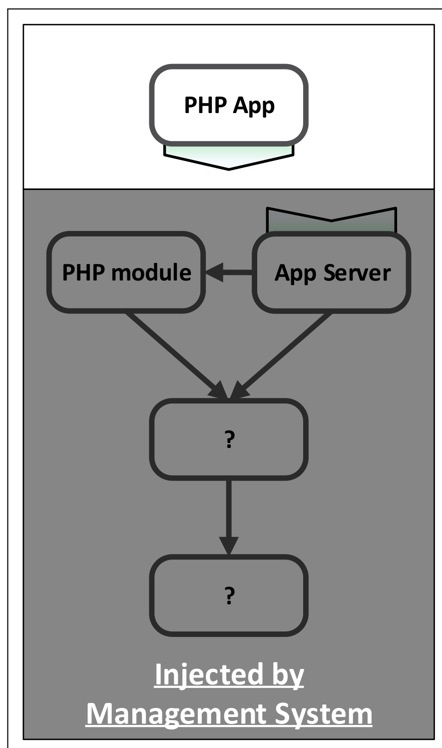


Figure 6.8. Solution Sketch: Implicit Dependency Model pattern

Therefore, there is no way to model components which satisfy requirements of not modeled components. The satisfaction of modeled requirements has to be ensured by the management system. This enables to hide components which are not managed by the user, for example, the lower parts of the application layer.

But on the other hand, it enforces the user to use a management system which is capable of automatically handling incomplete application topologies.

To use such a management system reduces the modeling overhead. From the management system perspective, it enables to offer, for example, infrastructure or platform as a service. The user only books according the own requirements and the rest is automatically done as a managed service.

Next: The Implicit Dependency Model pattern serves to model an application topology in a loosely coupled manner, contrary to the Explicit Dependency Model pattern. But both can be combined, if the management system supports this. The Implicit Dependency Model pattern is needed for considering the Declarative Provisioning pattern. In general the Implicit Dependency Model pattern is helpful for the user to understand the application topology, but not necessary to apply the other patterns.

Known Uses: Using the manifest file for modeling the application to push and host with Bluemix, requirements to the underlying application can be defined without modeling the whole application. For example, within the manifest file the disk quota or available memory can be defined without modeling the operating system or virtual machine. [IBMf]

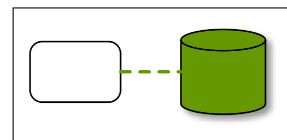
In the file metadata.rb in Chef, the user can define the requirements and capabilities of the cookbook with the keywords *depends* and *provides*. Additionally, with the keyword *depends*

version constraints can be defined to enable Chef to choose one cookbook with a matching version of different available cookbooks. No explicit dependency has to be defined. While the deployment process, the chef-client cares for retrieving cookbooks from the Chef server that are satisfying the dependencies. [Chen] Two cookbooks state dependencies to the hosting platform like the operating system or the infrastructure component like a virtual machine [Chek; Win] and the other eight do not [Agi; Cheg; Cheo; Esc; Fie; Fla; McL; vZoe].

Juju uses a similar concept: Within the file metadata.yaml, the requirements can be defined in the sections peers and requires as well as the capabilities in the section provides. To specify a dependency the relation name and an interface are named, but not the specific component implementing the interface [Canh]. For example, none of the analyzed artifacts state a requirement to the specific hosting platform like the operating system. They only state functional requirements [chaa; chab; chac; chad; chae; Juj; Opea; Opeb; Opec; Oped]. But this is not to confuse with Implicit Relations which enables the exchange of "lifecycle-oriented events and data" [Cani].

The definition of capabilities and requirements in TOSCA enables such mechanisms. [OASa; OASb] But up to now OpenTOSCA is not supporting such a feature. However, the TOSCA modeling tool winery [Unic] supports completing topology models encompassing capability and requirement definitions to specify dependencies between node types [HBBL14].

6.9 External Instance Data Access



Problem: How to model the imperative provisioning logic to enable it to independently access and handle the instance data of big and complex applications at an external persistence?

Context: The Imperative Provisioning pattern is applied to provision an application and the instance data about the application is handled by an external component, for example, a CMDB.

Forces: It has to be ensured that the imperative process model has all the needed instance data. Also, the instance data resulting of, for example, provisioning tasks may be needed later by others. The management system is not able to ensure the management of instance data in general. Also, the management system is not able to react or be adapted to all possible changes of instance data in general.

To customize a component towards the application different instance data are usually needed. These instance data are, for example, IP-address and credentials of the operating system on which the component has to be provisioned, credentials and endpoints to where store customer data, and so on. But the instance data are unlikely to be unified for all possible applications

that shall be managed. For example, credentials are not always just username and password, endpoints might reference a certain driver describing the connection mechanism instead of only stating an IP.

The unification of instance data is not possible in general for arbitrarily big and complex applications. Therefore, the management system may not be able to identify instance data to provide them to a parameterized process model, for example, described by the Parametrized Imperative Provisioning pattern.

Therefore, a means is required that allows an imperative provisioning logic to independently access and handle instance data at an external persistence.

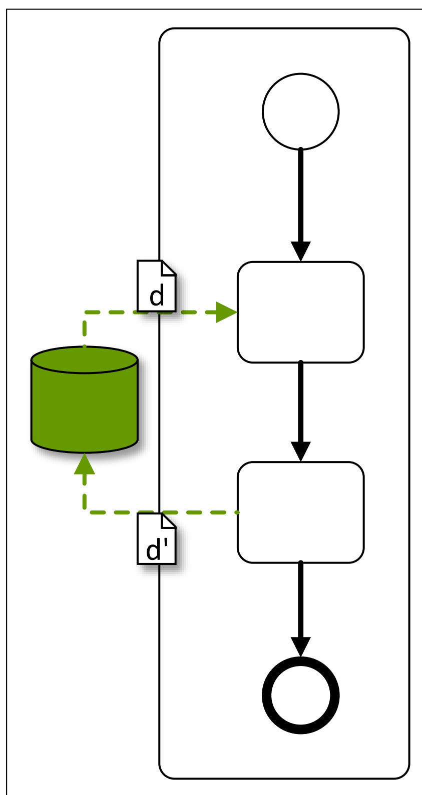


Figure 6.9. Solution Sketch: External Instance Data Access pattern candidate

Solution: Implement or use an external instance data persistence with which instance data can be accessed and managed. Model the imperative process model to access, retrieve, create, and update the instance data persistence.

In Figure 6.9, neither the start nor the end event is parameterized as it would be with the Parametrized Imperative Provisioning pattern. With the External Instance Data Access pattern, the activities are enabled to access an external persistence to get or store instance data that symbolized by the sheets of paper. The first activity retrieves instance data d from the instance data persistence and the second activity stores the results d'.

Results: The provisioning logic, modeled as a process model, is enabled to manage the instance data on its own. For that, the imperative process model accesses the instance data persistence directly to retrieve and update instance data. Therefore, the management system has not to be able to provide and, thus, identify the right instance data for a distinct process model instance. The management tasks know only their needed instance data and not all the instance data of the provisioning logic.

The complexity for the management system is reduced because it has not to manage and, therefore, not to understand the instance data. The management system is not the limiting factor because of its inability of understanding arbitrarily big and complex applications.

On the other hand, the instance data has to be confidential and not to be changeable by the wrong people. For all security issues has to be cared for in the access mechanism of the instance data persistence and the instance data persistence itself.

Next: The External Instance Data Access pattern candidate describes characteristics of the Imperative Provisioning pattern. Therefore, this pattern may be combined with the Component Lifecycle Interface pattern and Container Component Interface pattern to invoke predefined management operations that are described by the interfaces and with the Local Management Operation Execution pattern, if the management operations are not available in the application environment yet. On the other hand, this pattern is opposite to the Parametrized Imperative Provisioning pattern. Also, this pattern is independent of the Explicit Dependency Model pattern and Implicit Dependency Model pattern. But, this pattern may be used by the Declarative Provisioning pattern as a declarative management system eventually has to execute or invoke imperative provisioning logic.

Known Uses: In Bluemix instance data, for example, endpoint information and credentials needed to connect to other Services, are accessible in a *VCAP_SERVICES* called environment variable [IBMa; IBMm].

OpenTOSCA persists instance data for provisioned applications. These instance data are accessible via a dedicated instance data API. [Eis13]

6.10 Overview of the Pattern Language

In Figure 6.10, the pattern language is visualized. Each pattern is connected with the other patterns according the *next* Paragraph in the respective pattern sections in the Chapter 6. To improve the readability the connecting arrows are reduced to the minimum and arrows describing transitive connections are omitted. This means that only connections of the type: "If you use A, consider B" are depicted, but not "If you use A, consider B, if you also use C". For example, the Parametrized Imperative Provisioning pattern describes the characteristics of the Imperative Provisioning pattern. Therefore, both are connected. The Imperative Provisioning pattern is used by the Declarative Provisioning pattern in general, but the Declarative Provisioning pattern is not using the Parametrized Imperative Provisioning pattern explicitly. Therefore, the Declarative Provisioning pattern points to the Imperative Provisioning pattern, but not to the Parametrized Imperative Provisioning pattern.

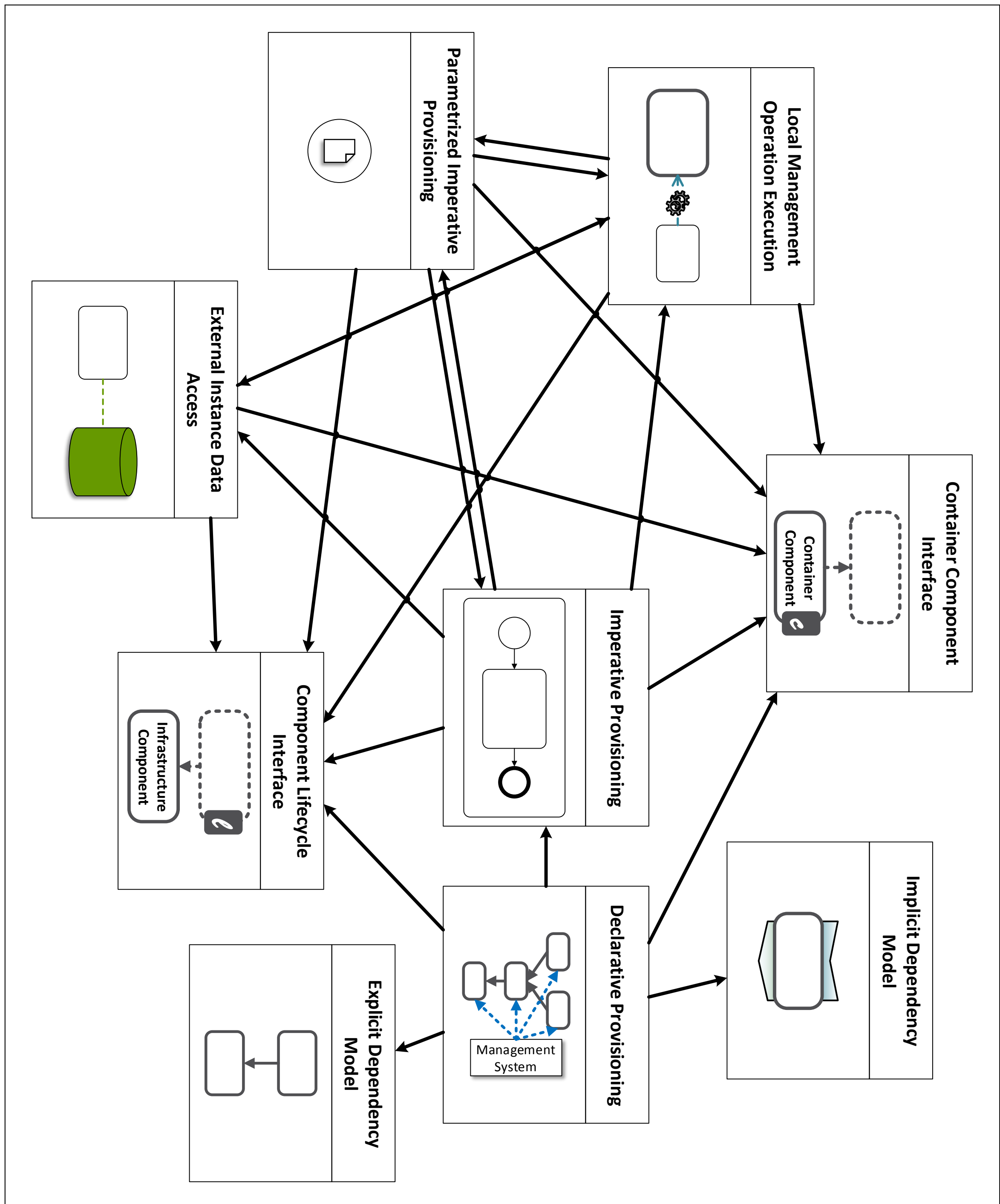


Figure 6.10. Overview of the Pattern Language

7 Discussion

This Chapter discusses the validity of the proposed results. In Section 7.1, an overview of the *known uses* of the patterns introduced in Chapter 6 is depicted. Subsequent, in Section 7.2, the maturity evaluation is presented. In Section 7.3, the potential threats to validity and in Section 7.4, the limits of theses master thesis are mentioned.

7.1 Overview of the Known Uses of the Patterns and Pattern Candidates

In this section, an overview is depicted which relates the known uses of the proposed patterns to the analyzed technologies.

In Table 7.1, each pattern is correlated to the four analyzed technologies and standard as well as to additional evidences summed up as *others*. These others are either other technologies or scientific works. A ✓symbol indicates that the pattern is supported in the technology or suggested in the scientific work. If a cell is left empty, the pattern is not supported. The detailed explanations of the known uses are in the *known uses* paragraphs of the patterns in the Chapter 6.

7.2 Maturity Evaluation of the Patterns and Pattern Candidates

In Table 7.1, an overview of the patterns, pattern candidates, and their evidences is depicted. Subsequent, in Table 7.2, the derived maturity evaluation is presented. The first column states the pattern or pattern candidate name. The second column presents the coverage of the pattern in the considered technologies. The third column states, if there are additional evidences, for example, in scientific works or prototypes. The fourth and fifth column state, if the *rule of three* is fulfilled and, therefore, the identified principle conforms to a pattern or a pattern candidate.

Name	Bluemix	Chef	Juju	TOSCA OpenTOSCA	Other Technologies	Scientific Work
Component Lifecycle Interface	✓	✓	✓	✓		✓
Container Component Interface	✓	✓			✓	✓
Declarative Provisioning	✓	✓	✓	✓		✓
Explicit Dependency Model	✓	✓	✓	✓		✓
External Instance Data Access	✓			✓		
Imperative Provisioning	✓	✓	✓	✓		✓
Implicit Dependency Model	✓	✓	✓	✓		
Local Management Operation Execution	✓	✓	✓	✓		✓
Parametrized Imperative Provisioning	✓	✓	✓	✓		

Table 7.1. Overview of the Known Uses of the patterns and pattern candidates

7.3 Threats to Validity

The in this document proposed pattern language was conducted as a master thesis. The master thesis did not encompass expert interviews as it is common in the pattern community. The used process [FBBL15] explicitly states that implemented technologies and their artifacts and documentations are valid as information source for working out patterns. Although this approach in combination with the rule of three [CA96] provides a statistical basis that quantitatively proves the repeatedly occurrence of the found principles and, therefore, the existence of the found patterns, in the pattern community it is common to discuss patterns in workshops with other experts. Whilst the proposed pattern language was presented to and discussed with members of the Institute of Architecture of Application Systems, a dedicated and structured workshop with experts from different companies or institutes was not conducted yet. Therefore, these workshops will be part of future research to also include further expert knowledge in this pattern language.

Name	Coverage in Considered Technologies	Additional Evidences	Rule of Three	Rating
Component Lifecycle Interface	80%	✓	✓	Pattern
Container Component Interface	60%	✓	✓	Pattern
Declarative Provisioning	80%	✓	✓	Pattern
Explicit Dependency Model	80%	✓	✓	Pattern
External Instance Data Access	40%			Pattern Candidate
Imperative Provisioning	80%	✓	✓	Pattern
Implicit Dependency Model	80%		✓	Pattern
Local Management Operation Execution	80%	✓	✓	Pattern
Parametrized Imperative Provisioning	80%		✓	Pattern

Table 7.2. Maturity evaluation of the patterns and pattern candidates

7.4 Limits of the Thesis

The proposed pattern language does not claim completeness. It is certain that there are other patterns that should be considered for the proposed pattern language. Also, the proposed pattern candidates are likely to fulfill the *rule of three*, if the scope of considered technologies is extended. Similarly, the pattern language can be extended, if the limit of provisioning tasks is extended to management tasks regarding the management and operation of cloud application components during their whole lifecycle.

8 Literature and Other Resources

8.1 Literature

[Ale79] Christopher Alexander. *The timeless way of building*. New York: Oxford University Press, 1979.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: Towns, buildings, construction (Center for environmental structure series)*. Oxford University Press, 1977.

[BBHK+13] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. "OpenTOSCA - A Runtime for TOSCA-based Cloud Applications." In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13)*. Springer Berlin Heidelberg, 2013.

[BBKK+14] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA." In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2014.

[BBKL13] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. "Pattern-based Runtime Management of Composite Cloud Applications." In: *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*. SciTePress, 2013.

[BBKL14] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. "Automating Cloud Application Management Using Management Idioms." In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, 2014.

[BBKL+13] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and Johannes Wettinger. "Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies." In: *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS 2013)*. Springer Berlin Heidelberg, 2013.

[BHS07] Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. John Wiley & Sons, Ltd., 2007.

[BMRS+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A system of patterns: Pattern-oriented software architecture*. Wiley West Sussex, England, 1996.

[CA96] James O Coplien and A Word On Alexander. *Software patterns*. Citeseer, 1996.

[EEKS11] Tamar Eilam, Michael Elder, Alexander V Konstantinou, and Ed Snible. “Pattern-based Composite Application Deployment.” In: *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*. IEEE, May 2011.

[Eis13] Marcus Eisele. *Verwaltung von Instanzdaten eines TOSCA-Cloud-Services*. 2013.

[EMEK+06] Kaoutar El Maghraoui, Alok Meghranjani, Tamar Eilam, Michael Kalantar, and Alexander V. Konstantinou. “Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools.” In: *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., 2006.

[End13] Christian Endres. *Management von Cloud Applikationen in OpenTOSCA*. 2013.

[FBBL15] Christoph Fehling, Johanna Barzen, Uwe Breitenbücher, and Frank Leymann. “A Process for Pattern Identification, Authoring, and Application.” In: *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2015.

[FLRS+14] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Science & Business Media, 2014.

[FLRS12] Christoph Fehling, Frank Leymann, Jochen Rütschlin, and David Schumm. “Pattern-Based Development and Management of Cloud Applications.” In: *Future Internet Special Issue “Recent Advances in Web Services” (pdf)* (2012).

[FME12] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. “Engage: A Deployment Management System.” In: *SIGPLAN Not.* (2012).

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[Han12] Robert Hanmer. *Pattern-oriented software architecture for dummies*. John Wiley & Sons, 2012.

[HLWL14] Simon Harrer, Jörg Lenhard, Guido Wirtz, and Tammo van Lessen. “Towards Uniform BPEL Engine Management in the Cloud.” In: *Proceedings des CloudCycle14 Workshops auf der 44. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*. Gesellschaft für Informatik e.V. (GI), 2014.

[HLNW14] Florian Haupt, Frank Leymann, Alexander Nowak, and Sebastian Wagner. "Lego4TOSCA: Composable Building Blocks for Cloud Applications." In: *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD 2014)*. IEEE, 2014.

[HBBL14] Pascal Hirmer, Uwe Breitenbücher, Tobias Binz, and Frank Leymann. "Automatic Topology Completion of TOSCA-based Cloud Applications." In: *Proceedings des CloudCycle14 Workshops auf der 44. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*. Gesellschaft für Informatik eV (GI), Sept. 2014.

[HKOH13] Eman Hossny, Sherif Khattab, Felix Omara, and Haitham Hassan. "A Case Study for Deploying Applications on Heterogeneous PaaS Platforms." In: *2013 International Conference on Cloud Computing and Big Data (CloudCom-Asia)*. IEEE, 2013.

[KMO98] Bartek Kiepuszewski, Ralf Muhlberger, and Maria E. Orlowska. "FlowBack: Providing Backward Recovery for Workflow Management Systems." In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. ACM, 1998.

[Ley09] Frank Leymann. "Cloud Computing: The Next Revolution in IT." In: Wichmann Verlag, 2009.

[LR98] Frank Leymann and Dieter Roller. "Building A Robust Workflow Management System With Persistent Queues and Stored Procedures." In: *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE)*. IEEE, 1998.

[Mie10] Ralph Mietzner. "A method and implementation to define and provision variable composite applications, and its usage in cloud computing." 2010.

[Sam97] Johannes Sametinger. *Software engineering with reusable components*. Springer Science & Business Media, 1997.

[SLTP+14] Chrystalla Sofokleous, Nicholas Loulloudes, Demetris Trihinas, George Pallis, and MariosD. Dikaiakos. "c-Eclipse: An Open-Source Management Framework for Cloud Applications." In: *Euro-Par 2014 Parallel Processing*. Springer International Publishing, 2014.

[vDTKB03] Wil MP van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. "Workflow patterns." In: *Distributed and Parallel Databases* (2003).

[WF12] Tim Wellhausen and Andreas Fiesser. "How to Write a Pattern?: A Rough Guide for First-time Pattern Authors." In: *Proceedings of the 16th European Conference on Pattern Languages of Programs*. ACM, 2012.

[Zdu07] Uwe Zdun. "Systematic pattern selection using pattern language grammars and design space analysis." In: *Software: Practice and Experience* (2007).

8.2 Online Resources

[Agi] Agile Orbit. *java Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/java (visited on 06/16/2015).

[Alfa] Alfresco Software, Inc. *Activiti*. URL: http://activiti.org/ (visited on 10/13/2015).

[Alfb] Alfresco Software, Inc. *RepositoryService (Activiti - Engine 5.18.0 API)*. URL: http://activiti.org/javadocs/index.html (visited on 10/01/2015).

[Alfc] Alfresco Software, Inc. *RuntimeService (Activiti - Engine 5.18.0 API)*. URL: http://activiti.org/javadocs/index.html (visited on 10/01/2015).

[Apaa] Apache Software Foundation. *Apache ODE textendash Apache ODEtexttrademark*. URL: http://ode.apache.org/ (visited on 10/09/2015).

[Apab] Apache Software Foundation. *Apache Tomcat 7 (7.0.64) - Manager App HOW-TO*. URL: https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html (visited on 08/26/2015).

[BK] Heiko Braun and Kabir Khan. *Admin Guide - JBoss AS 7.0 - Project Documentation Editor*. URL: https://docs.jboss.org/author/display/AS7/Admin+Guide#AdminGuide-HTTPManagementEndpoint (visited on 10/01/2015).

[Cana] Canonical Ltd. *About Juju | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/about-juju (visited on 07/01/2015).

[Canb] Canonical Ltd. *About Juju | Juju*. URL: https://jujucharms.com/about (visited on 10/12/2015).

[Canc] Canonical Ltd. *Charm hooks | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/authors-charm-hooks (visited on 07/06/2015).

[Cand] Canonical Ltd. *Constraints | Documentation | Juju*. URL: https://jujucharms.com/docs/devel/reference-constraints (visited on 09/28/2015).

[Cane] Canonical Ltd. *Creating and using Bundles | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/charms-bundles (visited on 09/28/2015).

[Canf] Canonical Ltd. *Deploying Services | Documentation | Juju*. URL: https://jujucharms.com/docs/devel/charms-deploying (visited on 09/13/2015).

[Cang] Canonical Ltd. *implementing actions in juju charms | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/authors-charm-actions (visited on 09/12/2015).

[Canh] Canonical Ltd. *Implementing Relations in Juju charms | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/authors-relations (visited on 09/13/2015).

[Cani] Canonical Ltd. *Implicit Relations | Documentation | Juju*. URL: https://jujucharms.com/docs/devel/authors-implicit-relations (visited on 09/13/2015).

[Canj] Canonical Ltd. *Machine Constraints | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/charms-constraints (visited on 09/28/2015).

[Cank] Canonical Ltd. *Managing Relationships | Documentation | Juju*. URL: https://jujucharms.com/docs/stable/charms-relations (visited on 09/28/2015).

[Canl] Canonical Ltd. *Store | Juju*. URL: https://jujucharms.com/store (visited on 07/06/2015).

[Canm] Canonical Ltd. *Welcome to the Juju charm browser | Juju*. URL: https://jujucharms.com (visited on 05/19/2015).

[chaa] "charmers" team. *apache2 | Juju*. URL: https://jujucharms.com/apache2/trusty/14 (visited on 07/06/2015).

[chab] "charmers" team. *haproxy | Juju*. URL: https://jujucharms.com/haproxy/trusty/11 (visited on 07/06/2015).

[chac] "charmers" team. *mysql | Juju*. URL: https://jujucharms.com/mysql/trusty/25 (visited on 07/06/2015).

[chad] "charmers" team. *postgresql | Juju*. URL: https://jujucharms.com/postgresql/trusty/23 (visited on 07/06/2015).

[chae] "charmers" team. *rabbitmq server | Juju*. URL: https://jujucharms.com/rabbitmq-server/trusty/32 (visited on 07/06/2015).

[Chea] Chef Software, Inc. *About Cookbooks textemdash Chef Docs*. URL: https://docs.chef.io/cookbooks.html (visited on 10/12/2015).

[Cheb] Chef Software, Inc. *About Cookbooks textemdash chef-client 12.4 Documentation*. URL: https://docs.chef.io/release/12-4/cookbooks.html (visited on 10/12/2015).

[Chec] Chef Software, Inc. *About Definitions textemdash Chef Docs*. URL: https://docs.chef.io/definitions.html (visited on 10/12/2015).

[Ched] Chef Software, Inc. *About Nodes textemdash Chef Docs*. URL: https://docs.chef.io/nodes.html (visited on 09/28/2015).

[Chee] Chef Software, Inc. *About Resources textemdash Chef Docs*. URL: https://docs.chef.io/resource.html (visited on 10/12/2015).

[Chef] Chef Software, Inc. *About Run-lists textemdash Chef Docs*. URL: https://docs.chef.io/run_lists.html (visited on 09/28/2015).

[Cheg] Chef Software, Inc. *apt Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/apt (visited on 06/16/2015).

[Cheh] Chef Software, Inc. *Chef*. URL: https://www.chef.io/ (visited on 05/19/2015).

[Chei] Chef Software, Inc. *Chef Server Components textemdash chef-client 12.4 Documentation*. URL: https://docs.chef.io/release/12-4/server_components.html (visited on 10/12/2015).

[Chej] Chef Software, Inc. *Chef-client*. URL: https://docs.chef.io/chef_client.html (visited on 06/15/2015).

[Chek] Chef Software, Inc. *chef-client Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/chef-client (visited on 06/16/2015).

[Chel] Chef Software, Inc. *Cookbooks - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks?order=most_downloaded (visited on 06/16/2015).

[Chem] Chef Software, Inc. *knife deps | Chef Docs*. URL: http://docs.chef.io/knife_deps.html (visited on 08/22/2015).

[Chen] Chef Software, Inc. *metadata.rb | Chef Docs*. URL: http://docs.chef.io/config_rb_metadata.html (visited on 08/22/2015).

[Cheo] Chef Software, Inc. *mysql Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/mysql (visited on 06/16/2015).

[Doca] Docker, Inc. *pull*. URL: https://docs.docker.com/reference/commandline/pull/ (visited on 10/01/2015).

[Docb] Docker, Inc. *rm*. URL: https://docs.docker.com/reference/commandline/rm/ (visited on 10/01/2015).

[Docc] Docker, Inc. *rmi*. URL: https://docs.docker.com/reference/commandline/rmi/ (visited on 10/01/2015).

[Docd] Docker, Inc. *run*. URL: https://docs.docker.com/reference/commandline/run/ (visited on 10/01/2015).

[Doce] Docker, Inc. *start*. URL: https://docs.docker.com/reference/commandline/start/ (visited on 10/01/2015).

[Docf] Docker, Inc. *stop*. URL: https://docs.docker.com/reference/commandline/stop/ (visited on 10/01/2015).

[Docg] Docker, Inc. *What is Docker?* URL: https://www.docker.com/whatisdocker (visited on 10/13/2015).

[Esc] Escape Studios Development. *newrelic Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/newrelic (visited on 06/16/2015).

[Fie] Mike Fiedler. *nginx Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/nginx (visited on 06/16/2015).

[Fla] Brian Flad. *docker Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/docker (visited on 06/16/2015).

[IBMa] IBM Corporation. *Anwendungen bereitstellen*. URL: https://www.ng.bluemix.net/docs/manageapps/deployingapps.html (visited on 10/08/2015).

[IBMb] IBM Corporation. *Auto-Scaling - IBM Bluemix*. URL: https://console.ng.bluemix.net/catalog/auto-scaling/ (visited on 10/01/2015).

[IBMc] IBM Corporation. *Befehlszeilenschnittstelle*. URL: https://www.ng.bluemix.net/docs/cli/cli.html#container_cli (visited on 08/22/2015).

[IBMd] IBM Corporation. *Bluemix - Übersicht*. URL: https://www.ng.bluemix.net/docs/overview/overview.html#ov_arch (visited on 10/08/2015).

[IBMe] IBM Corporation. *Boilerplates*. URL: https://www.ng.bluemix.net/docs/starters/boilerplates.html (visited on 09/28/2015).

[IBMf] IBM Corporation. *Deploying with Application Manifests | Pivotal Docs*. URL: http://docs.pivotal.io/pivotalcf/devguide/deploy-apps/manifest.html#precedence (visited on 10/22/2015).

[IBMg] IBM Corporation. *IBM Bluemix*. URL: http://www.ibm.com/cloud-computing/bluemix/ (visited on 05/19/2015).

[IBMh] IBM Corporation. *IBM Bluemix - Entwicklungsplattform für Cloud-Apps der nächsten Generation*. URL: https://console.ng.bluemix.net/home/ (visited on 10/08/2015).

[IBMi] IBM Corporation. *Java DB Web Starter - IBM Bluemix*. URL: https://console.ng.bluemix.net/catalog/java-db-web-starter/ (visited on 10/01/2015).

[IBMj] IBM Corporation. *Katalog - IBM Bluemix*. URL: https://console.ng.bluemix.net/catalog/ (visited on 10/08/2015).

[IBMk] IBM Corporation. *Laufzeiten - Übersicht*. URL: https://www.ng.bluemix.net/docs/starters/rt_landing.html (visited on 09/28/2015).

[IBMl] IBM Corporation. *Webanwendungen erstellen*. URL: https://www.ng.bluemix.net/docs/starters/index.html (visited on 10/08/2015).

[IBMm] IBM Corporation. *Wert der Umgebungsvariable VCAP_SERVICES abrufen*. URL: https://www.ng.bluemix.net/docs/cli/retrieving.html (visited on 10/02/2015).

[Juj] "Juju GUI Charmers" team. *juju gui | Juju*. URL: https://jujucharms.com/juju-gui/trusty/33 (visited on 07/06/2015).

[KPM] KPMG AG in collaboration with Bitkom Research GmbH. *Cloud-Monitor 2015. Cloud-Computing in Deutschland textendash Status quo und Perspektiven - Cloud_Monitor_2015_KPMG_Bitkom_Research.pdf*. URL: https://www.bitkom.org/Publikationen/2015/Studien/Cloud-Monitor-2015/Cloud_Monitor_2015_KPMG_Bitkom_Research.pdf (visited on 10/28/2015).

[McL] Bryan McLellan. *windows Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/windows (visited on 06/16/2015).

[Mic] Microsoft Corporation. *Professionelle Diagramm- und Flowchart-Software | Microsoft Visio*. URL: https://products.office.com/de-de/visio/flowchart-software (visited on 10/16/2015).

[MSSM+] Mohamed Sellami, Sami Yangui, Samir Tata, Mohamed Mohamed, and Chan Ngoc Nguyen. *COAPS API*. URL: http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/ (visited on 10/01/2015).

[OASa] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. URL: docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html (visited on 05/25/2015).

[OASb] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html (visited on 05/25/2015).

[Obja] Object Management Group, Inc. *BPMN Specification - Business Process Model and Notation*. URL: http://www.bpmn.org/ (visited on 10/16/2015).

[Objb] Object Management Group, Inc. *Business Process Model and Notation (BPMN) Version 2.0*. URL: http://www.omg.org/spec/BPMN/2.0/PDF/ (visited on 10/16/2015).

[Objc] Object Management Group, Inc. *Unified Modeling Language (UML)*. URL: http://www.uml.org/ (visited on 10/16/2015).

[Opea] "OpenStack Charmers" team. *glance | Juju*. URL: https://jujucharms.com/glance/trusty/22 (visited on 07/06/2015).

[Opeb] "OpenStack Charmers" team. *keystone | Juju*. URL: https://jujucharms.com/keystone/trusty/26 (visited on 07/06/2015).

[Opec] "OpenStack Charmers" team. *nova cloud controller | Juju*. URL: https://jujucharms.com/nova-cloud-controller/trusty/58 (visited on 07/06/2015).

[Oped] "OpenStack Charmers" team. *openstack dashboard | Juju*. URL: https://jujucharms.com/openstack-dashboard/trusty/14 (visited on 07/06/2015).

[Reda] Red Hat, Inc. *JBoss Developer*. URL: http://www.jboss.org/ (visited on 10/13/2015).

[Redb] Red Hat, Inc. *JBoss Web - JBoss Web Web Application Deployment*. URL: https://docs.jboss.org/jbossweb/3.0.x/deployer-howto.html (visited on 10/13/2015).

[Thea] The Hillside Group. *EuroPLoP*. URL: http://hillside.net/conferences/europlop (visited on 10/19/2015).

[Theb] The Hillside Group. *PLoP*. URL: http://hillside.net/conferences/plop (visited on 10/19/2015).

[Unia] Universität Stuttgart - Institut für Architektur von Anwendungssystemen. *OpenTOSCA - Open Source TOSCA Ecosystem*. URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/ (visited on 05/21/2014).

[Unib] Universität Stuttgart - Institut für Architektur von Anwendungssystemen. *PatternPedia - Wiki-based Pattern Repository*. URL: http://www.iaas.uni-stuttgart.de/forschung/projects/PatternPedia/index.php (visited on 10/23/2015).

[Unic] Universität Stuttgart - Institut für Architektur von Anwendungssystemen. *Winery | projects.eclipse.org*. URL: https://projects.eclipse.org/projects/soa.winery (visited on 10/22/2015).

[vZoe] Sander van Zoest. *apache2 Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/apache2 (visited on 06/16/2015).

[Var] Seth Vargo. *bacon Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/bacon (visited on 06/16/2015).

[VMw] VMware, Inc. *Bare-Metal-Hypervisor vSphere ESXi | VMware Deutschland*. URL: https://www.vmware.com/de/products/esxi-and-esx/overview (visited on 10/13/2015).

[Win] Jamie Winsor. *artifact Cookbook - Chef Supermarket*. URL: https://supermarket.chef.io/cookbooks/artifact (visited on 06/16/2015).

9 Appendix

In the appendix, all figures and tables included in this document are listed.

9.1 List of Figures

1.1 Statistic about the usage of cloud computing in German companies between 2011 and 2014 7

2.1 Sketch of the Pattern Identification, Authoring, and Application Process 13
2.2 Sketch of the pattern writing approach 14
2.3 Chef-client configuration process of an application 17

3.1 Sketch of the phase *Pattern Identification* 21
3.2 Sketch of the phase *Pattern Authoring* 23
3.3 Sketch of the phase *Pattern Application* 24
3.4 Sketch of the Adapted Process 25
3.5 Sketch of the Adapted Pattern Authoring Phase 25

5.1 Example Sequence Diagram 33
5.2 Example Imperative Process Model 33
5.3 Example Topology 34
5.4 Example Management System 34
5.5 Example Management Access 34
5.6 Example Technical Statements 34
5.7 Example Data Access 35
5.8 Example placeholder for an icon 38
5.9 Example of a solution sketch 39

6.1 Solution Sketch: Imperative Provisioning pattern 43
6.2 Solution Sketch: Declarative Provisioning pattern 46
6.3 Solution Sketch: Parameterized Imperative Provisioning pattern 48
6.4 Solution Sketch: Local Management Operation Execution pattern 51
6.5 Solution Sketch: Component Lifecycle Interface pattern 53
6.6 Solution Sketch: Container Component Interface pattern 55
6.7 Solution Sketch: Explicit Dependency Model pattern 58

6.8 Solution Sketch: Implicit Dependency Model pattern 60
6.9 Solution Sketch: External Instance Data Access pattern candidate 62
6.10 Overview of the Pattern Language 64

9.2 List of Tables

4.1 Chef marketplace - total downloads ranking 27
4.2 Cookbook API characteristics 28
4.3 Juju store - total deployment ranking 29
4.4 Charm API characteristics 29

5.1 Definition of the pattern primitives of the Application Provisioning Modeling Pattern Language 38

7.1 Overview of the Known Uses of the patterns and pattern candidates 66
7.2 Maturity evaluation of the patterns and pattern candidates 67

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.

Done at Stuttgart, 04 November 2015.