Modeling and Analyzing Software Ecosystems

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Bahar Schwichtenberg

Faculty of Computer Science, Electrical Engineering and Mathematics
Paderborn University

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

January 2022

To Simon and Nick

Abstract

Nowadays, successful software companies attain enhanced business objectives by creating software ecosystems by opening their platforms to thousands of third-party providers. When creating software ecosystems, many architectural design decisions have to be made. The set of decisions results in an overwhelming design space of architectural variabilities. Until now, there are no architectural guidelines and tools that explicitly capture the design of ecosystem architecture. As a result, systematic knowledge is missing; platform providers have to fall back to ad-hoc decision-making; this bears consequences such as unhealthy ecosystems with risks of failure and extra costs. The lack of architectural knowledge is specifically manifested in two major groups of challenges: challenges related to a lack of an architectural knowledge base and challenges related to a lack of methodical knowledge. An architectural knowledge base would provide guidance on constituents of software ecosystems and their interdependencies while methodical knowledge would facilitate the creation of these systems.

In this thesis, we address the described challenges by developing an ecosystem modeling architecture framework called SecoArc. The contribution of this thesis is twofold: a) the SecoArc framework comprises a knowledge base that contains reusable architectural design decisions of software ecosystems. We develop the knowledge base by developing a software ecosystem product line by extracting and consolidating the architectural knowledge of existing ecosystems and literature. The product line comprises architectural commonalities and variabilities of software ecosystems. And, b) the SecoArc framework provides methodical knowledge to design and analyze the ecosystem architecture in models. We develop this knowledge by first identifying three architectural patterns. Each pattern captures different relations between architectural design decisions to the quality attributes of ecosystem health and business objectives. We use the architectural patterns and the product line to develop a modeling framework that includes a design process, modeling language, architectural analysis technique. The modeling framework facilitates modeling, analyzing, and comparing the architectures.

We implemented the SecoArc framework in a prototype. Our validation approach consists of two studies that focus on the suitability of the framework. In our first validation study, the framework, as well as the prototype, are used to design and analyze two alternative ecosystem architectures. In the second study, we perform a comparative analysis of exiting ecosystems based on the architectural variabilities of the framework.

Zusammenfassung

Heutzutage erreichen erfolgreiche Softwareunternehmen ihre Geschäftsziele, indem sie Software-Ökosysteme schaffen, wo die Software-Plattformen für Tausende von Drittanbietern geöffnet geworden sind. Bei der Erstellung von Software-Ökosystemen müssen viele architektonische Designentscheidungen getroffen werden. Die Menge an Entscheidungen führt zu einem überwältigenden Gestaltungsraum architektonischer Variabilitäten. Bisher gibt es keine Architekturrichtlinien und -werkzeuge, die das Design der Ökosystemarchitektur explizit erfassen. Dadurch fehlt systematisches Wissen; Plattformanbieter müssen auf ad-hoc Entscheidungen zurückgreifen; dies hat Folgen wie ungesunde Ökosysteme mit Ausfallrisiken und Mehrkosten. Der Mangel an Architekturwissen manifestiert sich konkret in zwei großen Gruppen von Herausforderungen: Herausforderungen im Zusammenhang mit fehlender Architekturwissensbasis und Herausforderungen im Zusammenhang mit fehlendem Methodenwissen. Eine Architekturwissensbasis würde Orientierungshilfen zu den Bestandteilen von Software-Ökosystemen und deren Abhängigkeiten geben, während methodisches Wissen die Erstellung dieser Systeme erleichtern würde.

In dieser Dissertation gehen wir die beschriebenen Herausforderungen durch die Entwicklung eines Frameworks für die Ökosystemmodellierung namens SecoArc an. Der Beitrag dieser Dissertation ist zweifach: a) Das SecoArc-Framework umfasst eine Architekturwissensbasis, die wiederverwendbare Architekturentwurfsentscheidungen von Software-Ökosystemen enthält. Wir entwickeln die Wissensbasis, indem wir das Architekturwissen bestehender Ökosysteme und Literatur extrahieren und in einer Software-Ökosystem-Produktlinie konsolidieren. Die Produktlinie umfasst architektonische Gemeinsamkeiten und Variabilitäten von Software-Ökosystemen. b) Das SecoArc-Framework liefert methodisches Wissen, um die Ökosystemarchitektur in Modellen zu entwerfen und zu analysieren. Dieses Wissen entwickeln wir, indem wir zunächst drei Architekturmuster identifizieren. Jedes Muster erfasst unterschiedliche Beziehungen zwischen architektonischen Designentscheidungen zu den Qualitätsmerkmalen der Ökosystemgesundheit und den Geschäftszielen. Wir verwenden die Architekturmuster und die Produktlinie, um ein Modellierungsframework zu entwickeln, das einen Designprozess, eine Modellierungssprache und eine Architekturanalysetechnik umfasst. Das Modellierungs-Framework erleichtert das Modellieren, Analysieren und Vergleichen der Ökosystemarchitekturen.

Das SecoArc-Framework wurde im Rahmen eines Prototypen umgesetzt. Der Validierungsansatz besteht aus zwei Studien, die sich auf die Eignung des Frameworks konzentrieren. In der ersten Validierungsstudie werden das Framework sowie der Prototyp verwendet, um zwei alternative Ökosystemarchitekturen zu entwerfen und zu analysieren. In der zweiten Studie führen wir eine Analyse von existierenden Ökosystemen basierend auf den architektonischen Variabilitäten des Frameworks durch.

Acknowledgements

Writing this thesis would have never been possible without the support of many people.

I owe my deepest gratitude to my advisor Gregor Engels for supporting me through this journey. Gregor, I thank you for being open and trusting in me. Thank you for the opportunities that let me develop myself personally and professionally, and for being supportive whenever I felt lost. I would like to thank Olaf Zimmermann and Jochen Küster for our cooperation. You taught me new ways to surpass my limits. I have been deeply connected to the Collaborative Research Center 901 - On-the-Fly Computing for several years, which is why I am honored and particularly grateful for Friedhelm Meyer auf der Heide being a member of my examination committee as chairman of the Collaborative Research Center. I would like to thank Enes Yigitbas for the numerous constructive scientific discussions during my PhD.

In addition, I would like to thank my colleagues from the University of Paderborn, especially the colleagues at the Chair for Databases and Information Systems and, of course, the Collaborative Research Center for the inspiring conversations and the enjoyable collaborations.

I would like to express my gratitude to my friends and family. I thank Jan, Amelie, Andrea, Andy, and Aurora for reading my draft, and Judith for the encouraging conversations and coffee breaks. I also would like to thank my loving family, specially Homa, Sadegh, and Mahnaz for the lifetime believing in me, and my sisters Behnaz and Behrokh for their unconditional support and kindness. Thank you Simon for your dedication and love. It is hard to imagine this moment of success without your support. Last but not least, I would like to thank my little son, Nick, for being a powerful source of inspiration in my life.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Architectural descriptions are crucial pieces of knowledge for large-scale software systems to restrain complexity and leverage systems' functionality to achieve business objectives. Thus, understanding the architecture is vital for software success. However, as software grows, its structure is subject to even more complexity. In the last decades, many architectural paradigms, from the component-based software engineering and service-oriented computing, and up to the more recent perspectives such as perceiving the software via its set of architectural design decisions, all strive for an abstraction that can help deal with complexity and achieve system objectives [CJT^{+}16, SC06]. Nevertheless, by the increasing market competition and technological advancement, software use-cases have become increasingly demanding as well. In response to the demanding use-cases, software functionality is expected to advance. This, in return, requires novel architectural supports.

1.1 Software Ecosystems

Pioneers of today's software industry succeed to immensely grow by transforming their software products to *open platforms* whose functionality can be extended by third-party providers using open Application Programming Interfaces (APIs). Usually, online stores are used to distribute third-party developments among users. With this respect, *software ecosystems* are created. A software ecosystem is a collection of human actors interacting with each other and with the software elements. The term software ecosystem is inspired by ecological ecosystems that are the result of an interplay between organisms as well as interactions with a physical environment [Bos09, SHMS17].

Figure 1.1 depicts the key constituent elements of software ecosystems. A platform provider is the owner of a software platform and an online store. Third-party providers

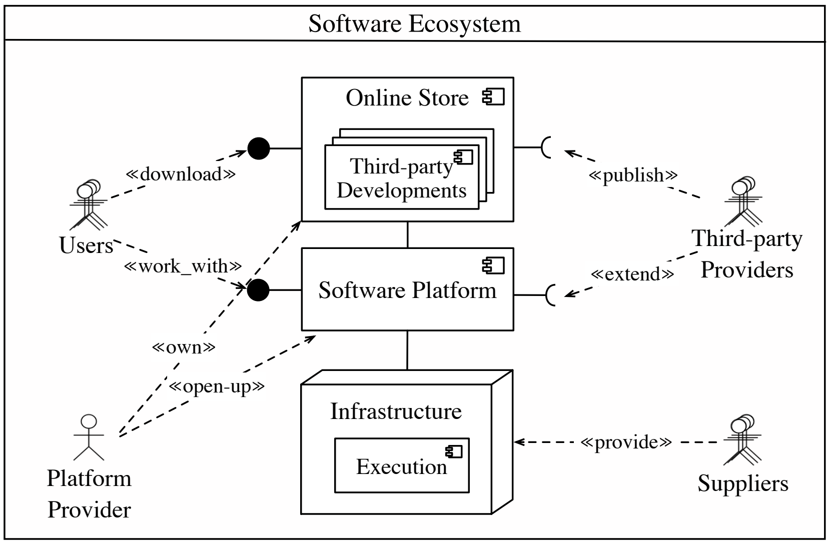


Fig. 1.1 Simplified Illustration of Software Ecosystems

can extend the functionality of the platform by developing third-party developments on the basis of the platform. The third-party developments are published on an online store, where the users can download them. In some ecosystems, the platform provider handles the service provision in cooperation with suppliers, who provide the required software or hardware resources. An example of this situation is when the third-party developments are cloud computing services that are executed on external servers provided by the suppliers [BB10a]. Google[1] is the provider of an ecosystem in which independent developers provide mobile Apps to the users of the Android platform, on an online store called Google Play[2]. An example of suppliers in the ecosystem is Samsung[3], which produces mobile devices for the Android platform.

In software ecosystems, platform providers are the keystone players and the main decision-makers, whose decisions regarding the design and governance of an ecosystem determine the health of the entire ecosystem [JBSL12]. Health of software ecosystems is inspired by the ecological ecosystems and refers to the overall well-functioning and performance of software ecosystems [BHSM13].

The idea of software ecosystems has its roots in other relevant concepts, e.g., *IT service markets* that offer a registry of services to develop web applications based on principles of service-oriented computing [SO11]. Examples of real-world IT service markets are *ProgrammableWeb.com* and *RapidAPI.com*[4]. Furthermore, the concept

[1]`https://www.google.com`, Last Access: January 10, 2022.
[2]`https://play.google.com/store`, Last Access: January 10, 2022.
[3]`https://samsung.com`, Last Access: January 10, 2022.
[4]Formerly known as `mashup.com`

of *business ecosystem* has been around for few decades. It describes an economic community of interacting stakeholders, where the stakeholders co-evolve overtime while their evolution is aligned with strategies defined by a central company [CM98]. One may consider software ecosystems as business ecosystems around software platforms. Yet, by the launch of Apple App Store and Google Play in 2008, *software ecosystem* became an established term in computer science [JFB09].

1.2 Problem Statement

Although some leading platform providers have succeeded to create their ecosystems, there is still a lack of systematic knowledge to develop healthy and sustainable software ecosystems [SHMS17, Man16, MH13]. Until now, in the absence of any reference model or, in general, a well-established common source of architectural knowledge, platform providers had to rely on ad-hoc decision-making [AS16, HSN$^+$15, Bos10]. This situation resulted in the sub-optimal architectural decision-making, where ecosystems were discontinued due to poor business models [Has18], developers left the ecosystems [Gol10], or malware and spams dominated the service provision [Kwa17].

Reports of failures show erroneous design decisions frequently made by the platform providers resulted in extra costs due to developing useless features that were rarely used in practice [BUI16]. Furthermore, the lack of strategic decision-making led to unwillingly exposing companies' intellectual property and causing serious business risks [Fau19, ONE12]. In addition, technical debt was continually imposed on the systems due to the sub-optimal decision-making made by separately located development teams. Technical debt refers to when software development processes are sped up by making fast but unconsolidated decisions, hoping for a solid reconstruction in future [DLCA17].

One facet of the problem is the missing knowledge that could help platform providers directly address critical architectural aspects of software ecosystems [SY15]. Figure 1.2 depicts a hierarchy of application domains based on the generality of their concepts. Domains with a narrower scope are subsets of the domains with a broader scope [VBD$^+$13, chap. 3]. Software ecosystems as an application domain have several sub-domains, e.g., open-source software development, mobile App, and cloud computing. Until now, architectural knowledge is developed to create software ecosystems in specific sub-domains. For instance, the ecosystems around the mobile App and open-source software development platforms are amongst the most studied ones due to the high popularity and availability of data [DHS19, FMM15, MI11].

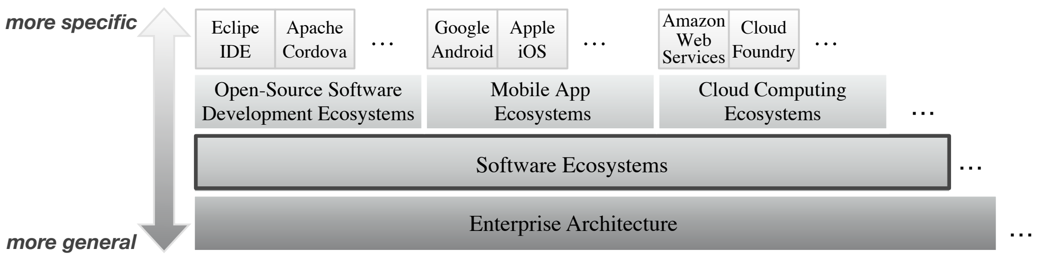


Fig. 1.2 Lack of Architectural Knowledge to Design Software Ecosystems

However, such architectural knowledge bears certain generalization concerns by being specific to those sub-domains. This results in the limited applicability to capture architectural characteristics of ecosystems in other application domains. Furthermore, as depicted in Figure 1.2, the architecture of software ecosystems is considered as a specific kind of enterprise architecture that is extended to the outside of enterprises by the direct involvement of third-party providers [JFB09, JBSL12]. With this respect, a multitude of Architecture Description Languages (ADLs) and frameworks exits that can be used for designing the architecture of software ecosystems. An example is the ArchiMate language[5]. Together with the TOGAF architecture framework[6], they facilitate enterprise architecture development. However, the abundance of notations in ArchiMate leads in practice to laborious and time-consuming approaches [HOZZ16]. In addition, while TOGAF is well-suited to specify the relations inside enterprises, it lacks a support to capture the relations between the enterprises and specifically entrance barriers that are often applied in software ecosystems [MSS$^+$13, AG19].

Another facet of the problem is that so far only single aspects have been investigated, but a complete view of organizational, business, and technical aspects is missing [BdA$^+$13, SHMS17]. One main reason is the lack of a comprehensive understanding of the interdisciplinary architectural aspects of software ecosystems. Therefore, it is not clear what the relations between these aspects are. Some works in literature [HO11, SO11] develop generic reference models for App stores that mostly focus on the elements of business architecture. However, technical aspects such as the development and execution of third-party developments are barely addressed. Moreover, some other works [WB15, HHYH09] provide a detailed perspective on technical aspects, while neglecting the relation to business decisions. For instance, openness is a major business concern that platform providers face. Suitable mechanisms at the technical level are needed to realize fine-grained openness policies in software ecosystems, in

[5]`www.opengroup.org/archimate-forum/archimate-overview`, Last Access: January 10, 2022.
[6]`www.opengroup.org/togaf`, Last Access: January 10, 2022.

order to enable third-party software development while reducing the risk of unwillingly exposing the intellectual property [SY17b].

The lack of architectural knowledge can be specifically considered in terms of two major groups of challenges that platform providers face while designing software ecosystems: Firstly, the challenges related to *a lack of an architectural knowledge base (C1-C3)*. Secondly, the challenges originated by *a lack of methodical architectural guidance (C4-C7)*. In the following, we discuss these challenges and refer to why current solutions do not suffice to overcome the challenges.

A Lack of an Architectural Knowledge Base

A lack of an architectural knowledge base refers to the absence of systematic knowledge regarding the structure of software ecosystems.

C1: Unknown Architectural Design Decisions

Reports show that a main challenge is that the platform providers miss to include core functionalities of software ecosystems such as suitable search functions and rating [VDGS18, SEL14]. The challenge is related to the lack of a unified view of interdisciplinary design decisions related to the organizational, business, and technical aspects of software ecosystems. Another problem is that many ecosystem functionalities are not applied in practice although they have been investigated and implemented in academia for several years. For instance, there are still a multitude of ecosystems, which, despite having a large market of third-party developments, miss key features like search, ranking, and recommendation system that makes the process of identifying the right third-party developments on the stores inefficient for the users.

C2: Bewildering Architectural Variability

When transforming a software product to an open platform and creating an ecosystem around it, many architectural design decisions have to be made, which are derived from companies' business objectives. The set of decisions results in an overwhelming design space of architectural variabilities that is bewildering for the platform providers [REV19, BPT^{+}14]. For instance, *what are different kinds of openness policies in software ecosystems? How can they be implemented? What are their relations to the choice of licensing?* The bewildering architectural variability is the reflected in a wide range of ecosystems created in diverse application domains. With this respect, "generalization is an issue. Due to the high variability in the field" [Man16]. First of all, architectural knowledge of the existing ecosystems cannot be directly used by other platform providers since this knowledge is implicit, i.e., available in a fragmentary way among online documentation and portals of those ecosystems. In addition, in the protection of intellectual property, existing documentation hardly reveals influential business decisions that affect the ecosystem architecture.

C3: Obscure Decision Interdependencies

Service provision is a complex task in software ecosystems that involves interdependent design decisions. In addition, bewildering architectural variability adds additional complexity to the management of decision interdependencies. In practice, platform providers are not often aware of mutual dependencies between the decisions. Therefore, they miss considering the interdependencies, e.g., the decisions that are enforced or excluded by their currently made decisions [BCH15]. For example, some ecosystems suffer from problems related to fake ratings. In many cases, the reason is the lack of security features that can ensure the integrity of the ratings because the platform providers are not aware of the interplay with the rating and security features.

A Lack of Methodical Architectural Guidance

A lack of methodical guidance refers to the missing methods, techniques, and tools that can be used to design ecosystem architecture.

C4: Trial Approaches To Address Business Objectives

A major part of architectural design decisions is derived from platform providers' business objectives. Part of the lack of systematic knowledge is related to the linkage between the architecture of software ecosystems and business objectives. Thus, to address business objectives, platform providers have to rely on their own experiences and spend their time and budget in processes that are based on tries and errors [MH13]. For example, even successful software companies have struggled in the past to incorporate creativity and innovation into their processes once they aimed for including external providers because they were too focused on their own business model [Roh19].

C5: Cumbersome To Design Ecosystem Architecture

As mentioned earlier, some of the existing work has generalization issues for the domain of software ecosystems by being specific to a sub-domain whereas another group of work imposes extra time and effort to design software ecosystems by being too general. Examples are EAM languages and Unified Modeling Language (UML) [OMG17]. All these adversely affect the efficiency of architect's work. Moreover, certain aspects of software ecosystems such as *different types of provider networks* and *openness policies* can not be directly specified by the existing work [DS14, SY17a].

C6: Failing To Prospect Ecosystem Health

Deficit decision-making, e.g., poor cross-team communications (between the platform providers and partners) or a lack of technical support for developers, harmed the sustainable well-functioning of software ecosystems in the past. Therefore, unhealthy ecosystems failed to continue to function. The root of the challenge resided in the lack of knowledge to address and ensure the health of their ecosystems [Sou18]. Some works [LYW+19, Jan14, HSN+15] identify relevant quality attributes associated with the healthiness of software ecosystems. However, there is a knowledge gap that is related to the missing connections between the quality attributes of ecosystem health and architectural design decisions, and how to prospect the fulfillment of those quality attributes during the design of software ecosystems.

C7: Lack of Tool Support

Currently, textual templates described by natural languages, i.e., the work proposed by Bosch and Bosch-Sijtsema [BB14], are available to the platform providers that help them guide the process of modeling and analyzing software ecosystems. However, using such templates is tedious and time-consuming in the first place. Another drawback is the poor analyzability and reusability of such templates during ecosystem evolution that makes these approaches inefficient in practice. While these templates can be seen as complementary documentation, they lack a structured and precise foundation for developing formal design and analysis techniques. An alternative option would be to use existing enterprise modeling tools such as *Sparx Systems Enterprise Architect* [Spa21]. However, this requires to work with the existing modeling language or to consider extending them with the ecosystem-specific concepts (cf. C5).

In the lack of a architectural knowledge base and methodical guidance to design software ecosystems, platform providers have to only rely on ad-hoc decision-making and face the consequences. This situation hinders the novel creation of healthy software ecosystems in a variety of application domains. Thus, the following problem statement forms the basis of this thesis:

How to systematically design healthy software ecosystems?

1.3 Overview of the Solution

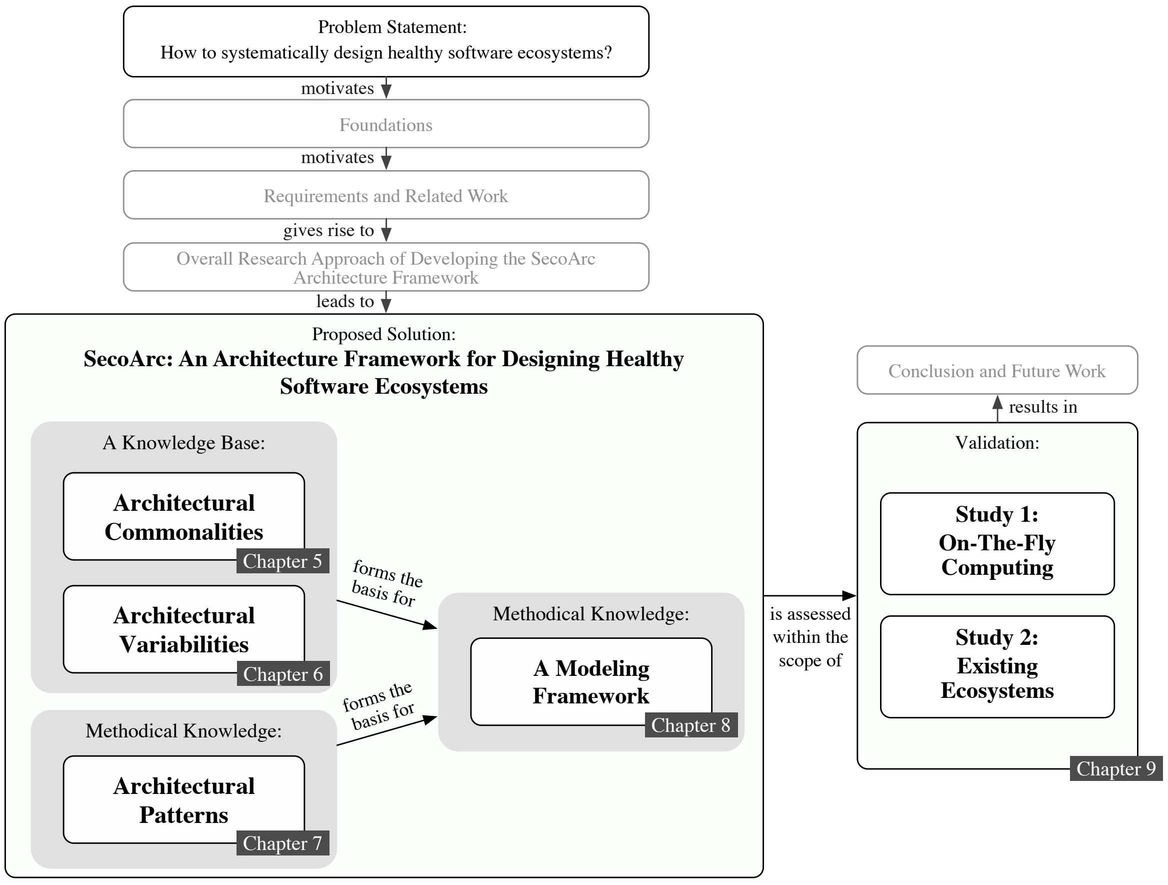
A solution to overcome the problem described in Section 1.2 is to provide novel architectural knowledge, including a structured knowledge base and suitable methodical support, that can assist platform providers in informed architectural decision-making while designing software ecosystems. In this thesis, we develop an *architectural ecosystem modeling framework* called *SecoArc*.

SecoArc is grounded on two orthogonal paradigms in software engineering: Software product line engineering (SPLE) and model-driven engineering (MDE). To deal with the architectural complexity, we examine software ecosystems from an SPLE point of view and develop a software product line for reusable architectural decisions of software ecosystems. We follow the MDE principles to adapt a model-based solution based on the knowledge of the product line. All together, they form the constituents of the SecoArc framework that enable platform providers to design ecosystem architecture and assess the architectural suitability.

Our primary research approach to develop the SecoArc framework follows the *design science methodology* [Wie14]. Design science provides guidelines to create new IT artifacts to improve certain aspects of a problem in the real world. Using the design science's guidelines, suitability of the artifacts is evaluated with respect to those aims. In this thesis, the artifact is an architecture framework with the objective to support platform providers to overcome the lack of architectural knowledge by making informed decision-making. During the development of the SecoArc framework, we perform a pattern-centric approach to extract architectural knowledge of well-established ecosystems in practice. Our research approach encompasses several empirical activities such as the examination of a wide range of existing ecosystems, a systematic literature review, and expert interviews.

Figure 1.3 shows an overview of the thesis's solution. The contributions of this thesis are of two main types: Firstly, SecoArc provides a structured knowledge base by means of a software ecosystem product line that includes *architectural commonalities* and *architectural variabilities* of software ecosystems. Secondly, SecoArc facilitates the methodical development of the ecosystem architecture by providing *three architectural patterns* and *a modeling framework*. The modeling framework comprises *a modeling language*, *architectural analysis technique*, and *design process*. In the following, we introduce these elements and elaborate on how they deal with the challenges discussed in Section 1.2.

In order to deal with the lack of knowledge regarding key architectural design decisions (C1), we identify critical design decisions of software ecosystems that play



Problem Statement:
How to systematically design healthy software ecosystems?
motivates
Foundations
motivates
Requirements and Related Work
gives rise to
Overall Research Approach of Developing the SecoArc Architecture Framework
leads to
Proposed Solution:
SecoArc: An Architecture Framework for Designing Healthy Software Ecosystems
A Knowledge Base:
Architectural Commonalities
Chapter 5
Architectural Variabilities
Chapter 6
forms the basis for
Methodical Knowledge:
A Modeling Framework
Chapter 8
forms the basis for
Methodical Knowledge:
Architectural Patterns
Chapter 7
is assessed within the scope of
Validation:
Study 1: On-The-Fly Computing
Study 2: Existing Ecosystems
Chapter 9
results in
Conclusion and Future Work

Fig. 1.3 Overview of Thesis's Solution

a key role in the ecosystem architecture in Chapter 5. We refer to these decisions as *architectural commonalities* because they are common among software ecosystems, independent of platform providers' organizational contexts and business objectives. Furthermore, we identify the interdependencies between the decisions (C3). The interdependencies show whether and how making a design decision demands other decisions to be considered by platform providers and how considering the interdependencies can influence the overall quality of ecosystems.

To deal with the architectural variability (C2), we identify design decisions that are variable across software ecosystems in different application domains based on an examination of existing ecosystems and the literature. On this basis, we propose a variability model comprising architectural variation points and variants in Chapter 6. The variability model captures design decisions related to the social, business, application, and infrastructure aspects and their relations. We introduce key dependency constraints that pertain to common interdependencies between the variable design decisions. Examples of the dependency constraints are alternative design choices, require-relations when some decisions become necessary to take, or exclude-relations when some decisions should be excluded from the architecture due to certain decision-making (C3). Afterward, we identify alternative scenarios of the overall service provision in software ecosystems based on interviewing experts.

While the architectural commonalities and variabilities form the knowledge base of the SecoArc framework, the framework consists of methodical knowledge to design the ecosystem architecture. To address business objectives and ecosystem health (C4, C6), we identify three architectural patterns for software ecosystems in Chapter 7. The architectural patterns capture the relations between concrete sets of design decisions and their linkage to business objectives and quality attributes of ecosystem health. Well-established ecosystems in practice are a rich source of knowledge. Based on an investigation of 111 ecosystems, we identify the architectural patterns by identifying and classifying recurrent design decisions. We clarify the relation between the architectural patterns and platform providers' organizational context by identifying classes of recurrent contextual factors. The contextual factors refer to organizational characteristics of platform providers. We describe the patterns developing stories and introducing exemplary real-world ecosystems to ease the future application of this knowledge. We provide further insight into the relations between the architectural patterns and their distribution among different application domains.

We develop a modeling framework that facilitates the design and analysis of the ecosystem architecture in Chapter 8. The modeling framework consists of a modeling language, architectural analysis technique, and design process. By means of a modeling language, ecosystem architectures using ecosystem-specific concepts can be specified (C5). The ecosystem-specific concepts are the domain knowledge of software ecosystems that we developed as a knowledge base of the SecoArc framework. Furthermore, an architectural analysis technique enables platform providers to assess the ecosystem architecture with respect to the quality attributes of ecosystem health (C6). Specifically, the analysis matches the ecosystem architecture with the knowledge of the architectural patterns using a set of rules that we developed for this purpose. A report of analysis provides insights into the design decisions made in the architecture and recommendations to improve the ecosystem health. Using the analysis technique, platform providers can perform a deep comparison of alternative architectures. In addition, a design process facilitates the usage of the SecoArc framework for a stepwise design and analysis of ecosystem architectures. Moreover, to address the lack of a tool support (C7), we develop a prototypical tool that is an implementation of the SceoArc framework[7]. It has a formal and precise foundation based on the principles of MDA. It facilitates the design and analysis activities in an integrated modeling workbench.

We validate the SecoArc framework with respect to its objectives by performing two studies in Chapter 9. The first study is conducted within the scope of on-the-fly computing. *On-the-fly Computing (OTF)* [SFB20] is a paradigm for the provision of tailor-made IT services by automatically composing basic services based on individual users' requests. During the validation, the SecoArc framework is used to design two alternative ecosystems around a platform. The ecosystems are designed using the modeling framework implemented in the SecoArc tool. Suitability of the ecosystems is analyzed and compared by using the SecoArc architectural analysis technique. In the second study, we examine existing ecosystems from different application domains on the basis of the SecoArc variability model to assess whether the variabilities introduced by the variability model can be used to capture architectural characteristics of diverse ecosystems. Specifically, we describe and compare variabilities of five ecosystems, i.e., the ecosystems around the Salesforce, Apple, Amazon AWS, Eclipse, and Amazon Alexa platforms.

Notably, the contribution of this thesis resides in extracting and consolidating the knowledge of a wide range of existing software ecosystems and the literature into an architecture framework. To the best of our knowledge, none of the existing approaches reached this goal in such a holistic way while providing an integrated view of the social, business, and technical aspects. Furthermore, this thesis includes several interdisciplinary works with researchers from the fields of *information systems* and *economics*. Besides, our cooperation with researchers outside Paderborn University offered us unique insights into the thesis's research topic.

1.4 Publication Overview

The thesis's solution has been published in peer-reviewed papers at various international conferences and workshops. Figure 1.4 depicts the overview of these publications. The publications are listed in two groups: a) The main publications that directly contribute to the development of the SecoArc framework and b) related research.

Our main publications are dedicated to the constituent elements of the SecoArc framework (cf. the top of Figure 1.4): In [JPEK16a], we identify architectural commonalities by performing a systematic literature review. In this work, we overcome

[7]`https://sfb901.uni-paderborn.de/secoarc`, Last Access: January 10, 2022.

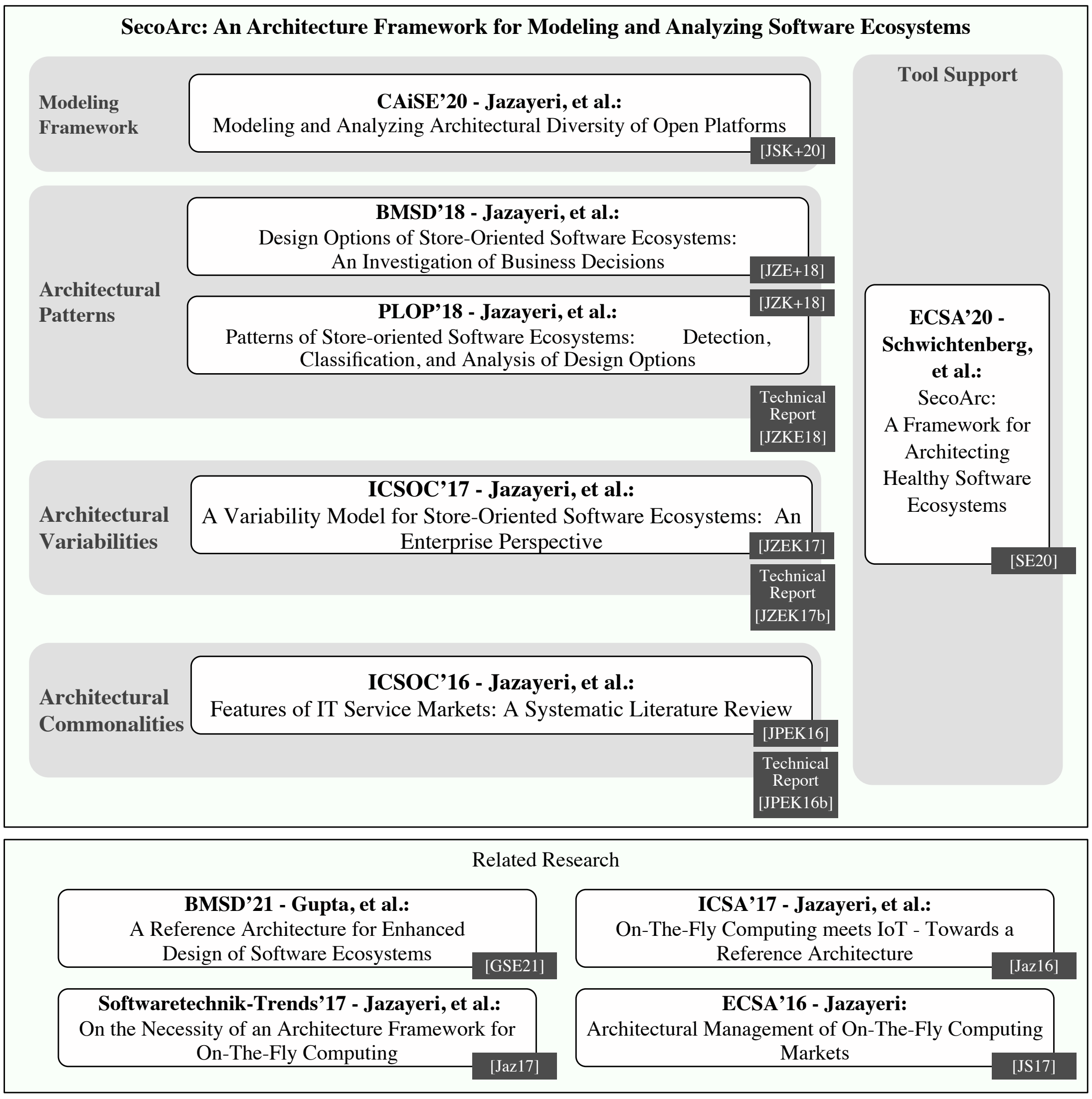

SecoArc: An Architecture Framework for Modeling and Analyzing Software Ecosystems
Modeling Framework
CAiSE'20 - Jazayeri, et al.: Modeling and Analyzing Architectural Diversity of Open Platforms
[JSK+20]
Architectural Patterns
BMSD'18 - Jazayeri, et al.: Design Options of Store-Oriented Software Ecosystems: An Investigation of Business Decisions
[JZE+18]
[JZK+18]
PLOP'18 - Jazayeri, et al.: Patterns of Store-oriented Software Ecosystems: Detection, Classification, and Analysis of Design Options
Technical Report [JZKE18]
Architectural Variabilities
ICSOC'17 - Jazayeri, et al.: A Variability Model for Store-Oriented Software Ecosystems: An Enterprise Perspective
[JZEK17]
Technical Report [JZEK17b]
Architectural Commonalities
ICSOC'16 - Jazayeri, et al.: Features of IT Service Markets: A Systematic Literature Review
[JPEK16]
Technical Report [JPEK16b]
Tool Support
ECSA'20 - Schwichtenberg, et al.: SecoArc: A Framework for Architecting Healthy Software Ecosystems
[SE20]
Related Research
BMSD'21 - Gupta, et al.: A Reference Architecture for Enhanced Design of Software Ecosystems
[GSE21]
ICSA'17 - Jazayeri, et al.: On-The-Fly Computing meets IoT - Towards a Reference Architecture
[Jaz16]
Softwaretechnik-Trends'17 - Jazayeri, et al.: On the Necessity of an Architecture Framework for On-The-Fly Computing
[Jaz17]
ECSA'16 - Jazayeri: Architectural Management of On-The-Fly Computing Markets
[JS17]

Fig. 1.4 Publication Overview

inconsistent terminologies used among the sources by applying grounded theory to conceptualize the commonalities among 60 final sources.

In [JZEK17a], we develop a variability model for software ecosystems by applying a taxonomy development method. In so-called empirical-to-conceptual iterations, we extract the variabilities from existing ecosystems while, in conceptual-to-empirical iterations, this information is complemented by using the literature. Furthermore, this work comprises our second validation study mentioned in Section 1.3.

Development of the architectural patterns are published in two papers: In the first publication [JZE$^+$18], we collect a data-set that includes the architectural design decisions of 111 existing ecosystems. From this data-set, we extract three groups of decisions that are frequently applied together in the ecosystems. In the second publication [JZK$^+$18], platform providers' recurrent contextual factors and their relation to the design decisions are identified. Furthermore, the patterns are described in terms of stories using a well-known pattern template and discussed within the pattern community in a *writers' workshop* [Gab02]. The development of architectural commonalities, variabilities, and patterns relies on the investigation of existing ecosystems and literature. We compiled the data-sets of these investigations in our technical reports [JPEK16b, JZEK17b, JZKE18].

The modeling framework and our first validation study are published in [JSK$^+$20]. This publication comprises an earlier version of the SecoArc design process as well. The design process presented in the thesis additionally captures the procedures of selecting and applying the architectural patterns.

As depicted in Figure 1.4, the SecoAtc tool support spans across all thesis's working packages. We implemented a major part of the framework in the tool and presented it in a tool-demo paper [SE20]. In this work, we discussed the implementation details concerning the architecture of the tool. Two exceptions for the concepts implemented in the tool are the SecoArc design process and social aspect. We expect that the architect performs the design process during the ecosystem design. Specially, this was our goal during the validation. In addition, the tool does not comprises the social concepts that we added to the SecoArc knowledge base in the final phase of the PhD, which was in 2021.

In addition to our main publications, other works related to the thesis have been published that are listed at the bottom of Figure 1.4. In initial works within the scope of on-the-fly computing [Jaz16, JS17b], we briefly discuss the necessity of an architecture framework and suitability of existing frameworks for designing service markets. In another work [JS17a], we use an initial version of the SecoArc framework to examine two existing ecosystems around the Internet-of-Things (IoT) platforms, i.e., IFTTT[8] and Stringify[9]. Based on this examination, we propose requirements of a reference architecture for the IoT ecosystems. In a follow-up work [GSE21], we discuss suitability of the ArchiMate enterprise architecture modeling language in the context of software ecosystems.

[8] `https://ifttt.com`, Last Access: January 10, 2022.
[9] `https://www.stringify.com`, Last Access: February 2017.

1.5 Structure of this Thesis

The thesis is structured into ten chapters. Chapter 2 presents the foundations. Among others, we address foundational research related to software ecosystems, software product line engineering, and model-driven engineering. Chapter 3 concerns the requirements of a solution concept to overcome the challenges discussed in the problem statement. Afterward, a detailed analysis of the problem is provided by discussing and comparing state-of-the-art approaches with respect to the requirements. Chapter 4 outlines an overview of our solution approach to develop the SecoArc framework. Chapter 5 is dedicated to our work on the architectural commonalities, which is followed by Chapter 6 that focuses on the architectural variabilities. Chapter 7 details the identification of three architectural patterns for software ecosystems. Chapter 8 elaborates on the SecoArc modeling framework for the purpose of designing and analyzing the ecosystem architecture. Chapter 9 focuses on the validation studies. Finally, Chapter 10 concludes the thesis and provides an outlook on future research.

Chapter 2

Foundations

This chapter presents an overview of the foundations that are relevant for the concepts developed in this thesis. First, Section 2.1 refers to the concept of enterprise architecture. Afterward, Section 2.2 elaborates on the foundations of software ecosystems and related topics, followed by Section 2.3, where relevant fundamental research on architectural knowledge management is introduced. Section 2.4 is dedicated to software product line engineering. Section 2.5 outlines the concepts concerning model-driven engineering.

2.1 Enterprise Architecture

Enterprise architecture is referred to fundamental and well-established practices that consider the IT system of organizations as a whole, in contrast to the implementation of software systems. In enterprise architecture, design and evolution of an integrated view comprising stakeholders, strategies, business processes, and information systems, and the interactions between these elements such as processes, functions, and data are in focus [SH92, Ses07, Gar]. Enterprise architecture introduced to enhance coping with changes when integration, consistency, and overcoming complexity became more important than to implement solely software systems [WF06].

Since the introduction of enterprise architecture, a multitude of enterprise architecture frameworks are defined that encompass principles to develop and govern enterprise architecture. The Open Group architectural framework (TOGAF) [The11], the Zachman framework [Zac96], and the integrated architecture framework (IAF) [VWWH$^+$10] are examples of such frameworks. While there are differences in the existing enterprise architecture frameworks that refer to their aims and objectives, these frameworks share certain fundamental commonalities. A core principle of these frameworks is to facilitate a closer alignment of business and technical aspects[Sch04].

2.2 Software Ecosystems

Software ecosystems are the result of a collaborative approach between a platform provider and third-party providers to provide software products and services on top of a software platform. The platform is opened to the third-party providers for software development, i.e., the they are enabled to develop software on the basis of the platform, by accessing the source code or using platform APIs. Examples of third-party providers are hardware/software suppliers.

Software ecosystems are understood as extensions of enterprises when the enterprises grow to the outside world by including third-party providers in the processes of service provision [AT16, ZGS^{+}14, JBSL12, ABD13, CFHW12]. This is while the social and organizational aspects in software ecosystems play a crucial role in the architecture. Symbiotic relations between the participants form a social network that allows certain types of communities and collaborations to shape. Therefore, a key part of requirements in software ecosystems are derived from the interaction among the participants and the participants with the software and hardware elements [Val13, TRG15].

2.2.1 Definition of Software Ecosystem

Related literature provides several definitions that each focuses on certain aspects of these systems. However, there is still no single definition for software ecosystems that is agreed upon among different communities. In general, the works providing a definition for software ecosystems can be categorized into three groups.

Social and Organizational View Jansen et al. [JFB09] define software ecosystems as a set of interacting businesses that communicate on the basis of a common technological platform and share a market. The interactions happen with the goals of exchanging information, resources and artifacts. The sustainable interactions between the participants determine whether the ecosystem succeeds to grow over the time.

Business and Management View A group of works [MKM11, BB10b] define software ecosystems from a business viewpoint. These works consider software ecosystems a kind of *business ecosystems* that is created around IT platforms. The definitions provided by these works specifically focus on revenue in software ecosystems. Platform providers' and third-party providers' revenue models are the topics mostly considered by these work.

Application and Infrastructure View Technical aspects of software ecosystems, i.e., application and infrastructure aspects, are the central point by some other works. Bosch and Bosch-Sijtsema [BB10a, Bos09] describe software ecosystems as a result of natural evolution of software product lines, where software development is escalated to the outside of product lines. Specifically, architectural variability is a central topic by such works.

Within the scope of this thesis, we use a definition given by Manikas and Hansen [MH13] that aimed at reconciling all views discussed above. Accordingly, a **software ecosystem** is defined as:

> *"The interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services. Each actor is motivated by a set of interests or business models and connected to the rest of the actors and the ecosystem as a whole with symbiotic relationships, while, the technological platform is structured in a way that allows the involvement and contribution of the different actors."* [MH13]

2.2.2 Ecosystem Architecture

Architecture of a software ecosystem can be viewed from different perspectives. A major part of the works in literature possesses a local perspective to a certain aspects, i.e., social / organizational, business / management, or application / infrastructure aspect as mentioned in Section 2.2.1. To study the design principles of software ecosystems as a whole, a comprehensive view to the all architectural aspects and their interrelations is required. However, such a comprehensive view should avoid being too complex that is an obstacle to understanding key architectural characteristics of ecosystems as well.

The term "*ecosystem architecture*" [CHKM14, Man15, BDS19, AGB19, Jär13, AG19, NSV19] has been used to refer to a view that underlines the main constituents of ecosystems by reducing the complexity into the extension parts of the enterprise architecture that are external to an enterprise, i.e., an open platform and the human actors around it. While ecosystem architecture is not supposed be complete in terms of capturing all architectural elements, it abstracts from several details in order to enable a discussion on key constituent elements that enable the service provision in software ecosystems [DS14].

2.2.3 Ecosystem Health

Health of software ecosystems refers to *the overall well-functioning of the ecosystems that enables them to endure and remain productive over time* [MH13, AS16, HSN$^+$15]. Health was first introduced by Lansiti and Levien [IL02, IL04a] as a major performance indicator in business ecosystems. According to the authors, health of business ecosystems is directly related to three attributes that are *robustness*, and the capability of *niche creation*. Robustness is the capability of an ecosystem to face threats and survive. Niche creation is the capacity of an ecosystem to exhibit creation of niches over time by supporting diversity and variety. On this basis, the majority of literature dedicated to software ecosystems has developed a similar understanding of ecosystem health as well [Man16, MH13].

In order to ensure an ecosystem is functioning well, a quality model is required that facilitates the assessment of health [MH13]. A strategy assessment model by Van den Berk et al. [VDBJL10] extend the definitions of the three main health attributes of business ecosystems, i.e., robustness, and niche creation, with the concepts specific to software ecosystems. For example, external threats are not only related to enterprise concerns such as competitive ecosystems, but also technical aspects such as malfunctioning third-party developments, which can be tackled with by community-building among the niche players and enhancing the knowledge sharing or defining entrance barriers. Other works [LYW$^+$19, Jan14] consider health in the context of open source software ecosystems. Metrics are introduced to measure the ecosystem health on the basis of quantitative data on how the ecosystems grew in the past, e.g., LOC added to the project or bugs fixed in a certain timespan.

Within the scope of this thesis, we use the quality model of ecosystem health introduced by Ben Hadj Salem Mhamdia [BHSM13]. The quality model is grounded on the work by Lansiti and Levien [IL04a]. Poductivity, robustness, and niche creation are discussed in the context of software ecosystems. In addition, further quality attributes of ecosystem health are identified, e.g., *sustainability*, *profitability*, *customer satisfaction*, *interoperability*, *modifiability*, *creativity*, and *cost-effectiveness*. Note that these are quality attribute related to the performance of software ecosystems that target the performance at an enterprise level rather than a technical level. Subsequently, the quality attributes, e.g., robustness and modifiability, concern the enterprise architecture of platform providers rather than the architecture of software systems.

2.3 Architectural Knowledge Management

Architectural knowledge refers to architectural design and design decisions as well as any other factor that affects an architectural design such as context [KLVV06]. Bosch and Jansen [JB05] view architecture as a set of architectural design decisions. Thereby, they recognize architectural design decisions as first class knowledge entities.

2.3.1 Architectural Design Decisions

"An architectural decision is a decision that affects the system architecture" [HAZ07]. In the last two decades, some work [Bos04, BCK03] raised attention to the importance of architectural design decisions without providing concrete information on their structure. Others [KLVV06, Kru04] suggested to document certain key attributes such as epitome, rationale, scope, state, etc. by proposing conceptual models and templates. While the task of documenting architectural design decisions by architects during the design of software ecosystems is not the focus of our work, we refer to this work to indicate the key parts of ecosystem-specific design decisions concerned by our work. With this respect, we mainly refer to epitome of an architectural design decision once we identify or discuss it.

Epitome is the core part of the decision (the decision itself), which is a short statement that is often used to refer to the decision. e.g., while listing the decisions or using them in diagrams. We discuss the possible rationale behind ecosystem-specific decisions when it is relevant. The rationale specifies why a decision should be taken into consideration. In addition, an architectural design decision has a scope, which can be universal or bound to certain limitations, e.g., when / where to apply. In our work, the decisions are universal except when we explicitly refer to a scope.

Relations between architectural design decisions are the focus of other work. Zimmermann et al. [ZKL$^+$09, Zim11] propose a formal graph-based approach to model architectural design decisions and their dependencies, in contrast to earlier work that focused on text templates. Using their approach, reusable decision models can be created and further used to verify integrity constraints and to organize the process of architectural decision-making.

2.3.2 Pattern-Centric Architectural Knowledge

"Pattern is an idea that has been useful in one practical context and will probably be useful in others" [Fow97, p. 8]. Effective documentation of architectural knowledge

can be enabled by the development and application of patterns [CJT+16]. Harrison et al. [HAZ07] consider architectural patterns and design decisions to be complementary to each other, i.e., using a pattern implies making a set of design decisions associated with the pattern. Patterns are solutions for recurring problems. Pattern-centric approaches exploit the benefits of patterns to enhance the documentation and application of design decisions. Initial pattern-centric approaches, e.g., *Gang of Four* [GHJ+95], concerned object-oriented software design, since then patterns related to many other areas have been introduced like patterns of architectural design decisions for enterprise architecture [Fow12].

A Pattern story [BHS07] is a narrative text that “captures a pattern sequence and specific design issues involved in constructing a concrete system or creating a particular design example”. Pattern stories are powerful means that have been used frequently within the community of software architecture to enhance knowledge communication. To develop pattern stories, we follow the knowledge of methodical pattern development in literature. There are different templates for the structure of pattern stories [Gro21, WF12]. Most of these templates agree on certain sections. In this thesis, we follow the knowledge of pattern development proposed by Buschmann et al. [BHS07, sec. 1.2] that Accordingly, a pattern comprises the following sections:

- ***Context***: It describes a situation in which one can apply a pattern.
- ***Problem***: A description of a difficulty to overcome by applying a pattern.
- ***Forces***: Forces clarify why the problem is difficult to solve. This includes to identify the risks when a pattern is not applied.
- ***Solution***: A description that specifies how to resolve the problem and its associated forces.
- ***Consequences***: Consequences are concerned with a guidance on how the solution resolves the forces.
- ***Known Uses***: Examples of a pattern that provide information on the application of the patterns and their typical application domains.
- ***Related Patterns***: The relations between a pattern and other patterns.
- ***Example***: An illustrative application of a pattern that demonstrates how the pattern is applied in a specific context.

2.3.3 Architecture Framework

Architecture framework is a term used by researchers and practitioners to refer to a set of practices that specify how to define and use the architecture of software systems. Throughout this thesis, we refer to the following definition of architecture framework provided by the international standard ISO/IEC/IEEE 42010: *"An architecture framework establishes a common practice for creating, interpreting, analyzing and using architecture descriptions within a particular domain of application or stakeholder community."* [ISO11]

Figure 2.1 shows the entities, which should be captured by architecture frameworks. An architecture framework is tightly linked to a community of stakeholders, who has certain interests in a system. The stakeholders' interests are referred as *concerns*. Furthermore, an architecture framework has certain viewpoints that frame the stakeholders' concerns by defining their scope. The viewpoints have at least a *model kind*, which is a metamodel representing elements of the architecture, their attributes, and relationships. Finally, correspondence is a relation within an architecture or between different architectures such as composition rules, refinement, consistency, dependency, or constraint. Correspondence rules are defined to track the fulfillment of such relations that are expected to exist in the architecture [ISO11].

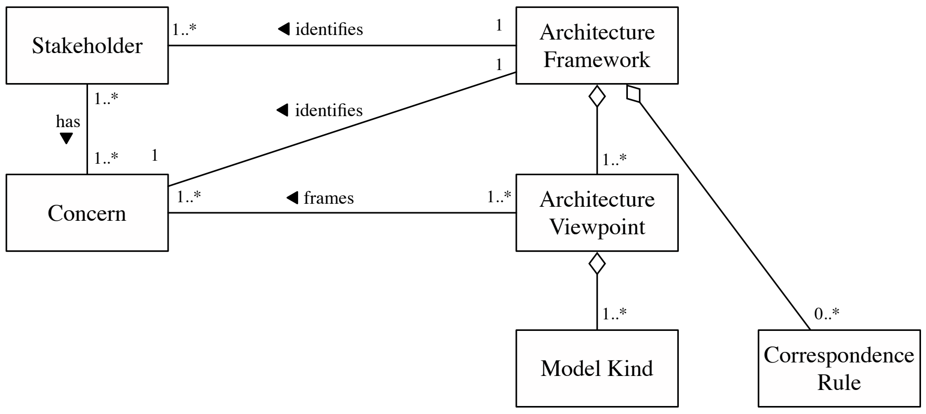


Fig. 2.1 Conceptual Model of Architecture Framework [ISO11]

2.4 Software Product Line Engineering

Software product line engineering (SPLE) [PBv05] is a paradigm to develop software using software platforms and mass customization by developing and applying product lines in order to manage the architectural complexity of a family of software-intensive systems [PBv05, p. 14]. The main objectives are to reduce production costs by enabling systematic reuse and improve the overall quality. SPLE comprises two main processes, i.e., *domain engineering* and *application engineering*. Domain engineering is the process of identifying reusable artifacts of a software platform in terms of commonality and the variability of a product line while application engineering is the process of deriving concrete applications from the product line developed during the domain engineering. [PBv05].

2.4.1 Application Domain

In the context of SPLE, an *application domain (a.k.a. domain) is a set of software systems that share certain characteristics.* Inside an application domain, domain knowledge is usually communicated by means of a common terminology among one or more communities of stakeholders [KCH+90]. In this thesis, we refer to the definition of an application domain mentioned above. In literature, similar definitions are provided for what an application domain is. For instance, in domain-specific language engineering [VBD+13], an application domain is a set of software with common characteristics that covers a body of knowledge about the real world. Furthermore, an application domain has a scope that defines the shared characteristics of the software systems inside that domain. In addition, domains can have hierarchical relationships with each other, i.e., domains with a narrower scope are a subset of the domains with a broader scope [VBD+13, chap. 3].

2.4.2 Domain Engineering

Domain engineering is the main process of SPLE to construct reusable artifacts, methods, and tools that address the problem of system development throughout a domain [KCH+90]. During domain engineering, the domain is analyzed and the complexity of the domain knowledge is handled by means of developing product lines. A *software product line* encompasses *common* and *variable* reusable artifacts. While the common artifacts are shared among all products of the product line, the variable artifacts are not common among all the products, but among some of them. Variabilities are described using *variation points* and *variants*. A variation point refers to the subject of a variability, i.e., a variable property/item. A variant represents the object of a variability, i.e., a particular instance of a variation point [PBv05, p. 62]. In general, reusable artifacts can be artifacts related to the different phases of software development, e.g., requirements engineering, design, and testing.

In the process of domain engineering, commonalities and variabilities of a product line are identified in terms of features [KCH+90, Bat05]. A *feature* is *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system"* [KCH+90]. In this context, a feature can be any function of the system that is visible to the end-user. Thereby, a feature is defined from an external viewpoint to the system rather than internal. Examples of internal characteristics are data-flow and state transitions.

The process of domain engineering encompasses five main activities, i.e., product management, domain requirements engineering, domain design, domain realization, and domain testing [PBv05].

The *product management* sub-process concerns the strategic approach to specify the scope and goal of the product line, which influences all other sub-process of domain engineering.

The goal of *domain requirements engineering* is to identify common and variable requirements of the product line. The requirements can be defined using natural languages or by using models. Variability models are a typical output of this activity.

Domain design, where a reference architecture for the family of software that belongs to the product line is defined. The reference architecture concerns a high-level structure that is common among all applications of the product line.

The goal of *domain realization* is to provide guidelines for the design and implementation of the reusable artifacts.

Finally, during *domain testing,* processes for validation of reusable artifacts are developed. Moreover, reusable test artifacts are provided that ensure early validation of requirements and designs.

2.5 Model-Driven Engineering

Model-driven engineering (MDE) is a software development methodology that promotes the development and usage of models during the software development. Models are created to capture certain aspects of the software systems relevant to a specific problem. Therefore, the models do not include every detail but the ones that are related to a certain abstraction.

2.5.1 Meta-Object Facility

Meta-object facility (MOF) [OMG16] is a standardization of MDE introduced by the object-management group (OMG) [G+13]. According to MOF, a system can be described in an infinite number of metamodeling layers while four meta layers are commonly used, i.e., M0–M3.

In Figure 2.2, we use an example to present the meta layers of the OMG. M0 refers to a layer where the real-world objects belong to. These objects are described using models in the M1 layer. Such models are aimed to represent the real-world objects by including the relevant aspects and hiding the details. In M1, a class namely *Person* is defined that has an attribute *name* of type *String*. The object in M0 is an instance of the class *Person*. It has a specific name that is *John Doe*.

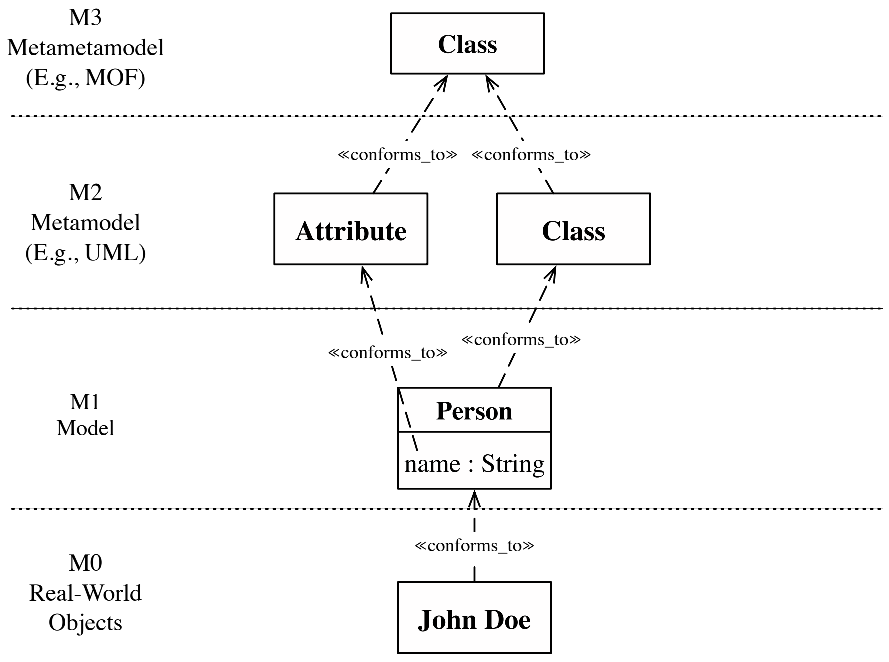


Fig. 2.2 Meta Layers Proposed by the OMG (Adapted from [VSB+13, p. 86])

Furthermore, the models in the M1 layer are further described using metamodels in the layer M2. Such metamodels include the rules according to which the models in M1 should be created. For example, different diagrams of the Unified Modeling Language (UML) [OMG17] conform to a the UML metamodel that specifies the diagram entities and the relations between them.

In Figure 2.2, the class in M1 is defined in M2 via the class *Class* while its name is an instance of an associated class namely *Attribute*. In M3, MOF resides that define how to express a metamodel in the layer M2. The models at the M3 layer are referred as metametamodel. MOF is self-describing. Thus, it can be used to define models at higher meta layers. Specifically, MOF specifies the concept of a *Class* that is used in Figure 2.2 to define the classes *Class* and *Attribute* in M2.

2.5.2 Model-Driven Architecture

Model-driven architecture (MDA) is a software design approach that follows the principles of MDE. It is proposed by OMG to systematize the process of software development using models.

MDA emphasizes on the separation of business and application logic from the underlaying technology by introducing several kinds of models at different abstraction levels for software systems. The different abstractions are expressed in form of models and can be later used as software specifications. The models in higher abstractions are transformed into models with more specific knowledge about the system. After all, code generators can be used to generate the source code from the models.

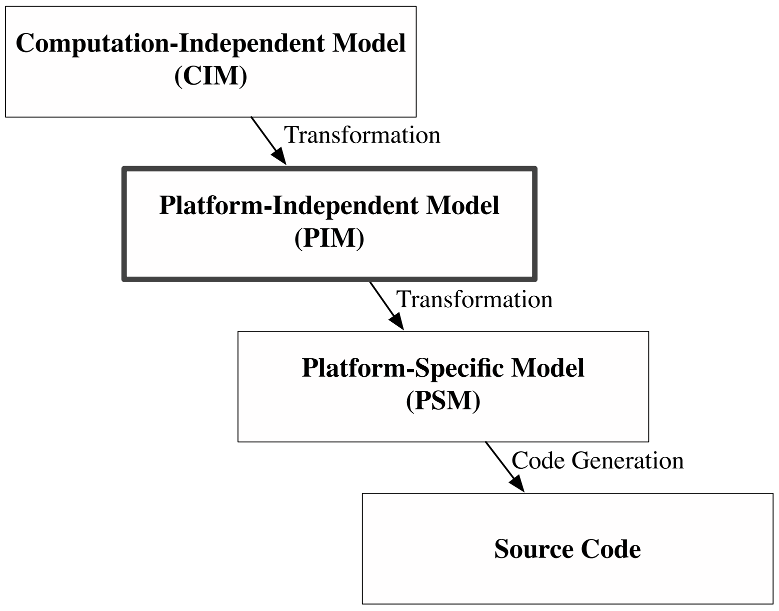

Computation-Independent Model (CIM)
Transformation
Platform-Independent Model (PIM)
Transformation
Platform-Specific Model (PSM)
Code Generation
Source Code

Fig. 2.3 Model-Driven Architecture (MDA)

In the context of MDA, a *computation-independent model (CIM)* expresses business logic of the software system such as roles, activities, organizational constraints, etc. This knowledge is independent of technical realization of the system.

A *platform-independent model (PIM)* is derived from the CIM and adds the knowledge regarding the technical realization of the software system. Nevertheless, the PIM should remain independent of a platform, i.e., the technology where the software system is going to function.

The information concerning the technical realization of the system on a specific platform is added to another model namely platform-specific model (PSM) that includes technology-specific knowledge.

According to the MDA principles, the PSM can be used to derive the implementation of the software system by generating code from it. For some reasons, the code might be partially automatically generated from the PSM and later be complemented manually. As motivated in Section 1.2, this thesis is mainly concerned with PIMs, i.e., *the domain-specific knowledge of platform-independent models of software ecosystems.* In the context of MDA, one can say the exiting architectural knowledge mainly concerns platform-specific models of ecosystem architecture.

Chapter 3

Requirements and Related Work

The goal of this thesis is to improve architectural decision-making in the lack of knowledge, by providing a solution that systematizes the design of healthy software ecosystems. In Section 3.1, we refer to the challenges defined in Section 1.2. From these challenges, we derive the requirements that a solution needs to fulfill. We evaluate state of the art with respect to the requirements and discuss why related work does not provide a solution for the problem in Section 3.2. In Section 3.3, we condense this discussion to a problem statement.

3.1 Requirements

In this section, we introduce requirements of a solution approach that aims at providing systematic architectural knowledge to design software ecosystems. We derive the requirements from the challenges C1-C7 discussed in Section 1.2. In connection to the challenges, the requirements are of two main types, i.e., requirements of an architectural knowledge base and requirements of methodical architectural guidance. In the following, R1-R9 refer to the functional requirements of a solution. In addition, such a solution should satisfy certain quality attributes that can facilitate the future applicability of the solution in practice.

Requirements of An Architectural Knowledge Base

This set of requirements characterizes a knowledge base provided by a solution approach concerning *what* design decisions are included in the ecosystem architecture.

R1: An Integrated View of Interdisciplinary Key Design Decisions

The solution should provide knowledge on the most influential design decisions that form the structure of software ecosystems. Notably, this knowledge should facilitate an integrated view to the organizational, business, and technical aspects by capturing architectural design decisions from different disciplines. In particular, the platform providers should be informed about primary features that are key to the ecosystem while the usage of the features does not depend on a specific application domain wherein an ecosystem is created.

R2: Architectural Variation Points and Variants across Application Domains

To successfully deal with architectural variabilities, platform providers should be informed about the knowledge of variable design decisions that concern the architecture of software ecosystems. To this end, first, architectural variation points of software ecosystems should be identified, i.e., the decision points, where the ecosystem architecture varies between the ecosystems in different application domains. In the second step, the knowledge of architectural variabilities should enable platform providers to choose among variable design decisions and to create customized ecosystem designs. Therefore, the solution concepts should provide knowledge on concrete variants that the variation points might have.

R3: Identification of Decision Interdependencies

The solution concept should specify the dependencies between the architectural design decisions. Here, it is specially relevant to declare mutual impacts between the design decisions or the one-way relations, where only a decision is affected by another one. Specifically, the solution concept should reveal what decisions need to be considered in the architecture once other decisions are made. Using this knowledge, platform providers can identify the decisions that enforced/excluded by currently taken decisions or the ones that are prerequisites to the existing decisions.

Requirements of Methodical Architectural Guidance

In the following, we present requirements of the methodical part of a solution approach concerning *how* to design the ecosystem architecture.

R4: Guidance on the Linkage between Design Decisions and Business Objectives

To assist platform providers in addressing business objectives, the solution concept should provide knowledge on the linkage between the ecosystem architecture and business objectives so that the platform providers can address those objectives while designing software ecosystems. Essentially, this knowledge should give insight into the most relevant business objectives that can be pursued by establishing ecosystems around software platforms. For this purpose, the linkage between concrete design decisions and the business objectives needs to be identified. This raises another requirement that concerns suitable design processes to apply this knowledge (R7).

R5: Ecosystem Modeling Support

A suitable design support for software ecosystems should provide platform providers with a knowledge base and enable the design of this knowledge in models of ecosystems. Therefore, the design support should follow the requirements of a knowledge base (R1-R4) by enabling platform providers to capture the key and variable design decisions and the decision interdependencies. The resulting models should facilitate an integrated view of the main architectural aspects. Moreover, using the modeling approach, platform providers should become able to further tailor the architecture.

R6: Guidance on the Linkage between Ecosystem Health and Design Decisions

A solution concept needs to clarify the relationship between ecosystem health and architectural design decisions. This requirement is two fold: Firstly, this knowledge should enable platform providers to address ecosystem health during the design by adapting suitable architectural decision-making. Specifically, The literature concerning quality attributes of ecosystem health should be taken into account and the linkage between those attributes and architectural design decisions of software ecosystems needs to be specified. Secondly, platform providers should be able to check the linkage between ecosystem health and architectural design decisions in the ecosystems that are already designed or when the ecosystem architecture changes. Thus, the solution should comprise analysis techniques that enable platform providers to assess suitability of the ecosystem architecture with respect to the quality attributes of ecosystem health. In particular, concrete implications about the design decisions made in the architecture should be provided by the analysis technique. In addition, platform providers should be able to compare alternative architectures on the basis of variabilities. The analysis technique should draw attention to the points in the architecture, where the quality can be improved. With this respect, platform providers should receive concrete guidelines on improvement actions.

R7: Design Process

While the requirements R4-R6 characterize the methodical knowledge that should be provided by a solution concept, they do not address how this knowledge should be applied in practice. Therefore, we add another requirement that demands stepwise a design process, which clarifies how platform providers can specify an ecosystem architecture by using the solution. In particular, such a process should capture the design and analysis activities described as parts of R4-R6.

R8: Tool Support

The solution approach should be supported by suitable tooling. The goal is to reduce the overall effort of designing and analyzing ecosystem architectures. A tool support should provide platform providers with seamless integration of design and analysis activities in one unified modeling environment. Similar to the modeling support, the tool support should satisfy a reasonable degree of ***maintainability*** when the source code needs to be changed. For instance, what can be the efforts to implement a new modeling element in the tool? Specially relevant for the tool support is to ensure its ***portability*** that guarantees that the tool will function in different working environments.

R9: Formality

The solution approach should have a formal foundation, i.e., a precise description for parts of the solution that can be processed by a computer system. The goal of such a precise foundation is to enable the applicability of the solution approach in models. For instance, the linkage between ecosystem health and design decisions (R6) should be automatically calculated using models of the ecosystem architecture.

The first quality concern for future applicability of the solution is ***functional suitability*** [ISO12], which refers to the way that the solution meets the stated requirements (R1-R9). The goal is to assess whether the solution contributes to the final goal, which is to improve architectural decision-making. To this end, the subcharacteristic of functional suitability should be considered. The framework captures a correct understanding of ecosystem-specific design decisions. Furthermore, such an understanding should be *complete* in terms of covering different architectural aspects of software ecosystems, i.e., social & organizational, business & management, and application & infrastructure. Another example is whether the framework provides an *appropriate abstraction* to design software ecosystems without being too general or too specific.

A further concern is ***maintainability*** that refers to the effort, which is required to update the knowledge base when the state of the art of the body of knowledge changes, e.g., when new architectural characteristics emerge in software ecosystems.

Furthermore, usability of the knowledge provided by a solution matters as it affects productivity of architects' work. Specifically, ***learnability*** of the knowledge base, methodical knowledge, and tool support should be considered. It refers to the effort that is required to get familiar with the architectural knowledge provided by the solution and work with the tool. Another relevant quality attribute is ***efficiency*** that refers to the time associated with designing and analyzing software ecosystems.

In summary, a solution for the research problem of this thesis needs to fulfill the requirements (R1-R9) in order to succeed in addressing the key architectural challenges of designing ecosystem architecture. In the rest of this chapter, we discuss related work with respect to the requirements.

3.2 Related Work

In this section, we present the state-of-the-art of architectural knowledge to design software ecosystems. With this respect, we consider two main groups of related work that are concerned with the challenges C1-C7, i.e., *work providing an architectural knowledge base* and *work providing methodical architectural guidance*. We discuss suitability of the related work with respect to the requirements R1-R9.

3.2.1 An Architectural Knowledge Base

In this section, we provide an overview of related works that consider structural knowledge of software ecosystems. First, the works concerning key design decisions are discussed. This mainly includes reference architectures and generic models of software ecosystems. Afterwards, the works that refer to architectural variabilities of software ecosystems are presented.

Reference Architectures & Generic Models

A common understanding of ecosystem architecture is the first topic that has drawn the attention of research communities. Thereby, generic architectural models and reference architectures have been introduced. We evaluate these works against the requirements of the architectural knowledge base (R1-R3). Table 3.1 provides an overview of the related works and the fulfillment of the requirements by these works. In the following, we discuss them.

Table 3.1 Evaluation of Architectural Knowledge Base Approaches

			Criteria		
			R1: An Integrated View of Interdisciplinary Key Design Decisions	R2: Architectural Variation Points and Variant across Application Domains	R3: Identification of Decision Interdependencies
Approaches	Reference Architectures & Generic Models	[EB14]	O	—	O
		[APA14]	+	—	O
		[SO11]	+	—	—
		[GAM+17]	O	—	O
		[ZJD14]	O	—	O
		[HJZ12]	+	—	O
	Variability Models & Techniques	[BPT+14]	—	+	O
		[JBSL12]	O	+	O
		[GAT13]	O	O	O
		[REV19]	O	+	O

Requirement is fulfilled (+) / not fulfilled (—) / partially fulfilled (O)

Eklund and Bosch [EB14] propose a reference architecture for an open software ecosystem in the context of embedded systems. The reference architecture includes three categories of key design decisions and their dependencies, i.e., fundamental decisions related to the development of the platform and third-party applications, ecosystem facilitation decisions to achieve required mechanisms, and platform implementation decisions to realize and deploy the ecosystem. While the reference architecture captures technical design decisions, business decisions are overlooked. Furthermore, architectural characteristics of ecosystems in other application domains, and accordingly architectural variabilities, are out of scope of this work.

Similar to the previously described approach, Axelsson et al. [APA14] present the main architectural characteristics of Federated Embedded Systems (FES). The work points out the inclusion of plug-in software and external communication in embedded systems. However, in contrary to the previous approach, this work concerns the business and organizational characteristics such as the actors, possible business models, effects on product development processes in such ecosystems. To obtain these characteristics, the authors carry out an exploratory study on the basis of interviews with 15 domain experts. In addition, architectural variability is considered out of scope.

Schlauderer and Overhage [SO11] provides a conceptual model for design decisions of IT service markets from an economics perspective. The authors emphasize the importance of strategic decisions taken by market providers in market success. They first identify the deficiencies of existing marketplaces by examining StrikeIron, Salesforce AppExchange, and Google Play. Strategic decisions of successful markets from the economic literature are used to address those deficiencies. For example, market providers can produce certification for high quality service, which shows the consumers that a service conforms to market standards. This work generalizes about several markets in different application domains, however, it remain unclear which part of the decisions could be variable or differently realized in the ecosystem architecture. Furthermore, inter-dependencies between the decisions are not covered.

Gomez et al. [GAM+17] present a reference architecture for mobile App stores that exploit crowdsourced information related to Apps, devices, and users. The authors propose an architectural solution to collect information from the user reviews and App logs, and further use it to improve the overall service quality. While the results are discussed in the context of mobile App stores, the solution provided by the reference architecture can be reused in other application domains to address a similar problem. However, the reference architecture can be complemented in several ways: Firstly, it is specific to a certain problem addressed by this paper. Secondly, although it includes key architectural components and their relations, the relation to the rest of App store architecture and to the business aspects remain open. Furthermore, variabilities that might be applied to the reference architecture in different domains are overlooked.

Zuiderwijk et al. [ZJD14] consolidate essential elements of open data ecosystems. There, architectural elements and critical activities are identified at three levels, i.e., producer, data, and tools. In particular, open data ecosystems need to facilitate releasing and publishing, finding and viewing, interpreting, and discussing the data. The goal is to enhance data-sharing between data producers and users. The scope of this work is limited to the governance of open data. Comparing with our work, our work provides insight into a wide range of architectural elements, including knowledge sharing. Moreover, architectural variability is not considered. Consequently, the required generality is not attained. Furthermore, only the architectural decisions and decision dependencies regarding technical architecture are identified.

Harman et al. [HJZ12] provide empirical evidence on the existence and relation of rating, ranking, and pricing features in mobile App stores. The work is narrowed to the mobile App domain and limited features. Therefore, the required generality for software ecosystems in other domains is missing. Accordingly, variable architectural design decisions and their dependencies are not discussed.

Variability Models & Techniques

Various studies [Man16, MH13, Bos10] consider high variability as the major challenge to generalize the architectural knowledge of software ecosystems. In recent years, several works are dedicated to the architectural variabilities, i.e., in ways that the ecosystem architectures differ. In the following, we evaluate the suitability of these works with respect to the requirements of methodical architectural guidance (R4-R9).

Berger et al. [BPT$^+$14] propose a framework for variability mechanisms in software ecosystems, which is the result of analyzing five open-source platforms from the Linux kernel to Eclipse and Google Android. Specifically, the framework captures representations of variabilities, design decisions, encapsulation, interaction, and dependencies in terms of variation points and variants. For example, a variation point is *encapsulation interface mechanisms*, which differs from *C header* files in the Linux kernel to *Java interfaces* in the Eclipse platform. Comparing to our work, this work mainly captures technical aspects while key architectural design decisions and business aspects are marginally discussed. Furthermore, our work is the result of a study of both commercial and open-source ecosystems. In general, our solution is built on this work while extending it by methodical architectural support.

On opening software platforms, Jansen et al. [JBSL12] consider the spectrum of business model openness including the variabilities. The concept of openness consists of strategic, tactical, and operational views, which are derived from the platform provider's business model. While this work focuses on social and organizational characteristics, it excludes a specification of software and infrastructure elements and how the degree of openness might impact the inclusion and exclusion of such elements in the architecture.

Galster et al. [GAT13] presents a framework comprising key constraints for the process of designing reference architectures for variability-intensive service-oriented systems. The goal is to aid designing competent reference architectures that eliminate the need to design these systems from scratch while using them in different environments. Comparing to our work, the constraints here pertain to the strategic decisions concerning interoperability and integration of enterprise architecture at both organizational and technical levels. Yet, commonalities and variabilities are not explicitly identified in this context. In addition, business aspects are out of the scope of this work. Part of these constraints, e.g., interactions with external parties, are extended by our work, wherein our solution specifies concrete variabilities for generic interfaces between the parties. Other parts, e.g., consideration of legacy architecture, complement the concepts introduced by our work.

REVaMP[2] [REV19] was a European project, dedicated to the conceptualization and implementation of techniques for the life cycles of Software-Intensive Systems and Services (SIS) product lines. The project aimed at facilitating the development of mass-customized software in any arbitrary application domain. Variability extraction and visualization techniques are especially in focus. Similar to our work, this work exploits model-based engineering approaches to manifest system commonalities and variabilities using models (e.g., variability models and metamodels). However, the business design decisions and their dependencies to other decisions are not covered. Furthermore, techniques for variability management are extensively considered. Our work provides domain knowledge that assists with improved architectural decision-making.

As discussed above, existing works considering the architectural commonalities and variabilities do not support all the requirements of a solution for systematic knowledge base for software ecosystems. We notice that the existing works hardly capture both commonalities and variabilities and their interdependencies at the same time. Furthermore, business aspects are rarely addressed.

3.2.2 Methodical Architectural Guidance

In recent years, the works related to the architectural knowledge of software ecosystems shifted more to the direction of providing methodical support, design processes, and modeling techniques. In the following, we assess these works based on the requirements of methodical knowledge (R4-R9). Table 3.2 provides an overview of this assessment.

Woods and Bashroush [WB15] propose a visual architectural description language (ADL) to specify architectural descriptions for large and complex information systems. The ADL was developed for industrial experiences and includes tool support that has been required by the project. The visual notation is in principle conventions that are agreed in the project without having any formal basis. Moreover, the approach is designed purely for creating architectural descriptions whereas the linkage to business objectives and architectural analysis are not considered.

Yu et al. present a design approach [YD11, SDY15] using an i* goal-oriented social modeling technique, accompanied by a process for modeling collaboration environment inside software ecosystems. The resulting models are high-level while exposing the relationships between actors, tasks handled by the actors, and their goals. On the basis of the modeling approach, the authors propose an analysis technique [SY17a] for specifying and assessing non-functional requirements in software ecosystems. In particular,

Table 3.2 Evaluation of Methodical Architectural Guidance Approaches

		Criteria					
		R4: Guidance on the Linkage between Design Decisions and Business Objectives	R5: Ecosystem Modeling Support	R6: Guidance on the Linkage between Ecosystem Health and Design Decisions	R7: Design Process	R8: Tool Support	R9: Formality
Approaches	[WB15]	—	+	—	—	O	—
	[YD11, SDY15, SY17a]	+	+	O	+	—	+
	[eCHKM14]	+	+	—	+	—	O
	[MHT16]	—	—	—	+	—	—
	[BB14]	—	—	—	+	—	—

Requirement is fulfilled (+) / not fulfilled (—) / partially fulfilled (O)

the analysis technique aims at assessing openness requirements and calculating their trade-offs. This includes the assessment of business decisions that are originated from the openness strategies, and the subsequent technical quality requirements imposed on the architecture. Two case studies are used to show the application of the approach. Although our work and this work have some common motivations, they have different focuses. As the choice of social modeling technique in this work reveals, the resulting models are quite high-level whereas they only capture the main elements of collaborations, i.e, actors, tasks, and goals. However, our work concerns a more granular view on ecosystems that includes the actors and their interactions with software and infrastructure elements, and the relations between the elements in the architecture. Moreover, the analysis technique in this work assists architects to find the trade-off between the requirements based on the weight and importance that are given to the requirements by the platform provider. However, in our work, platform providers are provided with the knowledge of well-established ecosystems in practice. Furthermore, the lack of the design process for the analysis phase and tool support hamper the applicability of this work. Despite formality being supported by i* modeling, no tool support is provided.

Christensen et al. [CHKM14] conceptualizes the design and analysis of ecosystem architecture in the context of telemedicine. The ecosystem is described in terms of organizational, business, and software aspects. Business model canvas is used to present business strategies. In particular, business objectives are included in terms of value proposition, an example being *cheaper IT development*. Furthermore, the authors propose to use UML Deployment Diagrams to describe software components and their relations. We consider that formality is not fulfilled in case of organizational and business aspects as both of these aspects are described using text in natural languages. Besides, the relations between the text and the software architecture is not clarified. Moreover, no kind of tool support is provided by this work. In our work, the architectural knowledge is the result of a broad study of existing ecosystems from diverse application domains. As a result, our solution concept comprises fine-grained business aspects, e.g., openness policies, that are described as a part of a domain model.

Manikas et a.l [MHT16] presents a high-level process for the design of software ecosystems that distinguishes among three different ways that software ecosystems are created. Part of the process is concerned with an actor-rooted approach, i.e., the ecosystem is created around a strong actor consortium. In addition, a business-rooted approach is taken into account, when the ecosystem is built around a strong business. An infrastructure-rooted approach refers to the establishment of the ecosystem around a technological infrastructure. While this classification is helpful to handle design activities related to each situation, it makes the application of the process less flexible because software ecosystems are often created as a result of an intertwined combination of this classification.

Bosch and Bosch-Sijtsema [BB14] propose ESAO (Ecosystem, Strategy, Architecture, and Organizing) for analysis of software ecosystems. ESAO is a conceptual model comprising six dimensions, with the goal to raise awareness of ecosystem-specific design decisions and to describe them aligned with the (internal) enterprise architecture. The alignment is achieved when the organizations that are involved in an ecosystem describe their strategy, architecture, and organization, once at the enterprise level and once at the ecosystem level, i.e., *Internal Company / Ecosystem Strategy*, *Internal / Ecosystem Architecture*, *Internal / Ecosystem Organizing*. A major shortcoming is the missing linkage to business objectives. In addition, the architectural description created by ESAO are described in natural languages, which can lead to ambiguity. Thus, formality is not fulfilled. Furthermore, there are no processes, tool support, or modeling language to describe the architecture of ecosystems.

To summarize, existing approaches that aim at providing methodical support do not fully satisfy the provision of the modeling approach, architectural analysis, and design process. Most importantly, these approaches have major limitations in providing a solid formal foundation.

3.3 Overall Problem Statement

The assessment of the related work in Section 3.2 showed that there is no single approach that fulfills the requirements R1-R9. On this basis, Table 3.3 summarizes how the related work fulfills the requirements. Regarding R1, a valuable amount of research considers the key design decisions affecting the ecosystem architecture while their focus is either on technical or non-technical aspects. Our solution approach should be built on this research to obtain an integrated view of the technical and non-technical aspects.

The requirement R2 is partially fulfilled since the existing research mainly performed in the area of open source software platforms. Our solution should consider whether this research can be generalized for commercial ecosystems and extend this knowledge by new research concerning commercial ecosystems. Decision interdependencies (R3) were barely a central research focus by the related work. This topic is mainly discussed as byproduct research besides other research results.

Concerning R4, the related work proposes analysis and trade-off mechanisms that can be used to assess goals in general. An interesting open question is which business objectives can be suitably addressed by creating software ecosystems. In the area of ecosystem modeling (R5), the most relevant work to this thesis was actor modeling using a visual notation. However, the notation proposed was some agreements among the project members, without using metamodels or precise definitions. Other work considers ecosystems modeled using the UML notation in the M1 meta layer. Our solution should benefit from this work while proposing concepts using metamodels at the M2 layer.

Regarding R6, the health of open source projects has been the focus that is mainly related to how the source code is extended by developers. Our solution should firstly concern ecosystem health that is not necessarily specific to a certain group of ecosystems. Secondly, this should concern the managerial level rather than the code.

On one hand, a major part of the related work proposing design processes focuses on certain design activities within the scope of their research (R7). On the other hand, high-level design processes have been introduced. The level of generality can hinder the application of such work . To the best of our knowledge, R8 is still open. Existing work barely considered tooling that can be used out of the scope of the projects.

R9 is partially fulfilled because, until now, mainly the usage of natural languages in textual templates proposed to describe key architectural aspects. Formal methods in terms of graph-based analysis of openness policies are introduced. To address the previous requirements, the domain knowledge of software ecosystems should complement such methods.

Table 3.3 Summary of Requirement Fulfillment (R1-R9) by Related Work

Requirement	Description
R1: An Integrated View of Interdisciplinary Key Design Decisions	Partially Fulfilled - Polarized discussions of technical and non-technical aspects, with special focus on technical decisions of ecosystems around embedded system platforms, business decisions for market success, and rating and ranking in mobile App stores
R2: Architectural Variation Points and Variant across Application Domains	Partially Fulfilled - Variability models for open source software platforms proposed with focus on the Linux and Eclipse platforms
R3: Identification of Decision Interdependencies	Barely Fulfilled - Decision interdependencies implicitly discussed in terms of byproduct research results
R4: Guidance on the Linkage between Design Decisions and Business Objectives	Barely Fulfilled - Trade-off analysis mechanisms proposed, however, the domain knowledge of software ecosystems poorly addressed
R5: Ecosystem Modeling Support	Partially Fulfilled - Actor modeling proposed, without a precise specification. Modeling in M1 using the UML notation presented
R6: Guidance on the Linkage between Ecosystem Health and Design Decisions	Barely Fulfilled - Research mainly dedicated to the health of open source projects. Health at the managerial level hardly addressed
R7: Design Process	Partially Fulfilled - Processes with a limited focus on certain design activities introduced
R8: Tool Support	Requirement Open - Tool support limited to certain projects presented
R9: Formality	Partially Fulfilled - Graph-based analysis of openness policies discussed, however, relation to the domain knowledge of software ecosystems poorly addressed

In summary, two common drawbacks of the related work with respect to the main thesis's objective are a) being too general for the domain of software ecosystems or b) missing the generalization. The first drawback is related to the work considering the enterprise architecture modeling. There is *a lack of ecosystem-specific concepts* in this group of work that could be used to directly address ecosystem concepts. The second drawback related to the work is *being limited to a certain group of ecosystems.*

Despite some studies concerning ecosystem health in literature, *there is still a lack of knowledge on the relations between ecosystem health and the ecosystem architecture.* An additional challenge is the lack of formality to specify the ecosystem architecture. The current lack of tool support to specify the architecture and to utilize such a specification for automated analysis underlines the lack of a precise foundation as well.

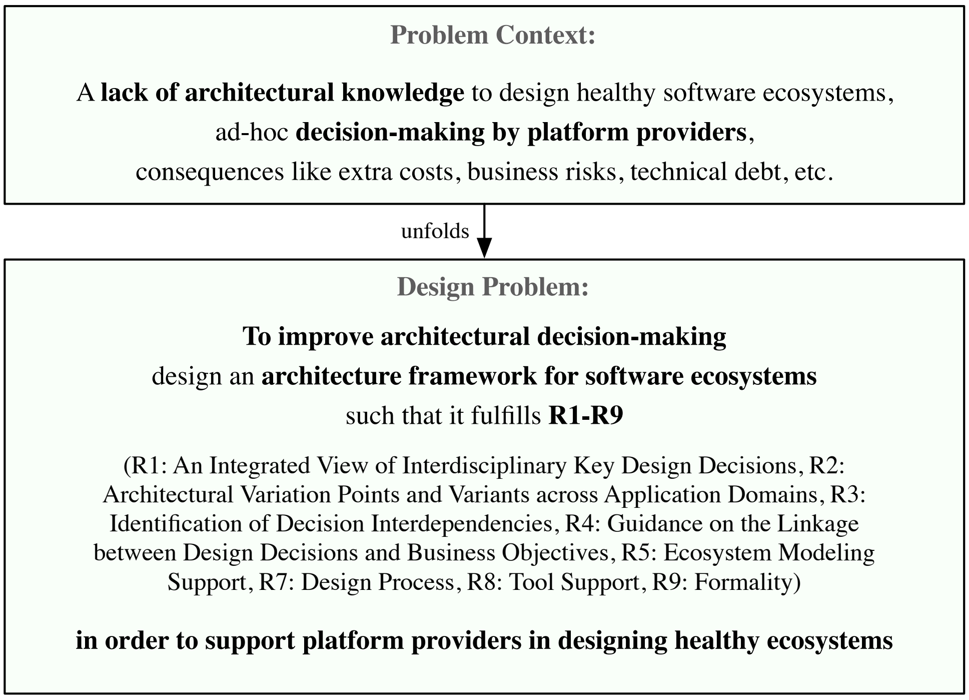
Chapter 4

Overall Research Approach of Developing the SecoArc Architecture Framework

In the previous chapters, we discussed the lack of architectural knowledge that could facilitate designing software ecosystems. In this chapter, we introduce the overall idea of the thesis's solution, which is an architecture framework for software ecosystems called *SecoArc*. Section 4.1 presents our rationale for developing the architecture framework and why this is a design problem. Section 4.2 elaborates on the procedure of developing the framework and the main research questions that are answered in this procedure. Section 4.3 provides an overview on the relations between the thesis's chapters and the research questions.

4.1 Design Problem

The lack of architectural knowledge and complexity of designing software ecosystems result in a situation, where platform providers have to rely on ad-hoc decision-making and face the consequences discussed in Section 1.2. In this thesis, we develop an architecture framework for software ecosystems aiming at making systematic architectural knowledge available for future use. We call the architecture framework *SecoArc* since we develop it for *architecting software ecosystems*. In general, the term *architecture framework* is used to refer to a set of guidelines that provide a common basis for the development of architecture descriptions within a particular application domain or stakeholder community [ISO11]. The SecoArc framework is meant for the domain


Problem Context:

A **lack of architectural knowledge** to design healthy software ecosystems, ad-hoc **decision-making by platform providers**, consequences like extra costs, business risks, technical debt, etc.

Design Problem:

To improve architectural decision-making design an **architecture framework for software ecosystems** such that it fulfills **R1-R9**

(R1: An Integrated View of Interdisciplinary Key Design Decisions, R2: Architectural Variation Points and Variants across Application Domains, R3: Identification of Decision Interdependencies, R4: Guidance on the Linkage between Design Decisions and Business Objectives, R5: Ecosystem Modeling Support, R7: Design Process, R8: Tool Support, R9: Formality)

in order to support platform providers in designing healthy ecosystems

Fig. 4.1 Thesis's Design Problem

of software ecosystems. In the context of this thesis, the most relevant stakeholder community is current and future platform providers because platform providers are the keystones and main decision-makers, who are responsible for designing and regulating the ecosystems.

According to ISO/IEC/IEEE 42010 [ISO11], an architecture framework is formed on the basis of five main elements that are *stakeholders*, *concerns*, *viewpoints*, *model kind*, and *correspondence rules* (cf. Section 2.3.3). Platform providers are the key stakeholders for the SecoArc framework as they are the main decision-makers. Furthermore, based on the main challenges of architectural decision-making discussed in Section 1.2, we described the main platform providers' concerns in terms of requirements (R1-R9). As we argued in Section 3.1, a solution should provide systematic knowledge regarding the structure of software ecosystems. This determines that the SecoArc framework should have a structural viewpoint to the architecture. In addition, a viewpoint should have at least a *model kind*, which should be a metamodel representing architectural elements of software ecosystems, and *correspondence rules* that should specify relations regarding the ecosystem architecture. The development of model kinds and correspondence rules constitute a core part of systematic knowledge provided by the SecoArc framework that we outline in Section 1.3.

Development of the SecoArc framework is a design problem that needs to address platform providers' needs in the context of the thesis's problem. A design problem is a kind of research problem that aims at designing an artifact so that it can help achieve certain goals [Wie14]. Thus, we use a *design science methodology* introduced by Wieringa [Wie14] for this purpose. Figure 4.1 describes the thesis's design problem. A *problem context* is a situation when certain stakeholders have to deal with unwanted consequences of a problem. The problem context unfolds a design problem. A *design problem* initiates the design of an artifact that can help the stakeholders to achieve their goals in that problem context. Earlier we described the thesis's problem context that includes the main challenges of architectural decision-making while designing software ecosystems. This problem context unfolds a design problem, which is to develop an architecture framework for software ecosystems to improve architectural decision-making such that it fulfills the requirements (R1-R9) in order to support platform providers in designing healthy ecosystems. In the next section, we present the overall development of the SecoArc framework.

4.2 Solution Approach

We develop the SecoArc framework by using a design science methodology proposed by Wieringa [Wie09]. The methodology introduces a spiral course of actions to answer inter-related and nested research questions that all contribute to the design of an artifact. The actions taken to answer a question are considered as a *cycle* (also known as an *iteration*). The design science methodology distinguishes between two kinds of research questions (RQs), i.e., technical research problems and knowledge questions. A *technical research problem* asks for an improvement in real-world. More importantly, it is related to the goals of certain stakeholders. Furthermore, it is answered through a so-called design cycle. A *design cycle (DC)* comprises three activities, i.e., problem investigation, designing a treatment, and validation. In addition, a *knowledge question* does not call for any change, but instead, it asks for an understanding of the world as it is. An *empirical cycle (EC)* is performed to answer a knowledge question, which includes the investigation of a research question, research design, validation, and execution of a solution by performing research activities that are specified in a research setup [Wie09].

Figure 4.2 shows eight main iterations that are performed during the development of the SecoArc framework. Results of the inner iterations are used in the outer ones. Furthermore, by performing each iteration, results of the previous iterations are re-

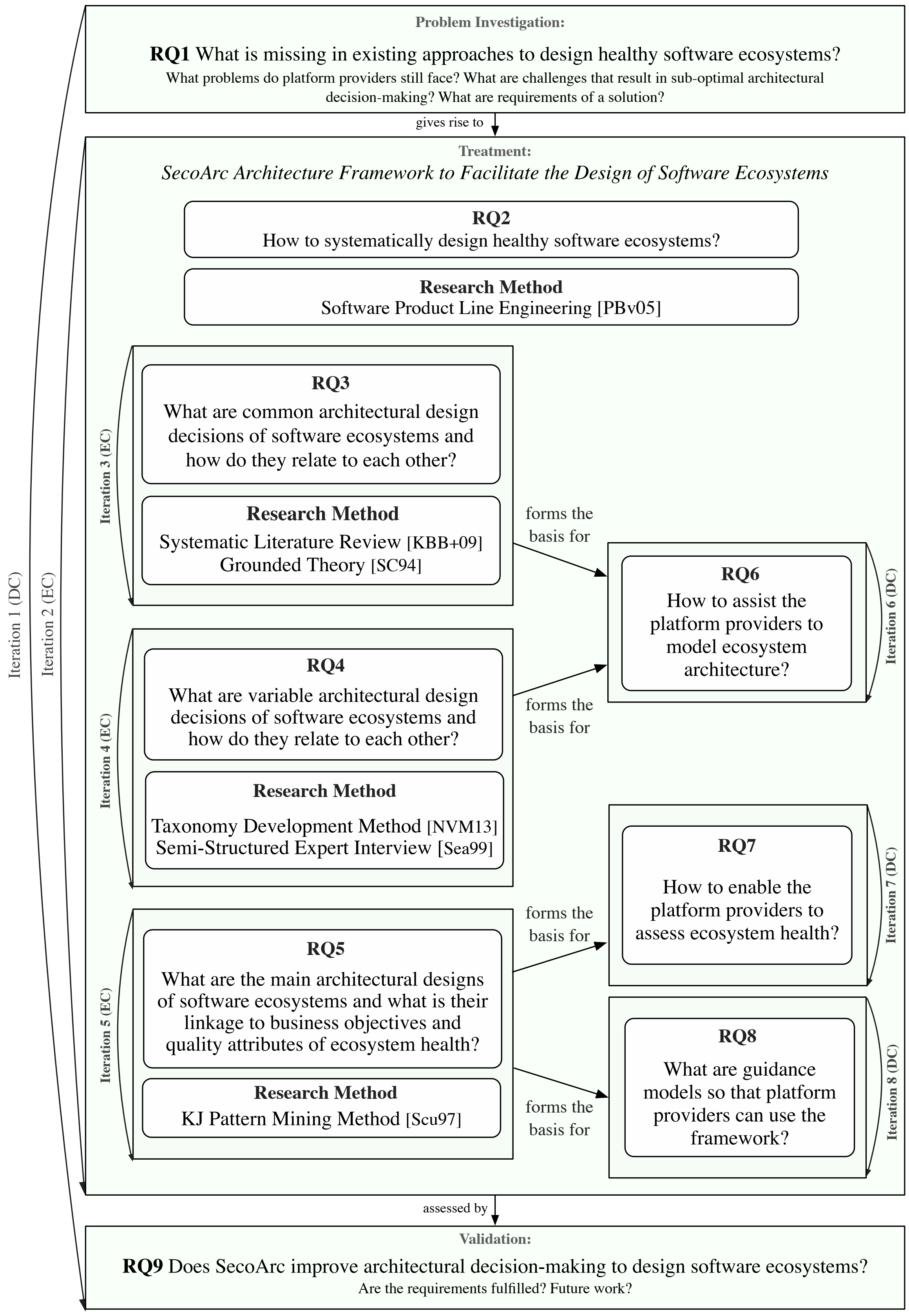

Problem Investigation:
RQ1 What is missing in existing approaches to design healthy software ecosystems?
What problems do platform providers still face? What are challenges that result in sub-optimal architectural decision-making? What are requirements of a solution?
gives rise to
Treatment:
SecoArc Architecture Framework to Facilitate the Design of Software Ecosystems
RQ2
How to systematically design healthy software ecosystems?
Research Method
Software Product Line Engineering [PBv05]
RQ3
What are common architectural design decisions of software ecosystems and how do they relate to each other?
Research Method
Systematic Literature Review [KBB+09]
Grounded Theory [SC94]
Iteration 3 (EC)
RQ4
What are variable architectural design decisions of software ecosystems and how do they relate to each other?
Research Method
Taxonomy Development Method [NVM13]
Semi-Structured Expert Interview [Sea99]
Iteration 4 (EC)
RQ5
What are the main architectural designs of software ecosystems and what is their linkage to business objectives and quality attributes of ecosystem health?
Research Method
KJ Pattern Mining Method [Scu97]
Iteration 5 (EC)
forms the basis for
forms the basis for
forms the basis for
forms the basis for
RQ6
How to assist the platform providers to model ecosystem architecture?
Iteration 6 (DC)
RQ7
How to enable the platform providers to assess ecosystem health?
Iteration 7 (DC)
RQ8
What are guidance models so that platform providers can use the framework?
Iteration 8 (DC)
Iteration 1 (DC)
Iteration 2 (EC)
assessed by
Validation:
RQ9 Does SecoArc improve architectural decision-making to design software ecosystems?
Are the requirements fulfilled? Future work?

Fig. 4.2 Thesis's Design Science Approach

considered if the results can be improved with the knowledge that is newly identified in the further iteration.

As shown in Figure 4.2, Iteration 1 represents the thesis as a whole. It is a design cycle since the development of the SecoArc framework is a design problem as discussed in Section 4.1. In this iteration, three key questions (RQ1–RQ3) are directly considered that concern the *problem investigation*, *treatment*, and *validation.* The treatment is the SecoArc framework that includes further nested iterations (Iteration 2 – Iteration 9). For explanatory reasons, Figure 4.2 depicts the most important parts of our design science approach. In the empirical cycles, the research questions and research methods are referred to whereas the validation and execution approaches are not mentioned. In the design cycles, the research questions are mentioned. The treatments are developed based on our results from the empirical cycles. Treatment validation in the design cycles is performed as part of the validation of the SecoArc framework. Furthermore, the processes of investigating RQ1 and RQ9 include further iterations, which are omitted from the figure for enhanced readability. In the following, we elaborate on Iteration 1 – Iteration 9. Further details related to the development of the SecoArc framework can be found in the corresponding chapters of the thesis.

Iteration 1: Development of The SecoArc Framework

The first iteration is a design cycle that is related to the thesis's design problem introduced in Section 4.1. The goal of the *problem investigation* is to investigate the problem context. Thereby, we should investigate whether existing approaches provide the required systematic knowledge to design healthy software ecosystems. Specifically, the question is:

RQ1 What is missing in existing approaches to design healthy software ecosystems?

We investigated the problem context of our research problem in Section 1.2, where the challenges of sub-optimal architectural decision-making entailed to platform providers were discussed (C1-C7). From these challenges, we derived the requirements of a solution concept (R1-R9) in Section 3.1. Afterward, we discussed the existing approaches with respect to the requirements in Section 3.2. Our discussion revealed that the existing approaches fail to fully satisfy the requirements. We identified the parts of a solution that can be developed based on the existing research results and what still is missing in the existing approaches.

Iteration 2: Consolidation of Systematic Knowledge

The problem investigation gives rise to the following question that remains open due to the lack of a solution:

RQ2 How to systematically design healthy software ecosystems?

The thesis's approach to answer this question is to extract and consolidate architectural knowledge of well-established software ecosystems in practice and related literature and make them reusable for future applications. The goal is to make this knowledge available in a structured way by developing novel methods and tools. For this purpose, we apply a *software product line engineering* approach for the domain of software ecosystems. As mentioned in Section 2.4, using application of software product lines is an effective way to overcome the architectural complexity by means of reusable artifacts. In our SPLE approach, we aim for identifying reusable architectural design decisions that is aligned with the thesis's goal, i.e., facilitating architectural decision-making by providing architectural knowledge.

First, we conduct a *domain engineering* for software ecosystems. Figure 4.3 shows the thesis's domain engineering approach. The process of domain engineering comprises five main sub-processes, i.e., *product management*, *domain requirements engineering*, *domain design*, *domain realization*, and *domain testing*.

In *product management* sub-process, economic aspects of products with respect to market strategies are defined. Furthermore, scoping techniques are used to define which products are inside and which products are outside of the product line. While our domain engineering approach is performed in the context of a research setup with a focus on architectural knowledge and ecosystem health, the market strategies become less relevant. However, to specify the scope of our study, we need a solid definition of software ecosystems using which we can identify ecosystems. For this purpose, we refer to a definition of software ecosystem presented in Section 2.2.1 that is widely used in literature.

In *domain requirements engineering*, we develop a product line for software ecosystems that encompasses architectural commonalities and variabilities of these systems. The commonalities and variabilities respectively refer to common and variable design decisions of ecosystems and their relations (R1, R2, R3).

In *domain design*, the goal is to identify reference architecture that can be used as a guideline to choose among the architectural variabilities. For this purpose, we identify the main architectural designs of software ecosystems in form of architectural patterns.

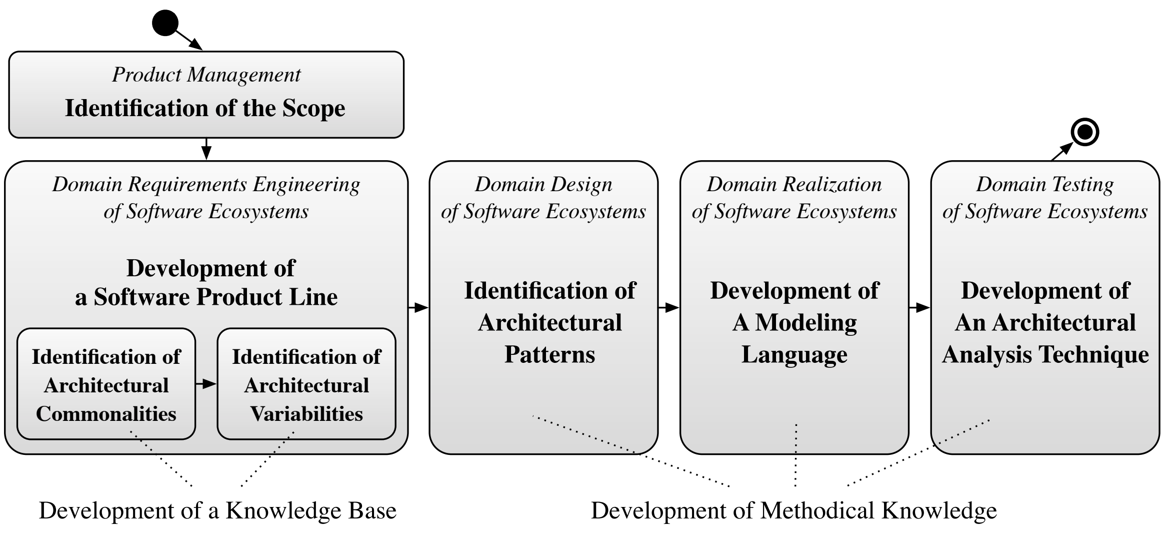

Product Management
Identification of the Scope
Domain Requirements Engineering of Software Ecosystems
Development of a Software Product Line
Identification of Architectural Commonalities
Identification of Architectural Variabilities
Domain Design of Software Ecosystems
Identification of Architectural Patterns
Domain Realization of Software Ecosystems
Development of A Modeling Language
Domain Testing of Software Ecosystems
Development of An Architectural Analysis Technique
Development of a Knowledge Base
Development of Methodical Knowledge

Fig. 4.3 Thesis's Domain Engineering Process
(Adapted from The SPLE Framework [PBv05, chap. 2])

The architectural patterns should be used to clarify the relation of the ecosystem architecture to business objectives and quality attributes of ecosystem health (R4, R6).

Furthermore, during the *domain realization*, we consider the design and implementation of reusable architectural design decisions. As discussed in Section 1.2, there is still a lack of a modeling approach that can be used to specify the ecosystem architecture. To enable the design of ecosystem architecture based on the architectural commonalities and variabilities, we develop a modeling language that makes this knowledge available for modeling purposes (R5).

In addition, *domain testing* needs to facilitate the validation of reusable architectural design decisions once they are realized in the architecture. With this respect, we aim at providing an architectural analysis technique that can help platform providers assess the ecosystem architecture with respect to the knowledge of the product line as well as the knowledge of ecosystem health (R6).

While the goal of our domain engineering approach is to provide systematic architectural knowledge, there is a need for further guidance on how to use the results of our domain engineering to design software ecosystems. As mentioned in Section 2.4, development of applications using the output of domain engineering is called *application engineering*. We develop a design process that can be used during application engineering to facilitate the creation of software ecosystem models (R7). Our domain engineering and application engineering approaches are conducted in the further nested iterations, i.e., Iteration 3 – Iteration 8.

Iteration 3: Identification of Architectural Commonalities

The first part of domain engineering is to identify architectural commonalities of software ecosystems. Such commonalities are the decisions that are key to the architecture independent of platform providers' organizational context. Therefore, our research question is:

RQ3 What are common architectural design decisions of software ecosystems and how do they relate to each other?

As discussed in Section 1.2, several concepts related to the architecture of software ecosystems have already been discussed in literature whereas a unified view of the interdisciplinary architectural aspects is still missing. Therefore, a solution is to obtain a unified view of the common architectural decisions by extracting the knowledge from the literature. For this purpose, we perform a *systematic literature review* [KBB$^+$09] of scientific sources from computer science and other fields such as business informatics and economics.

A main challenge to obtain a unified view from the interdisciplinary sources is the heterogeneous terminology used by these works while referring to different instances of software ecosystems and their architectural constituents [Bos09]. For example, *IT service markets*, *cloud computing markets*, and *App stores* are all instances of software ecosystems despite, sometimes, not being directly discussed in the context of software ecosystems. To overcome the heterogeneous terminology, we use *grounded theory* [SC94] that is introduced for interpreting and conceptualizing data available across different sources of knowledge.

Iteration 4: Identification of Architectural Variabilities

Understanding architectural variabilities of software ecosystems and their relations is a crucial part of their architectural knowledge that forms the second part of our software product line engineering. Therefore, the research question is:

RQ4 What are variable architectural design decisions of software ecosystems and how do they relate to each other?

Our approach to answer this question is twofold. Firstly, we develop a variability model by means of a *taxonomy development method* [NVM13]. We choose to use this method because it is designed to develop taxonomies based on different sources of information. In our work, we leverage the literature and the information available on the web, e.g., technical documentation concerning software platforms, developer forums, and platform providers' annual business reports.

Secondly, we identify alternative service provision scenarios in software ecosystems by conducting *semi-structured expert interviews*. While the variability model deals with single design decisions and their interrelations, the service provision scenarios provide an overview to the overall ways that services are provided in software ecosystems.

Iteration 5: Identification of Architectural Patterns

Upon being informed about variabilities of architectural design decisions, ecosystem architecture can be configured. To this end, it is important to know when to take a certain design decision and what relations between the ecosystem architecture, business objectives, and ecosystem health exist. Answer to the following question can ease this decision-making:

RQ5 What are the main architectural designs of software ecosystems and what is their linkage to business objectives and quality attributes of ecosystem health?

We answer this question by identifying three architectural patterns for software ecosystems. Each pattern represents a group of architectural design decisions and their relations to quality attributes of ecosystem health and business objectives. For this purpose, we apply a pattern mining method called *KJ method* [Ray97] to identify and cluster recurrent groups of architectural design decisions by investigating 111 software ecosystems in practice. We examine the ecosystems with respect to the variability model (cf. Iteration 4) to identify the recurrent architectural decisions that are realized in these ecosystems.

We investigate the organizational context in the ecosystems of our pattern mining data set. Based on the collected information and the clusters of design decisions, we identify whether there is a relation between the architectural designs and platform providers' organizational contexts. We clarify the linkage between the architectural designs, business objectives, and ecosystem health on the basis of a quality model of ecosystem performance introduced by Ben Hadj Salem Mhamdia [BHSM13].

Iteration 6: Development of A Modeling Language

We address the need for a modeling approach for software ecosystems by designing a modeling language on the basis of the architectural knowledge developed in the previous iterations. Developing a modeling language for this objective is a design problem that we formulate using the following question:

RQ6 How to assist the platform providers to model ecosystem architecture?

We develop a modeling language by following the principles of model-driven engineering (MDE) presented in Section 2.5. The modeling language comprises a metamodel and visual notations. The metamodel represents the domain knowledge developed in Iteration 3 and Iteration 4, i.e., the architectural commonalities and variabilities. The visual notations enable the specification of ecosystem architecture in models.

Iteration 7: Development of An Architectural Analysis Technique

We facilitate the assessment of ecosystem architecture by providing an architectural analysis technique that can be used by platform providers to analyze suitability of the architecture with respect to the quality attributes of ecosystem health and business objectives. Similar to the development of the modeling language in the previous iteration, this is a design problem that we describe as follows:

RQ7 How to enable the platform providers to assess ecosystem health?

We develop the architectural analysis technique on the basis of the architectural patterns developed in Iteration 5. The architectural patterns provide methodical guidance to design software ecosystems, however, they do not directly support the assessment of the designs. We implement the groups of architectural design decisions in the patterns by means of constraints. The architectural analysis technique uses the constraints to analyze the ecosystem architecture with respect to the knowledge of the patterns. The results of the analysis show to which extent the architecture conforms to each pattern. Further conclusions are drawn based on this knowledge, i.e., whether the quality attributes of ecosystem health and business objectives are addressed, or how the architecture matches with a platform provider's organizational context. The architectural analysis technique and the modeling language are implemented in a modeling workbench that provides a unified environment for designing and analyzing software ecosystems. We use the modeling workbench during the validation of the SecoArc framework.

Iteration 8: Development of a Design Process

There is a need for guidance on how to apply the architectural knowledge of the SecoArc framework in practice in order to design ecosystem architecture. As mentioned earlier in this Section, this part of our solution approach addresses application engineering of the product line developed in this thesis. Thereby, the question is:

RQ8 What are guidance models so that platform providers can use the framework?

We react to this question by developing a design process based on the methodical knowledge of the architectural patterns specified in Iteration 5. The design process helps answer questions such as how to select among the patterns? How to apply (a combination of) the patterns? And, how to consider platform providers' organizational contexts in the process of ecosystem design? Furthermore, they assist platform providers to analyze and compare more than one architecture by means of the architectural analysis technique.

Once the SecoArc framework is developed, it needs to be assessed with respect to its goal and the requirements (R1-R9) presented in Section 3.1. In other words, it should be assessed whether the artifact designed in this thesis causes an explicit improvement in the problem context. Our problem context described in Section 4.1 raises the following question:

RQ9 Does SecoArc improve architectural decision-making to design software ecosystems?

In design science, the assessment of design artifacts, which justifies whether the artifacts would be useful with respect to stakeholders' goals, is referred as *validation*. Validation of the SecoArc framework comprises two studies.

In the first study, we carry out a technical action research within the scope of *On-The-Fly Computing* [SFB20]. A *technical action research* (TAR) [Wie14, chap. 19] is an experimental usage of the artifact by the stakeholders. Results of this usage are assessed and further used to improve the artifact. On-the-fly computing confronts with architectural challenges that are related to the design of software ecosystems and pursued by this thesis, e.g., dealing with architectural variabilities and having stakeholders with different viewpoints.

In the second study, we examine existing ecosystems from different application domains by means of the SecoArc framework to assess whether the framework provides a suitable abstraction to capture the architectural variabilities of diverse ecosystems.

In general, the validation is a part of Iteration 1 that assesses the thesis's solution concept developed in Iteration 2. The output of Iteration 2 is the SecoArc framework.

4.3 Chapters and Associated Research Questions

Our solution approach to improve architectural decision-making while designing software ecosystems is to develop an architecture framework that makes systematic knowledge available for this purpose. We apply the design science methodology to derive and consolidate the architectural knowledge of best practices from the existing ecosystems and related literature. Our solution approach is related to nine main research questions discussed in Section 4.2, i.e., RQ1 – RQ9.

Figure 4.4 outlines an overview of the research questions and their relations to the thesis chapters. In chapter 1, we presented an introduction to the thesis's research problem and our solution approach, which is followed by Chapter 2, where we presented the research foundations of our solution concept. In Chapter 3, we introduced the requirements of a solution concept. Afterward, we investigated the thesis's problem context that concerns RQ1. Thereby, we discussed suitability of the related work with respect to the requirements.

In Chapter 4, we provided our overall research approach to develop the SecoArc framework as a solution for the thesis's problem. By developing the SecoArc framework, we aim at providing systematic architectural knowledge to design healthy software ecosystems, which is the focus of RQ2. While Chapters 5, 6, 7, and 8 concern RQ2 as well, each of these chapters is specifically dedicated to the development of different constituent parts of the SecoArc framework. In Chapters 5 and 6, we identify a product line for architectural design decisions of software ecosystems that comprises architectural commonalities (RQ3) and architectural variabilities (RQ4). Chapter 7 presents the development of architectural patterns (RQ5), followed by Chapter 8, where we introduce a modeling framework based on our results from the previous chapters. The modeling framework aims at providing a modeling language (RQ6), an architectural analysis technique (RQ7), and a design process (RQ8).

Chapter 9 validates the SecoArc framework with respect to our central validation question that is RQ9. Chapter 10 concludes the thesis by discussing the main contributions, requirements, and future research work.

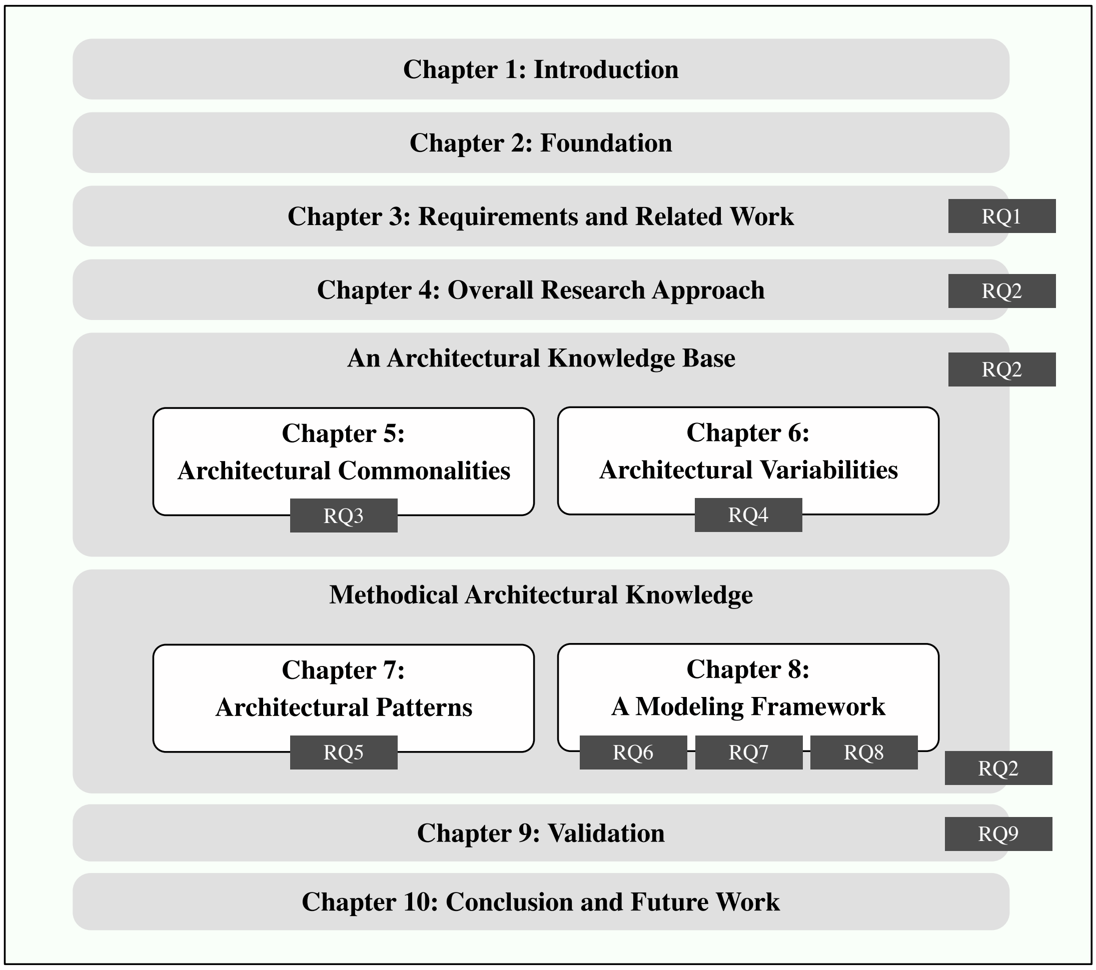

Chapter 1: Introduction
Chapter 2: Foundation
Chapter 3: Requirements and Related Work
RQ1
Chapter 4: Overall Research Approach
RQ2
An Architectural Knowledge Base
RQ2
Chapter 5: Architectural Commonalities
RQ3
Chapter 6: Architectural Variabilities
RQ4
Methodical Architectural Knowledge
Chapter 7: Architectural Patterns
RQ5
Chapter 8: A Modeling Framework
RQ6
RQ7
RQ8
RQ2
Chapter 9: Validation
RQ9
Chapter 10: Conclusion and Future Work

Fig. 4.4 Thesis Chapters and Associated Research Questions

Chapter 5

Architectural Commonalities of Software Ecosystems

In this chapter, we identify architectural design decisions that are common among software ecosystems. This chapter is part of our domain engineering approach to develop a product line for software ecosystems in response to the lack of an architectural knowledge base. The goal is to provide an answer to RQ3 introduced in Chapter 4: *what are common architectural design decisions of software ecosystems and how do they relate to each other?*

We, first, present the procedure of a systematic literature review in Section 5.1, where we identify as many publications related to the scope of software ecosystems as possible. This is followed by deriving the architectural knowledge from a final set of publications and consolidating it into a set of common design decisions in Section 5.2. We refer to the common decisions as *primary features* to denote the architectural elements resulting from applying the decisions and present them in Section 5.3. This is followed by a discussion on the interrelations between the primary features in Section 5.4. Our findings are further discussed and summarized in Section 5.5. This chapter is based on our previous publications [JPEK16a, JPEK16b].

5.1 Procedure of a Systematic Literature Review

The objective of our investigation is to extract primary features of software ecosystems that are discussed in literature across different disciplines. We follow Kitchenham's guidelines [KBB$^+$09] to perform a systematic literature review (SLR) in order to ensure reproducibility and minimizing biases regarding our results. We choose *Google Scholar* as the search database, as recommended by Kitchenham [KC07]. Google Scholar is a meta-search engine that performs searches through several digital libraries.

Initially, a *review protocol* is specified, which is derived from RQ3 and our fundamental research topic (software ecosystems). A review protocol defines the step-by-step actions that are undertaken in an SLR. In the following, we describe how the publications are filtered at each stage using precise criteria, i.e., search phrases and in-/exclusion criteria. In addition, to identify more relevant sources related to our context, we apply snowballing [Woh14] by inspecting the outgoing references cited by the sources.

1. Initial Set of Sources: To find an initial set of sources, we define a set of search terms. The ideas of the search terms are inspired from RQ3 and our observation of the existing software ecosystems. Our main search terms are *software* and *ecosystem*. We specified alternative terms to detect as many of the relevant sources as possible. *Service*, *App*, *application*, *third-party*, *plug-in*, and *component* are the alternatives to the term *software*. Furthermore, the alternatives to the term *ecosystem* are defined as *market*, *marketplace*, *store*, *"App store"*, *repository*, *"archive network"*, and *catalog*. Lastly, we determine our search phrase as a combination of the search terms and boolean operators. During the search, a source is selected if at least one of the combined terms from each set appears in the title. This process resulted in finding 529 sources.

2. Final Set of Sources: We filter the initial set of sources using the in-/exclusion criteria. We include a source if a) it deals with a definition of software ecosystems, b) it deals with one of the functional or the non-functional aspects of software ecosystems, c) it discusses a new instance of software ecosystems, or d) it considers the architecture of software ecosystems. If a source does not directly refer to software ecosystems as its research field, we refer to the definition of software ecosystems given in Section 2.2.1 to identify whether the object of the study is a software ecosystem.

Furthermore, the source must be available through the most prominent digital libraries, e.g., *SpringerLink*, *ACM Digital Library*, *IEEE Xplore*, *Citeseer library*, and *Science Direct*. We exclude a source from our survey if a) the third-party developments discussed are not IT solutions or b) in the ecosystem, discussed by the source, other services/products than IT solutions are provided. We add two additional exclusion criteria as follows:

- The source should not be in the form of a preface, tutorial, book review, or presented slide. This allows us to focus on the high-quality research, e.g., by excluding the publications, which have not been peer reviewed.

- The source should not be published earlier than 2008. The reason for choosing this specific year is the launch of the first mobile App stores, i.e., *Apple App Store* and *Google Play* in 2008 [JB13]. Using this exclusion criterion, we focus on the most recent work. We expect the most prominent research achievements, which were published before 2008, are reflected by the recent work.

We evaluate the sources firstly based on their abstracts and conclusions. Secondly, if reading the abstract and conclusion does not help us decide about the relevance of a source according to the in-/exclusion criteria, we read the whole source.

At this stage of filtering, we have a final set that includes 94 sources. The SLR helped us to identify the sources that objectively address software ecosystems (e.g., [LBWK18], [KDKB14], [SY17a], etc.). However, it is not yet clear which architectural characteristics of software ecosystems are discussed by these sources and whether there are relations between them. Therefore, we use the final set of sources in the next section to extract this information. The complete sets of sources can be found in our technical report [JPEK16b].

5.2 Extraction Procedure using Grounded Theory

This section presents the extraction procedure of primary features from the final set of sources that we identified in Section 5.1. The challenge regarding software ecosystems is that firstly, unlike other paradigms like cloud computing [MG11] and service-oriented computing [HS05], there is no reference model or one unified definition [MH13]. Consequently, the publications do not usually address software ecosystems directly. Secondly, when addressing software ecosystems, they use inconsistent terminologies according to the underlying technologies. For instance, we encounter alternative words for "software", e.g., "service", "application", "App", "SaaS", "API", etc. As a result, we cannot directly identify a set of features from the sources using keyword-based data search. Instead, an interpretation of the information provided by the sources is needed.

We develop an extraction scheme for primary features of software ecosystems based on an adoption of grounded theory (GT). Glaser and Strauss [GS67] originally proposed GT to support researchers to elaborate a theory or a theoretical report of the general features of a topic by performing a bottom-up conceptualization of the data. The data is collected based on empirical observations. We follow the guidelines provided by Wolfswinkel et al. [WFW13] for rigorously reviewing and analyzing the data provided by literature using GT.

The literature analysis consists of an initial excerpting and three stages of codings (*open coding*, *axial coding*, and *selective coding*). Initially, the research focus of the sources is excerpted according to an initial research question. Open coding is the process of grouping a set of excerpts into a *basic concept*. Axial coding is the process of defining *advanced concepts* and identifying the relation between them. An advanced concept is an abstract interpretation of its basic concepts. By selective coding, *main categories* and the relation between them are specified and further refined [WFW13].

To extract primary features of software ecosystems from the final set of sources, in the first step, we inspect the sources carefully, while having our RQ3 in mind. During reading, we look for possible answers to the question. Specially, we consider *what* the sources deal with. We highlight and make notes of the data whenever the architecture of software ecosystems is discussed. After extracting the important information, we apply the open coding, which results in 50 basic concepts in total.

In the second step, we perform axial coding. In comparison with the open coding, axial coding captures less specific architectural topics or larger features. The axial coding results in grouping the basic concepts into 28 advanced concepts. Tables 5.1, 5.2, and 5.3 show the advanced concepts and their relation to the sources. The concepts are presented in three tables while each table reflects the concepts related to an architectural aspect of software ecosystems introduced in Section 2.2. A cell marked with X denotes

Table 5.1 Advanced Concepts related to Social and Organizational Aspects ("X" means the concept is considered by a source.)

	Social and Organizational Concepts				
Source	Contribution Model	Distribution Channels	Release Model	Interaction Model	Knowledge Sharing
[APA14, CSS$^+$13]		X			
[AOJ17, JC13, VDBJL10]				X	X
[DGSB10, BJ12]				X	
[JFB09]		X	X		X
[VK11]	X		X	X	
[VAKJP11]	X	X			
[SEL14]	X			X	
[Bos09]	X			X	X
[JBSL12, MI11]	X				

Table 5.2 Advanced Concepts related to Business and Management Aspects ("X" means the concept is considered by a source. "O" means impacts of other concepts on the concept considered by a source.)

	Business and Management Concepts										
Source	Revenue Model	Price Model	Product Portfolio	Strategies	Ranking	Reviewing	Rating	Dynamic SLA	Openness Policies	License	Policy
[KKLK13]	X		X								
[MVG+14, MKM11]	X										
[HJZ12, JB13]	X	X									
[JBL12, RA12]		X		X	O						
[Car12]	X			X							
[TTP11]	X			X							
[LMD+13, IvJ11, HHYH09, WB10]			X								
[GWB10, BCPR09]	X	X	X								
[PO09]	X	X									
[WGB11]		X		X							
[HJZ12]	O	O			X	X					
[JBL12, Car12, LR11]	X	X	X	X	O						
[MHJ+15]						X					
[CLH+14, FLL+13, GMBV12]						X	X				
[WGB11]								X			
[KDKB14, VDBJL10, JB13, Bou10, HO11, JBSL12]									X		
[KPKL14]	X		X								
[BJ12, Wei18, Aza15, SA12, AAS09]									O	X	X

that a concept is discussed by a source. Some sources additionally consider the relation between the concepts. In this case, a cell marked with O denotes that a source studies the impact of a concept on its main research topic (marked with X). Notably, some concepts belong to one aspect, while some others are common among the aspects. This indicates that some concepts are discussed from different perspectives by the sources. An example is *openness policies* whereas some sources [KDKB14, VDBJL10, JB13, Bou10, HO11] discuss critical business decisions while opening software platforms and other sources [SY17a, BHH15, SBH14, AJ10, KDKB14] consider tools and techniques to realize such business decisions.

In the next step, we specify the main categories by performing selective coding, which is the act of grouping the related concepts into new categories. The main categories represent the most abstract architectural building blocks of software ecosystems. In

Table 5.3 Advanced Concepts related to Application and Infrastructure Aspects ("X" means the concept is considered by a source. "O" means impacts of other concepts on the concept considered by a source.)

	Application and Infrastructure Concepts													
Source	API Documentation	API Management	Privacy	Static Code Analysis	Run-Time Malware Detection	Repository	Store	Quality of Service (QoS) Analysis	Service Composition	Semantic Interpretation	Service Discovery	Service Matching	Openness Policies	Monitoring System
[Bos10, RL11, LS20, RLR12, MSJ+17]	X	X												
[Aza15, MRK13, CG12, SA12]	X													
[LBWK18, SBL+19]		X												
[EJM+14]				X										
[CG12, ZWZJ12]					X	O								
[GCCJ11]			X	X			O							
[HA12, DMA+10, SK11, TSB10, PO09, IKC09, LFZD09]			X											
[ZWZJ12]						X								
[DSTH12]						X								
[CJP+11]			O					X						
[HO11]							X						O	
[CWO+11]						X		X						
[BRC10]						X	X	X						
[LFZD09]						X								
[APB+14]									X		X	X		
[MVG+14]									X		X			
[FHT+12]												X		
[TSB10]									X	X				
[WB10, AHKZ08, KW08]									X	X	X			
[HHYH09]												X		
[SY17a, BHH15, SBH14, AJ10, KDKB14]													X	
[PRS09, LFZD09]														X

addition, we consider *what interrelations* the research results reveal between the concepts.

In the following, we elaborate on an example to show how the coding technique results in generating one of the main categories, namely reputation system. Reputation system is an important feature of software ecosystems that generates value for trustworthiness and quality of services or participants' activities on the basis of users' ratings [HGS13]. Figure 5.1 shows the results of the coding process. After reading the sources and extracting excerpts, we start to perform open coding from [LB13], which resulted in the identification of two concepts: ranking chart and download rank. Afterward, we notice that [HJZ12] shares the concept download rank with [LB13]. In addition, it introduces a new concept App mining.

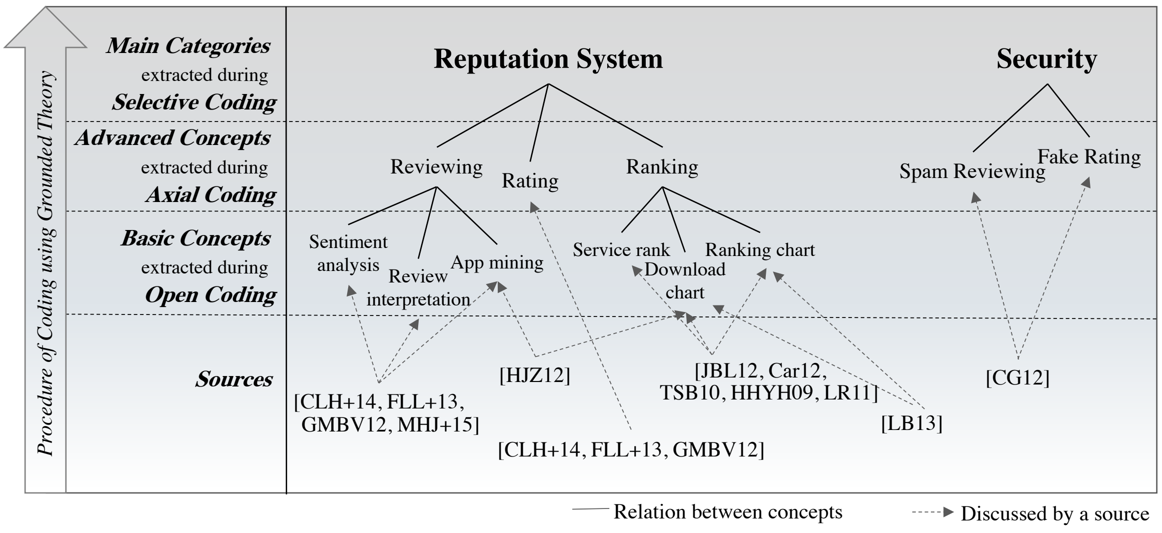

Procedure of Coding using Grounded Theory
Main Categories extracted during Selective Coding
Advanced Concepts extracted during Axial Coding
Basic Concepts extracted during Open Coding
Sources
Reputation System
Security
Reviewing
Rating
Ranking
Spam Reviewing
Fake Rating
Sentiment analysis
Review interpretation
App mining
Service rank
Download chart
Ranking chart
[CLH+14, FLL+13, GMBV12, MHJ+15]
[HJZ12]
[CLH+14, FLL+13, GMBV12]
[JBL12, Car12, TSB10, HHYH09, LR11]
[LB13]
[CG12]
Relation between concepts
Discussed by a source

Fig. 5.1 Coding of Reputation System and Related Security Concepts

We proceed this process with [JBL12], [Car12], [LR11], [TSB10], and [HHYH09], which results in sharing two concepts, namely ranking chart and download rank, with the previous sources and in generating a new concept service rank. Later, by performing axial coding, we grouped service rank, ranking chart and download rank into one category: ranking. Furthermore, the open coding of [MHJ+15], [CLH+14], [FLL+13], and [GMBV12] shares App mining with the previously coded sources and generates two new concepts: review interpretation and sentiment analysis. These concepts are categorized as reviewing through the axial coding. The open coding of [CLH+14], [FLL+13], and [GMBV12] generated the category of rating. Finally, we grouped three categories of ranking, reviewing, and rating into a main category of reputation system. In addition, the coding process reveals links (sharing codes) between reputation system and two other main categories, which we have identified later (security and business model).

We terminate the process of coding when the so-called *theoretical saturation* happens. This is a situation when no new category, concept, or relation related to the research question can be found [SC94]. The result of our coding procedure is the identification of ten main categories, i.e., reputation system, business model, API management, actor participation model, software supply network, social network, software element, security, recommendation system, and service level agreement (SLA), and 28 advanced concepts (cf. Figure 5.2). We refer to the main categories as *primary features.* As mentioned in Section 2.4.2, a feature is a key external characteristic of a system and a basic constituent of a product line. We also call their advanced concepts *sub-features* since they represent more specific design decisions that help realize the functionality of the primary features. We have already mentioned the results of the axial and selective codings. Details about the results of the open coding can be found in our technical report [JPEK16b].

5.3 Primary Features of Software Ecosystems

In this section, we introduce the primary features of software ecosystems that are the results of our extraction scheme in Section 5.2. Figure 5.2 provides an outline of the primary features and their sub-features using the notation of a feature model [LKL02]. The features are distributed among three aspects, i.e., social and organizational, business and management, and application and infrastructure. While some features belong to one aspect, some others like openness policies related to more than one aspect since they hold characteristics of those aspects. In the following, we elaborate on the features.

Reputation System: A reputation system is responsible for collecting and aggregating users' opinions regarding software quality and generating rankings. A well-functioning reputation system builds trust among ecosystem participants [RKZF00]. A reputation system consists of both business and technical aspects. Among others, the business aspects are concerned with the validity of users' opinions (in contrast to being faked) and the impact of a reputation system on market success. Software components facilitating the functionality of a reputation system constitute the technical aspects.

The main sub-features, identified by our literature analysis, are rating, reviewing, and ranking (cf. Figure 5.2). Using the rating feature, users can express their satisfaction

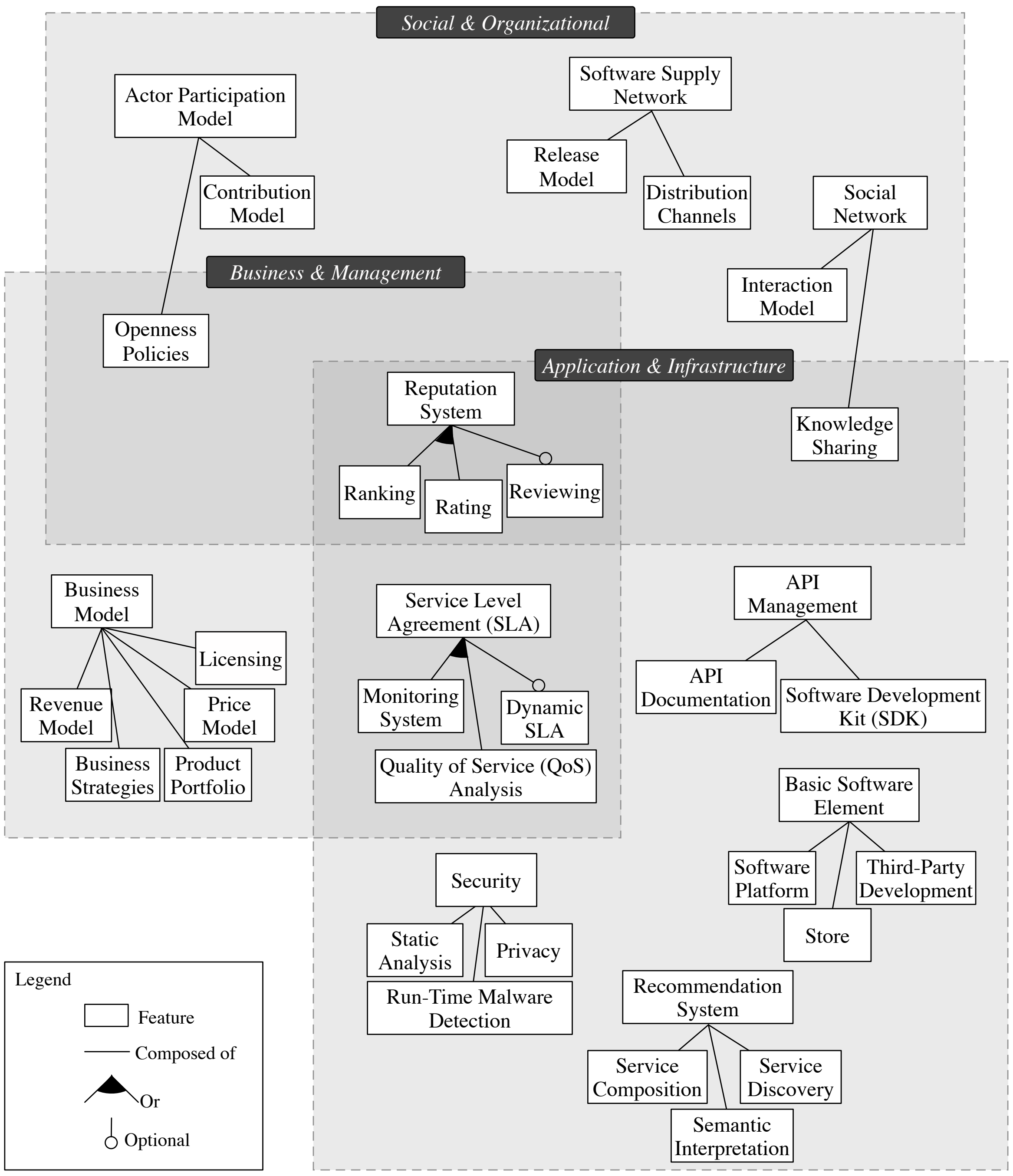


Fig. 5.2 SecoArc Model of Primary Features of Software Ecosystems

in terms of digits. Binary, star, and scale rating are examples of the rating feature in software ecosystems. Reviewing enables users to insert their opinions about software provision in ecosystems. In addition, it helps to interpret the opinions using opinion analysis techniques such as sentiment analysis. An informative interpretation of a mass number of reviews and detecting inconsistencies between user comments and ratings are the exemplary tasks handled by reviewing [FLL$^+$13]. Furthermore, rankings are associated with the third-party developments, ecosystem participants, and reviews. Ranking algorithms highly rely on the *download rank* as a key quality indicator to generate ranking charts [LB13].

Business Model: A business model outlines the elements that make a business successfully generate and deliver value to its stakeholders including customers [Tee10]. A platform provider's business model consists of a holistic range of business decisions that affect the whole ecosystem in a variety of ways. However, some key parts of such decisions are implemented by means of online stores, where the third-party developments are distributed.

The most important sub-features are revenue model, price model, product portfolio, business strategies, and licensing. A revenue model is associated with platform providers' business design decisions to choose revenue sources, revenue sharing with third-party actors, and generated revenue for them [MVG$^+$14, MKM11]. Price model includes the decisions to choose pricing schemes a) for the third-party actors and in return granting access to the platform, and b) for the users to use the third-party developments [RA12]. Furthermore, product portfolio represents the strategies regarding characteristics of a service, e.g., product diversification, which is the support for multi-homing [IvJ11] (i.e., a company's strategy to support multiple platforms with one software product). Further examples are covering several thematic categories and targeting different groups of users. Business strategies [Tee10] are analytic plan-makings regarding the competitive market environment, e.g., third-party providers' decisions on licensing greatly influence their survival in the ecosystem. Finally, licensing concerns a legal aspect of both open platforms and third-party developments that specify the circumstances under which they can be used or further shared in future [SA12].

Recommendation System: A recommendation system handles the discovery and delivery of the desired services by employing the existing knowledge and statistics [SKKR00]. Our results show that recommendation systems in software ecosystems include service discovery, service matching, semantic interpretation, and service composition. Service discovery is a range of techniques to optimize the discovery of third-party developments in the pool of existing developments. Examples of such techniques are service matching using comprehensive service specifications and SLA-based service selection. Service matching is a decision-making function that evaluates an approximate matching of a request to a service specification or a software service to an execution resource. Service composition enables the dynamic provision or recommendation of individually composed services, each provided by different providers. The outcome of a recommendation system can be enhanced by taking the advantages of semantic interpretation, for instance, by employing ontologies to improve the service discovery [MVG$^+$14, APB$^+$14, KKLK13].

API Management: It refers to a group of software utilities and shared libraries that enable the development of third-party software on top of the platforms. Therefore, third-party actors can develop platform-specific software. While an API management system may consist of various parts, the most recognized ones are software development kit (SDK) andAPI documentation. An API documentation is a detailed list of a platform's application programming interfaces (APIs). The APIs allow the third-party software to function on the basis of the platform and get access to software or infrastructure resources provided by the platform [LS20]. Examples of API documentation are *Android API reference*[1] and *Apple Developer Documentation*[2].

Another critical part of successful software development on top of open platforms is related to API compatibility. Incompatibility of new and the old APIs during an update is managed using API management tools and techniques. Various techniques are used to capture different kinds of compatibility issues. Forward and backward incompatibilities are among the most considerable issues. Backward incompatibility is considered from the platform's perspective. It is the situation, where the APIs of a third-party development do not match the older versions of a platform and, thereby, leading to software failures. This is while forward incompatibility is when some APIs of a third-party development are deprecated according to the newest version of a platform [LBWK18, RLR12].

Actor Participation Model: An actor participation model is concerned with a group of strategies that define how the third-party actors can *enter an ecosystem* and *contribute*. One of the most popular instances of such strategies are openness policies. In this context, ecosystems become more closed by applying entrance barriers such as fee or qualification criteria for the third-party actors [KDKB14]. Furthermore, which

[1]`https://developer.android.com/ndk/reference`. Last Access: January 10, 2022.
[2]`https://developer.apple.com/documentation`. Last Access: January 10, 2022.

parts of the platforms' source code is available to third-party actors and whether their contributions are integrated into the platform are the other questions that are answered by a set of design decisions that we refer to them as contribution model. For instance, in free and open-source software (FOSS), approved third-party developers can contribute to the development of the platforms since their third-party developments are integrated after a suitable amount of testing and quality check is performed [JBSL12].

Software Supply Network: It indicates the relationship between the actors, which determines software production and delivery in ecosystems. Distribution channels specify how a new version of a platform or third-party developments are delivered to the users [JBSL12]. A typical design decision is to use online stores to distribute the software. Two examples are Apple App Store[3] in mobile applications and AppExchange[4] for cloud computing services. Additionally, a release model defines the way that an update is initiated by the actors and propagated throughout the ecosystem. An update can be initiated by users e.g., when there is a feature request from the user's side, by third-party actors, e.g., to improve the software and removing some bugs, or by ecosystem providers, e.g., as a matter of the scheduled and regular platform update [RS12].

Social Network: It refers to the interactions between the community of human actors in software ecosystems [Val13]. An interaction model clarifies who can communicate with whom and using which technologies such interactions should happen. There are several models to specify the interactions between the actors. In some cases, third-party developers are not connected. In other cases, they can communicate with each other but not with the partners. Another possibility is to help third-party developers get connected to the partners. A second concern is knowledge sharing that refers to certain technologies used in software ecosystems to facilitate social interactions and establishing common knowledge. Examples of knowledge sharing technologies are *mailing-lists* to broadcast news and updates, *issue tracking system*, and *Q&A forums* for problem-solving.

Basic Software Element: A basic software element is predominantly used in software ecosystems. With this respect, we identified three types of elements, software platform, store, and third-party development. Software platform provides the basis for the third-party providers to develop third-party developments, and for the users to run

[3]`https://apple.com/ios/app-store`. Last Access: January 10, 2022.
[4]`https://appexchange.salesforce.com`. Last Access: January 10, 2022.

the developments. The third-party developments are less complex software compared with the software platform, which are devoted to a specific functionality. From the users' perspectives, the functionality of the platform is extended by the third-party developments to perform certain use cases. A store acts as an intermediary between the users and providers of software by linking several thematic catalogs to each other and allowing the users to search through those catalogs in electronic marketplaces. Specially, in stores, third-party providers make their software or their specifications available for the users. Furthermore, repositories often host black-box services or the source code of the software that is already published in the stores [HO11, CJP$^+$11].

Security: Security is another key part of ecosystem architecture. The sub-features are privacy, policy, code analysis, and run-time malware detection. Software ecosystems need to protect the integrity of users' sensitive data, which can be misused by third-party applications. Moreover, source code analysis, intrusion detection, and malware detection algorithms need to be employed to avoid malicious behavior of the applications or violation of users' privacy. Platform providers usually employ such security techniques to ensure their organizational laws and social regulations [ZWZJ12].

Service Level Agreement (SLA): An SLA is a contract between the providers of third-party software and the users, or another actor, such as ecosystem providers, who is in charge to protect users' right to receive high-quality software. The goal of SLA is to ensure a certain degree of software quality. This is directly related to the on-time and reliable fulfillment of quality expectations, which is achieved by using techniques related to quality of service (QoS) analysis and monitoring system. An SLA is specially used in ecosystems, where software is executed on a remote server, and results of the execution are sent to the users. In addition, dynamic SLAs support frequent changes in users' requirements and the heterogeneity of execution resources (cf. Fig. 5.2) [WGB11, PRS09].

While the architecture of software ecosystems is the result of a wide and interdisciplinary range of architectural design decisions, we have identified ten key architectural building blocks and referred to them as primary features. Accordingly, these features are related to different aspects of ecosystem architecture. Specifically, reputation system business model, and SLA are related to the business aspects and management of software ecosystems whereas actor participation model, software supply network, social network characterize social and organizational aspects. Moreover, recommendation system, Software element, security, and API management are the key part of the application and infrastructure of software ecosystems.

5.4 Primary Feature Interrelations

There are interrelations between the features introduced in Section 5.3, i.e., the functionality of the features can affect each other. The knowledge of feature interrelations is based on the literature analysis performed in Section 5.2, where we identified the sources that consider the interrelation between the concepts (see the cells marked with O in Tables 5.1- 5.3). Figure 5.3 summarizes these interrelations. An arrow from feature *A* to feature *B* shows that *A* influences *B* in a certain way that is shown as an arrow label.

Business Model – Reputation System: Both business model and reputation system significantly affect each other. On one hand, strategic decisions taken regarding price model and product portfolio mainly influence rating and ranking. A suitable pricing scheme improves users' ratings, service rank, and customer loyalty. For instance, third-party developments with a combination of free and paid price models receive a better download rank [LR11]. Furthermore, strategies regarding product portfolio like diversifying service categories improve rating and service rank [KPKL14, LR11]. On the other hand, rating and ranking affect sales performance and users' willingness to pay. Consequently, they influence business model and revenue model [Car12, HJZ12].

Business Model – Actor Participation Model: A business model defines the circumstances under which the third-party actors are allowed to contribute to the ecosystem. A prominent part of such circumstances concerns entrance barriers, which determine how easy the third-party actors can enter the ecosystem.,e.g., whether they have to pay fees or fulfill certain requirements. Furthermore, the business model itself is influenced by the type of platform that specifies the degree and quality of third-party actors' contributions. For instance, in the FOSS community, the platform grows by direct collaborations of third-party developers. However, even in the case of FOSS, different collaboration models with different degrees of freedom are followed.

Reputation System – Recommendation System: If an ecosystem includes a reputation system, the results of rating, ranking, and reviewing are used to identify high quality services while recommending services to the users. Thereby, the functionality of the recommendation system can be improved by using the knowledge about high quality

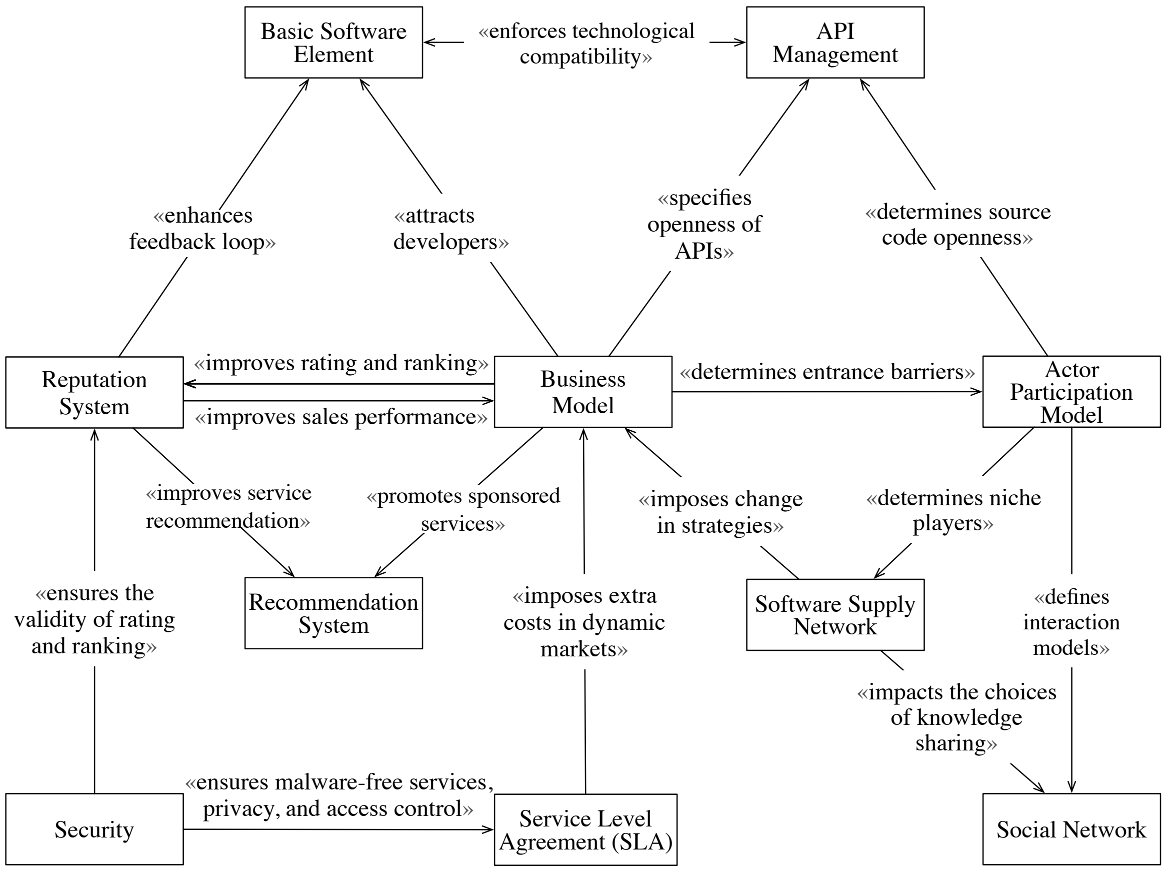

Basic Software Element
«enforces technological compatibility»
API Management
«enhances feedback loop»
«attracts developers»
«specifies openness of APIs»
«determines source code openness»
Reputation System
«improves rating and ranking»
«improves sales performance»
Business Model
«determines entrance barriers»
Actor Participation Model
«improves service recommendation»
«promotes sponsored services»
«imposes change in strategies»
«determines niche players»
«ensures the validity of rating and ranking»
Recommendation System
«imposes extra costs in dynamic markets»
Software Supply Network
«defines interaction models»
«impacts the choices of knowledge sharing»
«ensures malware-free services, privacy, and access control»
Security
Service Level Agreement (SLA)
Social Network

Fig. 5.3 Primary Feature Interrelations

services [APB+14]. A typical use case is to use the results of ranking, e.g., a list of featured services, as input to the recommendation system, whereas thematic relevance of the services is ensured by the service matching as a function of the recommendation system.

Business Model – Recommendation System: One of the main ways that a business model influences the outcome of a recommendation system is the strategic decisions taken by platform providers to promote partners' services in the ecosystem. Another strategy is to promote any service that is sponsored by a provider. In both cases, the platform providers might apply certain quality assurance techniques to ensure the quality of sponsored service before recommending them to the users.

Service Level Agreement (SLA) – Business Model: Execution of software services demands execution resources. Third-party providers normally purchase such resources from external resource providers. In dynamic markets, users of a service change frequently, which implies changes in requirements and the corresponding SLAs. Service providers, who would like to support a wider range of users, need to be able to cope with such heterogeneous SLAs. To avoid SLA violations, they have to take care of heterogeneous execution resources that are needed by different SLAs. This situation continuously imposes extra costs on the service providers. Such trade-offs between cost and the fulfillment of SLAs need to be foreseen in a business model by choosing suitable resource allocation algorithms that handle dynamic SLAs with minimum costs [WGB11].

Software Element – Reputation System: Software ecosystems need to attract third-party providers by providing transparency of processes in the ecosystem that make the third-party providers interested in self-promotion and improving their reputation in the ecosystem. A reputation system in stores enables such self-promotion by enhancing the feedback loop between the developers and users using rating and ranking features. This is additionally supported by providing incentives to developers [DSTH12].

Software Element – Security: Store as a sub-feature of basic software element need to detect new samples of known malware families in order to ensure malware-free services [ZWZJ12]. Another security concern of stores is privacy and access control, which demand encrypted queries in the stores [CJP^{+}11]. Furthermore, stores improve the policy enforcement. For instance, platform providers may apply such policies to third-party applications before granting access to the ecosystems. An example of such policies is security validation to avoid misusing users' privacy-sensitive data [GCCJ11, HO11].

Software Element – Business Model: Business strategies taken regarding basic software elements greatly impact on attracting developers to the ecosystem. However, such strategies usually come with trade-offs. For instance, a store makes third-party developers' businesses centralized and more accessible to their users. In addition, it reduces the distribution costs imposed on the third-party providers. Such costs include the maintenance costs of updating their software and costs of reaching and developing a target group of potential users. However, stores restrict third-party providers' freedom, because they have to conform to a set of policies defined by platform providers, i.e., licensing. Once they cannot conform to those policies, they have to leave the ecosystem [HO11].

Reputation System – Security: Generating valid rankings demands a high degree of security in preventing and detecting manipulated ratings and spam reviews.

In addition, such manipulated ratings and rankings unjustly persuade service consumers. The consequences are disturbing trust and decreasing the QoS delivered in the ecosystem [CG12].

Business Model – API Management: The choice of licensing highly affects the management of the platform APIs and specifically their openness. Licenses are used to determine the ownership of software platforms and third-party developments. Platform providers are in the role of keystone to define whether the platform and the developments are open-source and whether they can be shared and re-used by other actors. In this context, the platform providers sometimes differentiate between the third-party developers and partners by following different business strategies, which depends on the degree of platform's commerciality and profitability of the partnerships [Wei18].

Software Element – API Management: The APIs of software platforms are shared and managed by API management. The way that the APIs are made accessible to the third-party providers demands technological compatibility between the software platform run on the providers' side and the API management. This essentially affects the quality of providers' work [MSJ$^+$17]. Furthermore, if the third-party developments are delivered in a remote way, they should be executable on an external server, which implies the interoperability between the platform APIs and such external servers. To handle this task, the API management needs to be aware of critical APIs of the third-party developments and how to communicate with them.

Actor Participation Model – API Management: The strategies that characterize an actor participation model determine which actors and how can access the source code of the platform and third-party developments. These regulations form an important part of the openness policies. Moreover, an actor participation model defines whether the third-party developers can directly contribute to the platforms. For instance, in the free and open-source software (FOSS) community, under certain regulations, the platforms are developed using the direct contribution of third-party developers.

Actor Participation Model – Software Supply Network: An interaction model as a part of actor participation model defines the communication channels. Such channels can give some actors more privilege than the others and make them niche players in ecosystems. A niche player is an actor, who focuses on a small part of the market and possesses specialized capabilities that are differentiating him/her from the opponents [IL04b]. The niche players are directly related to software supply network and release model, which defines software release patterns in the ecosystems [JBSL12].

Business Model – Software Supply Network: Business strategies defined as a part of the platform providers' business models include the design choices to deliver users the software. In this context, platform providers might want to run the services on a remote server and only deliver the results of the executions. Another way would be to install the software locally on users' devices. Software delivery is related to further business strategies regarding software deployment and the integration on users' execution environment. Furthermore, due to the dynamic environment of software ecosystems, the relationships between the actors are constantly evolving. Thereby, requirements and the importance and relevance of some services change for certain actors. With this respect, platform providers should consider changing business strategies regarding the delivery of certain software services to partners, developers, or users according to ecosystem dynamics.

Reputation System – Recommendation System: If an ecosystem includes a reputation system, the results of rating, ranking, and reviewing are used to identify high quality services while recommending users services. Thereby, the functionality of the recommendation system can be improved by using knowledge about high quality services.

Business Model – Recommendation System: One of the main ways that a business model influences the outcome of a recommendation system is the strategic decisions taken by the platform providers to promote partners' services in the ecosystem. Another strategy is to promote any service that is sponsored by a provider. In both cases, the platform providers might apply certain quality assurance techniques to ensure the quality of sponsored service before recommending them to the users.

Software Supply Network – Social Network: While a software supply network determines the way that the software is distributed in the ecosystem, the decisions regarding the distribution channels specify the characteristics and quality of social interactions between the actors. In particular, knowledge-sharing technologies are chosen to facilitate required interaction models. For instance, in FOSS, where the platforms are developed by the direct contributions of third-party developers, the developers mainly interact using issue tracking systems. In contrary, in commercial ecosystems, third-party developers' interactions happen through Q&A forums [JFB09, VK11].

Actor Participation Model – Social Network: Design decisions regarding an actor participation model, specifically the choice of a contribution model and openness policies, help platform providers define clusters of actors, who belong to one interaction model that follows the same policies. Third-party developers often belong to one group whereas they might communicate using Q&A forums that are provided by platform providers or using external Q&A forums while partners can register in partner programs. Partner programs facilitate objective partner communication using online training courses, webinars, etc.

The knowledge of the feature interrelationships shows that the features impact each other. This is while the business model plays a central role in the architecture since most of the features are in relation to it. It influences the features of technical and non-technical aspects.

5.5 Summary and Scientific Contributions

In this chapter, we identified ten primary features of software ecosystems and their sub-features. The features form the commonalities of a product line for software ecosystems that is part of the knowledge base developed in this thesis. We extracted the features by conducting a systematic literature review and developing an extraction scheme using grounded theory. During the literature review, we used a wide range of alternative search terms in order to overcome terminological heterogeneity and capture the architectural characteristics of software ecosystems from the different perspectives.

Our results reveal that the design choices of the features are not independent, but rather, they influence the outcome of each other. Such effects ultimately contribute to the ecosystem's success. We provided an in-depth discussion of the interrelations between the features.

This knowledge gives ecosystem operators, IT enterprises, and service providers insight into ecosystem architecture and their design choices regarding an enhanced market development and feature integration. The results enable reusing well-established solutions, which is beneficial to the research community as well as to practitioners from two different perspectives: (a) Providers of such ecosystems, like enterprises or individuals, can use the insight into the design choices of software ecosystems in order to develop and integrate new ecosystem features and, thereby, improve their market's success. (b) Developers of certain functionalities like reputation systems, recommendation systems, etc. benefit from the possibility to take into account how their components can or have to interact with the rest of the architecture, in order to become applicable in practice. In summary, the scientific contributions of this chapter are as follows:

- Our study reveals a categorization of the primary features of software ecosystems and the relations between them.
- Our findings provide a joint integrated architectural view from a wide perspective comprising computer science and business disciplines for software ecosystems.
- This chapter consolidates the architectural knowledge of related concepts, e.g., IT service markets, cloud markets, App stores, etc. These are instances of software ecosystems. But they have been discussed separately and not in the context of software ecosystems in the past.
- Our results provide information about how the features are related to each other and how they influence the total well-functioning of ecosystems.

Chapter 6

Architectural Variabilities of Software Ecosystems

In this chapter, we identify architectural variabilities of software ecosystems that are a key part of the software ecosystem product line developed in this thesis. To this end, we consider RQ4 introduced in Chapter 4: *what are variable architectural design decisions of software ecosystems and how do they relate to each other?*

This chapter has two key outcomes: a) a variability model that captures variable architectural design decisions, their dependencies, and their relations to the ecosystem architecture, which are presented in Section 6.1, and b) variable service provision scenarios in software ecosystems that are discussed in Section 6.2. The main difference between the two outcomes is that the former is concerned with the structure of software ecosystems while the latter relates to the interactions that happen in ecosystems. In Section 6.3, we conclude with a discussion and summary of scientific contributions. This chapter is based on our former publications [JZEK17a, JZEK17b].

6.1 A Variability Model for a Software Ecosystem Product Line

In this section, we describe the process of developing a variability model for software ecosystems by using a taxonomy development method in Section 6.1.1. In Section 6.1.2, we present the variability model, followed by introducing the dependencies between the variabilities in Section 6.1.3. Afterward, we provide insights into the relations between the variabilities and the ecosystem architecture in Section 6.1.4. The relations between the variabilities in this chapter and the architectural commonalities in Chapter 5 is the topic of Section 6.1.5.

6.1.1 Taxonomy Development Approach

This section presents our research approach to develop a variability model for software ecosystems by applying a taxonomy development method proposed by Nickerson et al. [NVM13]. We use this method because it enables us to consolidate (a taxonomy of) architectural variabilities from two different types of sources, i.e., existing ecosystems and the literature. The knowledge obtained from one source can be approved or complemented by investigating the other source.

The method helps to define a taxonomy for a set of objects of study. In this context, a taxonomy is a classification of *dimensions*, which differentiate among the set of objects. Each dimension is characterized by a set of *characteristics*. More precisely, a taxonomy (T) is a set of dimensions $(D_i(i = 1,...,n))$ while each dimension (D_i) includes $(k_i \geq 2)$ mutually exclusive and collectively exhaustive characteristics $(C_{ij}(j = 1,...,k_i))$. The mutual exclusive restriction implies that no object can hold two different characteristics that belong to a dimension. The goal is to ensure the distinctiveness of characteristics. The collectively exhaustive restriction implies that each object needs to have at least one of the characteristics in a dimension. The goal is to ensure the universality of dimensions.

As mentioned in Section 2.4, variabilities are described in terms of variation points and variants. Accordingly, our resulting taxonomy is a variability model whereas each dimension is a variation point and each characteristic is a variant. To capture the full semantics of a variability model, we allow to diverge from the mentioned restrictions of the original method *only if there are two dominant groups of ecosystems while one group shows to have a characteristic / one of the characteristics of a certain dimension and the other group does not have that characteristic / any characteristics of that dimension.* This divergence refers to our goal of developing a taxonomy, which consists of architectural variabilities. This specifically means by allowing the characteristics to be mutually inclusive, we can identify *alternative choices.* Furthermore, making exceptions for the collectively exhaustive restriction under the mentioned condition allows us to identify *optional variation points.* Both alternative choices and optional variation points are essential parts of variability models [PBv05, sec. 4.6] [MPH^{+}07].

Figure 6.1 outlines the stepwise process of taxonomy development. In the following, we introduce these steps while elaborating on our actions to develop the taxonomy.

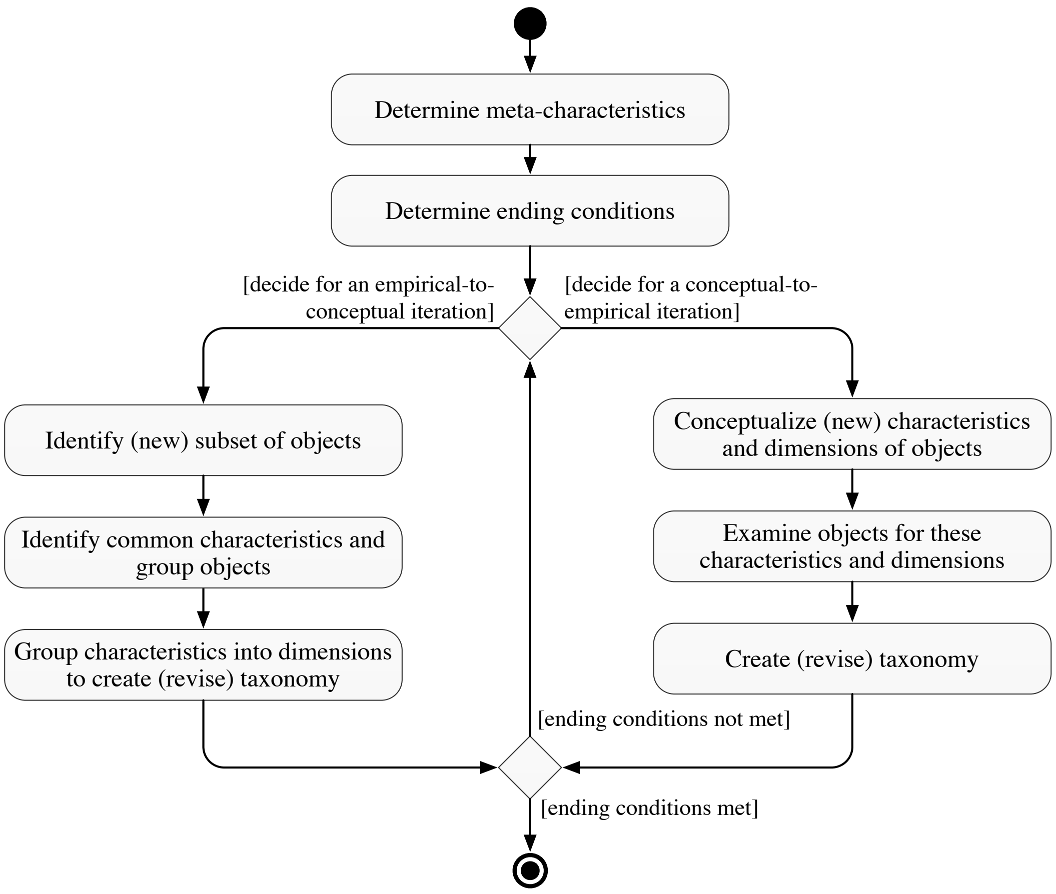

Determine meta-characteristics
Determine ending conditions
[decide for an empirical-to-conceptual iteration]
[decide for a conceptual-to-empirical iteration]
Identify (new) subset of objects
Identify common characteristics and group objects
Group characteristics into dimensions to create (revise) taxonomy
Conceptualize (new) characteristics and dimensions of objects
Examine objects for these characteristics and dimensions
Create (revise) taxonomy
[ending conditions not met]
[ending conditions met]

Fig. 6.1 Process of Taxonomy Development [NVM13]

Meta-Characteristics

The first step of developing the taxonomy is to determine *meta-characteristics*. Meta-characteristics are the most comprehensive characteristics of a group of objects. They serve as a basis to identify further characteristics for each dimension of a taxonomy.

Considering RQ4, our meta-characteristics should assist with identifying the differentiation points, where the architecture of software ecosystems differ with each other, i.e., the sources of architectural variabilities. We identify three meta-characteristics by referring to the definition of software ecosystems presented in Section 2.2.1: a) What a platform consists of, b) Collaborations upon which services are provided in an ecosystem, and c) What an ecosystem offers as third-party products/services. During the process of taxonomy development, we identify characteristics on the basis of these meta-characteristics.

Ending Conditions

Ending conditions specify when the process of taxonomy development is terminated. We set the ending conditions as follows: The development of the taxonomy is terminated when the last empirical-to-conceptual or conceptual-to-empirical iterations do not result in the identification of any new dimension or characteristic or when the taxonomy is not further enriched by using the new data.

Empirical-To-Conceptual Iterations

The research method includes two types of iterations that lead to the development of a taxonomy, i.e., *empirical-to-conceptual* and *conceptual-to-empirical* iterations. Nickerson et al. [NVM13] recommend starting with an empirical-to-conceptual iteration if rich sources of knowledge already exist in real-world that can provide information about the objects of study. As our first iteration, we choose to begin the taxonomy development by conducting an empirical-to-conceptual approach, because a diverse range of ecosystems already exists. This can be a valuable source of data, which is very beneficial for an empirical approach.

In an empirical-to-conceptual iteration, the researchers analyze a sample of the object of study to identify common characteristics that are characteristics that need to be logical consequences of the meta-characteristic. A characteristic needs to be a differentiator among the objects and not to have the same value for all of the objects. Once the characteristics are identified, they are grouped into dimensions by applying formal (e.g., statistical) or informal (e.g., manual) grouping techniques. The researchers are in charge to name the dimensions suitably [NVM13].

We analyze existing ecosystems with respect to their architectural design decisions and the meta-characteristics to draw the variation points of their variants. Table 6.1 lists the groups of ecosystems investigated in the empirical-to-conceptual iterations by referring to their software platforms and online stores. The groups indicate the application domains of software services provided in the ecosystems. While most of the ecosystems own their own software platforms, in some ecosystems, third-party services are developed on the basis of external software platforms. Examples of this situation are the ecosystems providing third-party Android/iOS Apps.

The goal of collecting such a list is to ensure that our examination captures ecosystems from a diverse range of application domains. This goal contributes to a rich variability model. We notice that the ecosystems in the different application domains differ with respect to the meta-characteristics. We analyze the ecosystems by inspecting

Table 6.1 Ecosystems Investigated in Empirical-to-Conceptual Iterations

Software Platform	Store	URL of Store[*]
Mobile Apps		
Apple iOS, MacOS	Apple App Store	https://www.apple.com/app-store
Google Android	Google Play	https://play.google.com
Windows Phone	Microsoft Store	https://www.microsoftstore.com
BlackBerry OS	Blackberry World	https://appworld.blackberry.com
Third-Party Android / iOS Apps		
Google Android	F-Droid	https://f-droid.org/packages
Google Android	Slideme	http://slideme.org
Google Android	Aptoide	https://en.aptoide.com
Apple iOS	Cydia	https://cydia.saurik.com
Google Android	Apkpure	https://m.apkpure.com
GetJar	GetJar	https://www.getjar.com
Google Android	Amazon APPstore	https://www.amazon.com/Appstore
Component & Plug-in Repositories		
GitHub	GitHub Marketplace	https://github.com/marketplace
Ruby	RubyGems	https://rubygems.org
Eclipse	Eclipse Marketplace	https://marketplace.eclipse.org
Enterprise Applications		
Intuit QuickBook	QuickBook Apps	https://apps.intuit.com
Amazon Web Services	AWS Marketplace	https://aws.amazon.com/marketplace
Salesforce, Lightning, etc.	App Exchange	https://appexchange.salesforce.com
GetApp	GetApp	https://www.getapp.com
API Registries		
OpenIntents	OI Apps	http://www.openintents.org/download
Mashape	Mashape Market	https://mashape.com [**]
ProgrammableWeb	API Directory	https://www.programmableweb.com
Others		
Mozilla Firefox	Add-ons for Firefox	https://addons.mozilla.org
Google Chrome	Chrome Web Store	https://chrome.google.com/webstore
Cytoscape	Cytoscape App Store	https://apps.cytoscape.org/apps
Amazon Alexa	Alexa Skills Store	https://www.amazon.de/skills

[*] Last Access: October 2021 [**] Last Access: May 2017

technical documentations that are available on the internet. Such documentations usually come in form of online manuals in developer portals, “how to" guides, and official instructions provided by platform providers as well as samples and demos. In addition, if enrollment in the ecosystems is made possible, we register as developer and examine the way that the ecosystems work. However, these activities are not yet helpful to gain a deep understanding of platform providers' business strategies. Therefore, we additionally explore platform providers' annual reports, e.g., using online sources such as *AnnualReports.com* and well-reputed business magazines as well as entries in questions-and-answers forums.

Conceptual-To-Empirical Iterations

In a conceptual-to-empirical iteration, the researchers conceptualize the dimensions without directly examining the objects. Thereby, a researcher “uses his/her knowledge of existing foundations, experience, and judgment to deduce what he/she thinks will be relevant dimensions” [NVM13]. Moreover, the characteristics need to logically follow the meta-characteristic.

In the conceptual-to-empirical iterations, we supplement our investigation with the implication from the literature regarding the meta-characteristics and the differences between the real-world ecosystems and the concepts addressed in the literature. We consider the most influential journal, conference, and workshop publications relevant to software ecosystems (e.g., [BB10a, Man16, JFB09, Bos09]) as well as the publications from the International Workshop on Software Ecosystems (IWSECO). The stepwise development of the variabilities is validated by examining ecosystems that are not in our list. Thus, we search for new information that can be extracted from the literature.

Process Termination

At the end of each empirical-to-conceptual and conceptual-to-empirical iteration, we check whether the ending conditions are met. In total, we performed eight iterations that comprise four empirical-to-conceptual and four conceptual-to-empirical iterations.

As already mentioned earlier in this section, we started with an empirical-to-conceptual iteration to exploit the architectural knowledge of existing ecosystems. Each empirical-to-conceptual iteration was followed by a conceptual-to-empirical iteration. The seventh iteration results in the conceptualization of no new characteristic or dimension. In this situation, researchers are allowed to terminate the process of taxonomy development. To ensure the validity of our decision regarding the termination of taxonomy development, we conduct another conceptual-to-empirical iteration as our eighth iteration. We continue inspecting the literature. This inspection results in identifying no new characteristic, which has not been included in the taxonomy. Therefore, according to the ending conditions, we terminate the process of taxonomy development at this point.

In total, our taxonomy development approach results in the identification of 25 variation points and their variants that are distributed across the main architectural aspects of software ecosystems (cf. Figures 6.2, 6.3, and 8.1). Complete lists of sources of the literature review and web links to the existing ecosystems can be found in our technical report [JZEK17b].

6.1.2 Variability Model: Variation Points and Variants

In this section, we present a variability model for architectural design decisions of software ecosystems. The variability model is the result of investigating existing ecosystems and literature performed in Section 6.1.1. For readability purposes, we present the variability model in three parts related to the main architectural aspects of software ecosystems introduced in Section 1.2, i.e., variabilities related to the social and organizational, business and management, and application and infrastructure aspects.

As shown in Figures 6.2, 6.3, 8.1, we use orthogonal variability model (OVM) notation [MPH$^+$07] for an enhanced expression of the variabilities. OVM is a language designed to describe architectural variabilities as first-class knowledge entities. Alternative ways are to use feature models or UML class diagrams. However, using these languages usually leads to ambiguity and loss of information since the concept of variability does not integrate well into such models [BRN$^+$13, BLP05].

In Figures 6.2 - 8.1, a triangle with a solid borderline is a mandatory variation point. A triangle with a dashed borderline is an optional variation point. Each rectangle represents a variant. A variant with a solid line variability dependency is a mandatory design choice whereas a variant with a dashed line variability dependency is an optional choice. There can be alternative choices among variants with optional variability dependencies. In the following, we elaborate on the three groups of variabilities and refer to the relevant real world examples from Table 6.1 .

Social and Organizational Variabilities

Figure 6.2 depicts variable architectural design decisions related to the social and organizational aspects of software ecosystems.

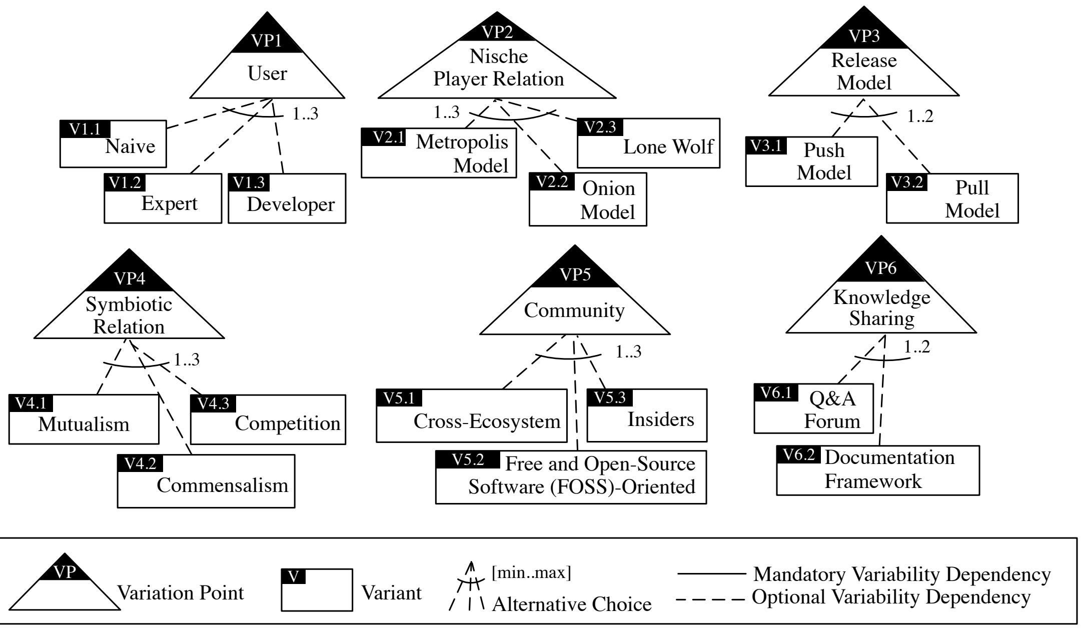


Fig. 6.2 SecoArc Model of Social and Organizational Variabilities

User: A user is a human actor, who uses a software product or service in the ecosystem. It is a key role in software ecosystems since the main part of architectural design decisions depends on the requirements of the target group of users. We distinguish among three types of users based on different levels of user expertise that are *naive user*, *expert*, and *developer*. A naive user does not require much experience or knowledge regarding the ecosystem products/services in order to use them. For instance, user mobile App ecosystems are considered as naive users since they can download and use the Apps on *Apple App Store* or *Google Play*, without having much prior expertise. An expert user has a high level of skills related to the services that are provided to him/her. This type of user emerges in form of individuals and/or enterprises. For example, *Salesforce* is a provider of cloud computing services. To use the services in *App Exchange*, the users require prior training or other forms of knowledge sharing mechanisms, such as documentation, that provide them with the knowledge on how to install and use the services. The third type of user in software ecosystems is developer. This is when an ecosystem provides source code as services and a developer is interested in using the source code, which is a common situation in open-source software communities. Examples of this type of user can be found in the *Programmable Web* and *GitHub* ecosystems. Notably, the variants of this variation point are concerned with the level of expertise expected from the users rather than the skills possessed by the users. For example, a person might be a naive user in a mobile App ecosystem while being a developer in another ecosystem.

Niche Player Relation: It concerns the social connections between the niche players. In a literature survey, Manikas et al. [MH13] identify three main models for the niche player relations: *Metropolis model*, *onion model*, and *lone wolf*. The metropolis model is when the platform provider is the business owner and policy-maker. The platform provider offers the fundamental functionalities of the platform. However, the platform is a basis for the third-party crowdsourced developments that provide a major value for a mass number of end-users [KC10]. An example of the metropolis model can be found in the *Apple* ecosystem, where a large number of third-party Apps are built upon the iOS platform. Furthermore, the onion model describes the situation, when there are several groups of third-party providers with different levels of importance for the ecosystem. The layers of onions are the metaphor that refers to the groups of providers. The providers on the internal layers are more critical for the ecosystem success [JSW11]. An example of the onion model is in the Eclipse ecosystem. The third-party developers are the less critical niche players while the members of Eclipse foundation are the key players. Moreover, the lone wolf approach is when a small group of providers has the major contribution. A lone wolf refers to a developer, who develops software alone and without working with other developers For instance, this concept is supported in the Ruby ecosystem by promoting the most popular third-party developments namely *gems* that are developed by lone wolfs. Lone wolfs are explicitly identified in the Ruby ecosystem [KJ11].

Release Model: This variation point refers to the way that the release of a new version of the platform is initiated. In this context, the platform provider might decide to publish a new version, which is referred to as *push model*. A push model is typically used in large ecosystems such as the *Android* ecosystem, where new versions of the platforms are regularly released. Another release model is when the users request a new version or new update for the current version, which is called a *pull model*. *Feature-requests*, *bug-reports*, and *pull-requests* are the exemplary ways that can initiate a pull-oriented release. For instance, pull-requests are made possible in the *OpenIntents* ecosystem. In some cases, the users are only in contact with third-party providers. Thereby, users' pull requests indirectly reach the platform provider via the third-party providers. In addition, the users may be interested in an older version of the platform, which is a barrier to pull the new version [RS12].

Symbiotic Relation: Symbiotic relation refers to the relationships that facilitate the co-existence of entities in the ecosystem, which is inspired by the ecological ecosystems. In the context of software ecosystems, an entity is a human actor or a technical element such as software or hardware. Three types of symbiotic relations are known to be relevant for software ecosystems: *Mutualism*, *commensalism*, and *competition*. Mutualism is when a relationship is beneficial for both sides of that relationship. Mutualism is common when the third-party developments are open-source so that the third-party providers can use each other's components. This is the case in *F-Droid* and *Apkpure* ecosystems, where open-source Android Apps are distributed. Commensalism refers to a situation when one side of a relationship benefits and the other side is unaffected, e.g., when a third-party developer clones the source code of a project from a *GitHub* repository. Competition is concerned with a situation when both sides are harmed. This is a relevant situation for ecosystems with commercial third-party developments like the *AWS* ecosystem, where the developers have to share their market of users [YRB08].

Community: Community is the general characterization of groups of the actors that are involved in software developments in the ecosystem. It defines the limits that are imposed on the regulations and social interactions between those actors. In a *cross-ecosystem community*, the third-party providers apply their knowledge in different ecosystems. Developers of Android Apps are an example of this community. In addition to *Google Play*, which is the official marketplace for the Android Apps, there are multiple third-party stores for Android Apps (cf. Table 6.1). There is a range of reasons that can hinder this situation, e.g., an important one is due to licensing reasons or when the knowledge is very domain-specific and cannot be applied outside of an ecosystem. Furthermore, free and open-source software (FOSS)-oriented community refers to a community that follows the rules of free and open-source software that are well-established and pre-defined in software development communities [Sca07]. In the FOSS community, entrance to the ecosystem solely depends on the third-party provider's decision since the platform provider has minimized the entrance barriers. An example of a FOSS community is around the *Mozilla Firefox* platform. In another situation, third-party providers become insiders by fulfilling certain entrance requirements. *SlideMe* is a provider of third-party Android Apps. In the *SlideMe* ecosystem, an insider community comprising Android device manufacturers exists. Such manufacturers need to submit their requests for a partnership. In case of acceptance, they can enter the ecosystem as suppliers.

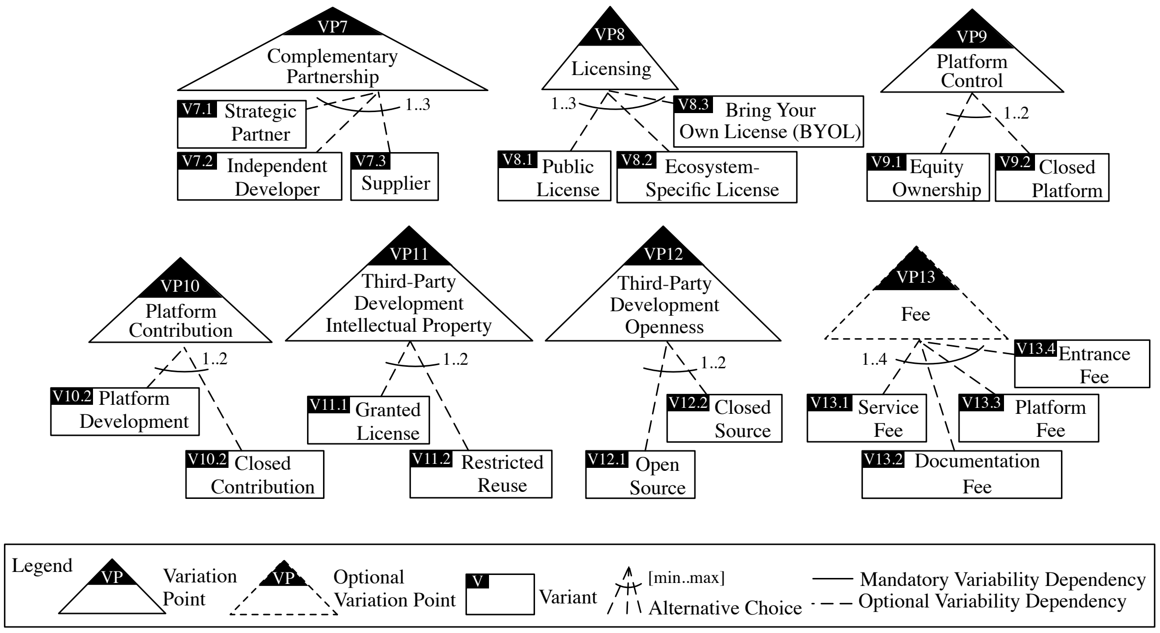


Fig. 6.3 SecoArc Model of Business and Management Variabilities

Business and Management Variabilities

Figure 6.3 shows the variabilities of business and management design decisions.

Complementary Partnership: Complementary partnership specifies, who supports the platform provider during the service provision in the ecosystem. To this end, the complementary partnerships provide value-adding solutions. A complementary partnership can be a *strategic partner*, *supplier*, or *independent developer*. A strategic partner is a long-term partner with access to the platform such as *IBM*[1] being a partner of *GitHub* by providing cloud-based solutions on the basis of the GitHab platform. A supplier (aka integrator) is responsible for providing a specific software or hardware resource or integrating the resources into the ecosystem, such as *Samsung*[2] manufacturing mobile devices for the *Android* platform. An independent developer develops third-party applications on top of the platform without having a direct partnership with the platform provider [EB12]. Providers of *Amazon Alexa Skills* are mostly developers, who develop them independent of each other. A human actor can be a user and a complementary partnership at the same time. An instance of this situation is when a developer uses the available code and produces new code in an ecosystem.

[1]`https://www.ibm.com`. Last Access:January 10, 2022.
[2]`https://www.samsung.com`. Last Access:January 10, 2022.

Licensing: Licensing refers to the legal rules governing usage and redistribution of software. An ecosystem may use a *public license* such as GNU General Public License (GPL). GPL allows free usage, execution, and altering the source code. Another licensing option is to introduce an ecosystem-specific license, e.g., Apple providing *Standard Apple EULA*. On one hand, the choice of licensing usually conforms to end-user license of agreement. On the other hand, ecosystems may allow external developers to freely decide to make their source code available under a different license [AAS09]. Another option is to allow for *bring your own license (BYOL)* such as the *AWS* ecosystem. BYOL enables third-party providers to sell their software under their own licenses in the ecosystem. In this case, the platform provider might additionally decide to offer an ecosystem-specific license too, but to give the third-party providers the freedom to choose between their own licenses and the ecosystem-specific one.

Openness policies are a set of business strategies in software ecosystems that determine whether the contents and methods of the ecosystem are subject to access or change by the third-party provider. Software ecosystems can hardly be judged as completely open or closed [JBSL12]. In the following, we discuss four main variation points that are used by platform providers to define fine-grained openness policies, i.e., *platform control*, *platform contribution*, *third-Party development intellectual property*, and *third-Party development openness*. These variation points are based on the openness strategies introduced by Boudreau [Bou10].

Platform Control: Platform control is a policy related to the openness of the software platform. It specifies the situation when the platform provider grants equity ownership of the platform to the third-party providers. Specifically, the third-party providers have access to the source code of the platform. Furthermore, changes to the source code are allowed. Hence, the providers are given control over (part of) the platform. For instance, the members of *Eclipse Foundation* have platform control rights that enable them to collaboratively develop new features for the Eclipse platform.

Platform Contribution: This variation point is used to specify another policy related to the platform openness that is a situation when the third-party providers are given the permissions to contribute to the development of the platform. The third-party providers' contributions are integrated into the platform under the observation of the platform provider. Thereby, the third-party providers add value to the platform without having equity ownership. An example of this situation can be found in the *Mozilla Firefox* ecosystem. Using *Firefox Developer Edition*, independent developers can contribute in the platform development by writing code, fixing bugs, or making certain add-ons.

Third-Party Development Intellectual Property: It concerns the openness of the third-party developments. It refers to the situation when the platform provider grants the third-party providers licenses for their developments that enable them to publish and reuse their developments in other ecosystems. If the reuse is restricted, the third-Party development can only be distributed inside the ecosystems. For instance, among others, *BSD*, *MIT*, and *Apache license* can be chosen for software projects in the *GitHub* ecosystem. All these licenses are under the category of public open source licenses and allow anyone to use, modify, and share the software [Git21].

Third-Party Development Openness: It specifies whether the third-party developments are open-source or not. In some ecosystems, platform providers are in charge to specify this policy while, in some other ecosystems, the third-party developers decide whether to make the source code of their developments open.

Fee: Fee is a means for platform providers to protect intellectual property by introducing different degrees of payments for the users and third-party providers. This has a direct relation to the different degrees of access permissions to the platform that is granted to an actor. We identified four main types of fees that are demanded in software ecosystems, i.e., *service fee*, *documentation fee*, *platform fee*, and *entrance fee*. A service fee refers to the costs that are associated with using third-party developments such as buying commercial mobile Apps or using paid cloud solutions in *Salesforce AppExchange*. A documentation fee refers to the costs entailed on third-party providers for using technical and marketing documentation shared by the platform provider. A platform fee is associated with the usage of the platform, which can be imposed on the users and third-party providers. For instance, using *AWS* is subject to an on-demand payment for the users. Examples of documentations are the source code documentation and user manuals. In addition, an entrance is related to the fee of entering the ecosystems. A platform provider may decide that third-party providers, who wish to publish their applications on the store, may need to pay a certain amount of periodic or one-time entrance fee [VAKJP11].

Application and Infrastructure Variabilities

This view presents architectural decisions concerning software development on the basis of platforms and the provision of software to the users. Figure 8.1 shows

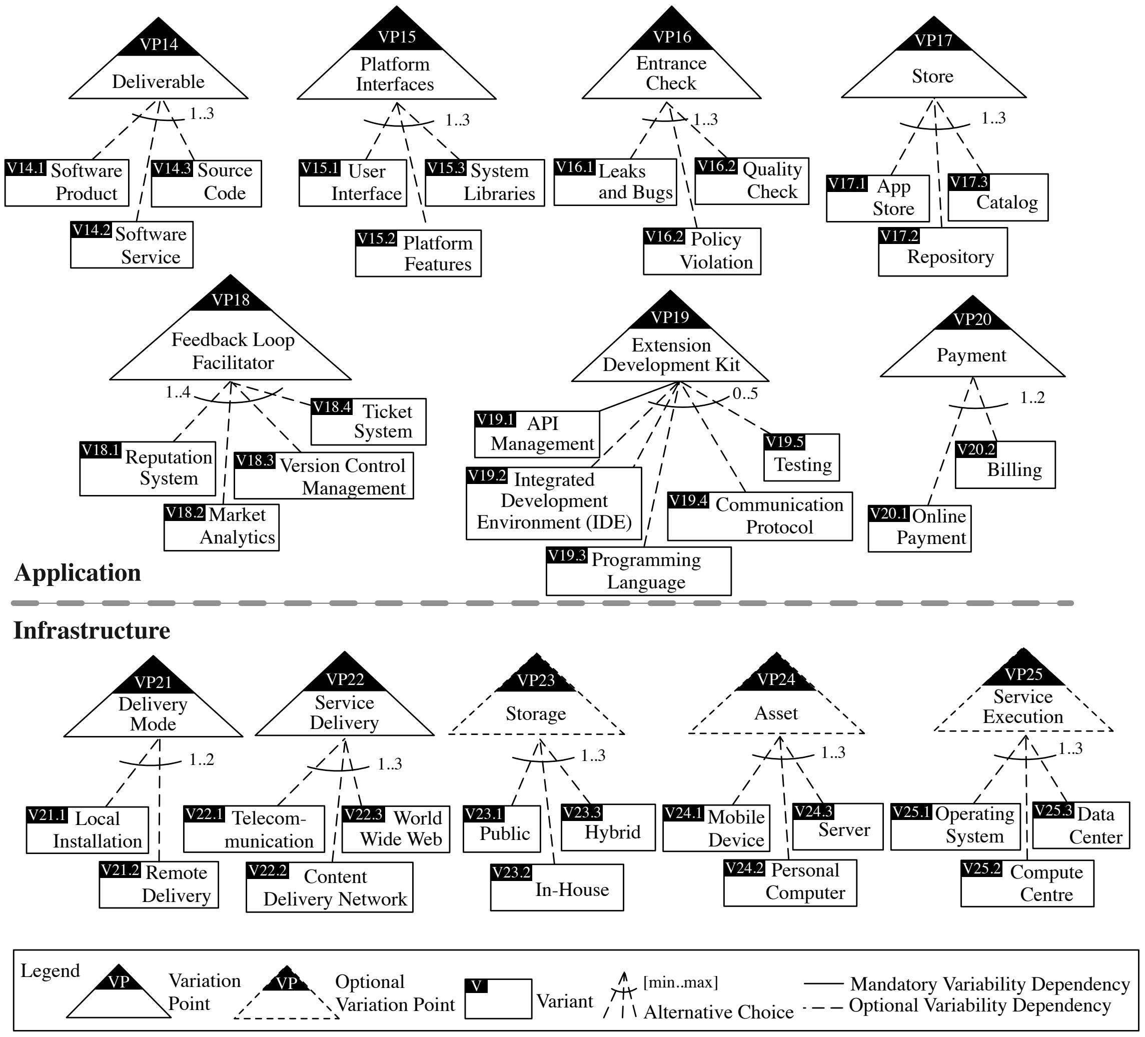


Fig. 6.4 SecoArc Model of Application and Infrastructure Variabilities

the variabilities related to the application and infrastructure aspects. For readability reasons, the variabilities are presented separately in two different groups. In general, the application-related variabilities concern the design decisions that enable the third-party providers to develop applications on top of the platforms whereas the infrastructure-related variabilities refer to the decisions that are necessary to realize functions of the application view. This includes delivery, deployment, and operation of software solutions at the user side. In the following, we present the variabilities.

Deliverable: Deliverable refers to the type of software artifact that is delivered to users. This can be software products, software services as well as source code. We distinguish between software products, e.g., mobile Apps, *Eclipse* plug-ins, and *Google Chrome* extensions that are installed on local devices, and software services, e.g, *Salesforce*, *AWS*, and *QuickBooks* cloud services that are executed on a remote server. Deliverables accordingly address different target groups of users, e.g., software developers, users of mobile devices, and enterprises.

Platform Interface: A platform interface is a gateway using which third-party providers can access the platform and develop software on top of it. The architecture of the software platform needs to provide the right modularity and granularity so that the external developers become able to correspond to relevant software components [Bos10]. The variability model recognizes different groups of components as follows: *System library*, *graphical user interface*, and *platform feature*. A system library provides access to core functionalities of platforms, e.g., access to OS kernel, memory management, and resource sharing. A graphical user interface facilitates accessing and extending functionalities of a user interface. For instance, a gaming mobile App requires interaction with the graphical user interfaces of a platform. Furthermore, there are direct interfaces to the built-in capabilities of software platforms. We refer to such capabilities as a platform feature. Platform features are specific to a certain platform or domain that developers often require to work with them, e.g., GPS, camera, and audio control in mobile OS platforms. Essentially, platform providers need to keep the interfaces consistent during ecosystem evolution. This includes suitable change propagation and update management among platform components and third-party applications [CH10].

Entrance Check: Entrance check includes a wide range of security and policy checking functions to protect ecosystems from malware, unwanted actions, and misuse. A typical way to realize such checks is to define a review process for the third-party providers, who would like to enter the ecosystem. The review process mainly considers *leaks and bugs*, and *policy violations* of externally developed applications before being published in stores. In addition, a *quality check* refers to the assessment of third-party providers. An example of this situation is when the platform provider wants to choose strategic partners.

Store: This variation point specifies an online platform that is used to distribute the third-party developments in the ecosystem. As discussed in Chapter 5, store is a basic software element that usually exists in the ecosystem architecture. It acts as an intermediary between the users and the providers. Our variability analysis reveals three variants for store in software ecosystems, i.e., *App store*, *repository*, and *catalog*. The term *App store* became popular by the launch of mobile App stores like the Apple App Store and Google Play [MSJ$^+$17]. Today, it is generally used to refer to a marketplace with the characteristics similar to the mobile App stores, e.g., including search, rating, and ranking functions. A store can even be a repository, where third-party developments and their source code are stored. To this end, the source code can be directly accessed by the users. F-Droid[3] uses this kind of store to share free and open source Android mobile Apps. Furthermore, we use the term *catalog* to refer to a list of software that platform providers publish on their websites to announce the available third-party developments. In this case, the developments are not directly accessible using such a list. An example of catalogs is OpenIntents[4].

Feedback Loop Facilitator: A feedback loop facilitator refers to software features that are used in software ecosystems to enable the users of services and the providers to communicate knowledge regarding the quality of services. Reputation systems are in the architecture when the market of services is large. As mentioned in Chapter 5, a reputation system includes rating, reviewing, and ranking features and is used in online stores such as mobile App stores. Furthermore, market analytics functions inform the third-party providers about quality of user experience. Examples of the market analytics is repository mining techniques used in App stores. Thereby, the partners can stay connected with their customers and track their market growth. A feedback loop between the developers, who commit to the code, and the ones, who reuse it, is created using version control management and ticket system features. A version control management, e.g., version control used by GitHub, helps communicate changes made in the source code among the developers. Additionally, using a ticket system, developers report bugs and issues in the source code to the platform provider.

Extension Development Kit: It includes techniques and tools that enable third-party providers to develop applications on top of the platforms. Integrated development environment (IDE), programming language, communication protocol, and testing functions are the various architectural components that a platform provider may include in the design of an ecosystem. In addition, it is interesting that all ecosystems, which we examined during the development of the variability model, include software a certain kind of API management in their architecture, for example in terms of software development kits (SDKs). Therefore, we define API management as a mandatory variant. For instance, iOS Apps can be developed using the *Xcode* IDE, by means of the *Swift* programming language. Xcode contains *iOS SDK* and testing frameworks. Moreover, by including Wikis and forums, social communications among

[3]`https://f-droid.org/packages`. Last Access:January 10, 2022.
[4]`http://www.openintents.org/download`. Last Access:January 10, 2022.

independent developers are supported [SEL14]. In addition, an ecosystem may enable the composition of external applications. For instance, Intents[5] allows developers to compose Android Apps.

Payment: Payment refers to a set of software functions that allow fees to be paid in the ecosystem. Fees can be paid directly using online transactions such as *Paypal*[6]. Another variant is to allow fees to be paid per bill. Thereby, bills should be generated and sent to a corresponding person/enterprise. Payments via billing can be often later performed using online transactions like online bank transfer as well.

Delivery Mode: Delivery mode determines how deliverables are provided to the users. This can be using a local installation of executable files or the remote execution of software services. For instance, network visualization Apps in *Cytoscape App Store* and most of the *Eclipse* plug-ins require a local installation on a device. Examples of remote delivery are cloud computing services in the *Salesforce, Amazon Alexa, and AWS* ecosystems or remote procedure calls to web services in the *Mashape* and *ProgrammableWeb* ecosystems. A hybrid approach is applied when both variants are applied in an ecosystem , e.g., Apple provides Apps that demand a local installation on Apple devices and iCloud services that are excuted remotely.

Service Execution: It implies suitable infrastructure for the execution of deliverables. It is an optional variation point, i.e., in some ecosystems, e.g., where the deliverable is the source code, no execution of deliverables is needed. While in other ecosystems, e.g., mobile ecosystems, applications require an operating system to be executed. Furthermore, service provision using cloud and web services requires the support of distributed compute and data centers.

Service Delivery: It includes the technologies that facilitate the delivery of deliverables to the users. This can be exclusively performed on the basis of the World Wide Web (WWW) or by the support of infrastructure suppliers, i.e., providers of telecommunications and content delivery network services. External suppliers may be employed to handle a high-performance service delivery in different networks like the intranet, extranet, and internet. For instance, results of activating an Alexa Skills is transmitted over the web while the tasks related to executing the skills are handled by AWS.

[5]`developer.android.com/reference/android/content/Intent.html`. Last Access: January 10, 2022.
[6]`https://www.paypal.com`. Last Access:January 10, 2022.

Storage: It determines where users' personal data are stored when needed. Examples of users' data are locations, calendars, financial information, and so on. For this purpose, three variants are used by platform providers. Firstly, public storage provided by other companies can be used to save costs. Another variant is to use in-house servers to store the data while a combination of these both can be applied in an ecosystem.

Asset: It refers to a device or a set of devices provided by platform providers to realize ecosystem deliverables on the user's side. Asset provision is an optional variation point, which can be delegated to complementary partnerships too. Three types of assets can be recognized, i.e., mobile device, personal computer, and server. The choice of an asset depends on the deliverable. Smartphones and tablets are examples of mobile devices that are used to run mobile Apps. Further examples are the *Cytoscape* networking Apps that are executed on personal computers and servers.

While the goal of the variability model presented in this section is to assist platform providers in dealing with architectural variabilities of software ecosystems, it is important to note that the variability model is subject to different realizations depending on platform providers' business strategies and technical context. The different realizations are originated by the contextual information, which comes from the context and domain of an enterprise or software project. The contextual information determines why and how a variability is realized in real-world.

6.1.3 Variability Dependency Constraints

The variability model introduced in Section 6.1.2 includes key dependency constraints between the variation points and variants. Violation of the dependency constraints results in contradictory decision-making, which can lead to design deficiencies. In this section, we discuss the main variability dependency constraints that we identified in our study.

If a variation point (VP_i) / variant ($V_{i.j}$) *requires* another variation point (VP_m) / variant ($V_{m.n}$) in order to be realized, then VP_m / $V_{m.n}$ needs to be included in the architecture once VP_i / $V_{i.j}$ is included. Moreover, if VP_i / $V_{i.j}$ *excludes* VP_m / $V_{m.n}$, then VP_m / $V_{m.n}$ needs to be excluded from the architecture if VP_i / $V_{i.j}$ is included. The dependencies constraints can be associated with four main topics, i.e., service, user, openness, and supplier. In the following, we elaborate on the constraint dependencies.

Service-Related Constraints

First of all, we notice that there is certain interplay between deliverable (VP15), delivery mode (VP21), service execution (VP25), and asset (VP24). The type of deliverable specifies the delivery mode. Specifically, delivery of software products (V15.1) and source code (V15.3) requires local installation (V21.1) on an asset on the users' side. In this case, the users download and install the software products on the assets and the developers download and use the source code in their development environment. Moreover, software services (V15.2) require remote delivery (V21.2). In addition, the remote delivery (V21.2) demands hardware and software resources for the service execution (VP25).

User-Related Constraints

The type of user (VP1) has certain implications for the architecture. Firstly, a naive user (V1.1) excludes the deliverable being source code (V13.3) since working with the source code demands certain programming expertise. Similarly, a naive user usually owns a mobile device or a personal computer. Thereby, being a naive user (V1.1) excludes that the asset is a server (V24.3) as installing software products on a server demands advance knowledge, which is not common for this group of users.

Openness-Related Constraints

Some dependencies are related to the business strategies and their relations to the architecture. Deciding for a FOSS-oriented community (V5.2) excludes BYOL (V8.3) because it takes the ownership rights away from the third-party providers. Thus, the licenses granted to the third-party developments can not be used outside of the ecosystem. Furthermore, if the source code of the platform is closed (V9.2), the platform provider needs to provide platform interfaces (VP15) so that third-party developments can be developed on the basis of the platform. Furthermore, some other dependencies concern the openness of the third-party developments. Closed source third-party developments (V12.2) enforce the necessity of checking bugs and leaks (V16.1) before they are published in the ecosystem. In addition, if fees (VP13) are defined for the entering the ecosystem or the usage of platform, service, documentation, etc., then payment (VP20) should be included in the architecture.

The variabilities of the social view enforce the following constraints: If the ecosystem possesses a FOSS community (V5.2), then the fee (VP13) needs to be excluded from the architecture. Furthermore, a FOSS community excludes closed platforms (V9.2) while it requires the rights for platform development (V10.2), granted license (V11.1), and open-source third-party development (V12.1) since these design decisions form the principles of a FOSS community [CF07, KHMA12].

Supplier-Related Constraints

Furthermore, in the empirical-to-conceptual iterations discussed in Section 6.1.1, we notice that a group of dependencies frequently appears in the existing ecosystems although we do not consider them as a kind of constraints, which must be fulfilled, because the violation of these dependencies does not lead to any design deficiency. Service execution (VP25), telecommunication (V22.1), content delivery network (V22.2), and asset (VP24) are usually provided by suppliers (VP7.3) in the ecosystems. This is due to the fact that platform providers are normally software providers, who wish to build an ecosystem around their software platforms. Providing all resources required for manufacturing assets and infrastructure to deliver and execute the services, if possible for the platform providers, is very costly [BB10a].

6.1.4 Relations to Ecosystem Architecture

An important applicability factor of the variabilities is to understand their relations to design artifacts [PBv05, Chap. 6]. In this section, we explore the relations of the variation points of the variability model developed in Section 6.1 to the main design elements of software ecosystems. Figure 6.5 presents these relations. In the middle, high-level components, i.e., software platform and store, human actors, and infrastructure are portrayed. The human actors, i.e., the platform provider, users, third-party providers, interact with these elements. Furthermore, the triangles represent groups of variations points and their relations to the rest of the architecture. A relation shows to which part of the architecture a variation point is applied.

The variation points on the left side of Figure 6.5, i.e., deliverable, and delivery mode, are applicable to the users' side interfaces, which define what to deliver as products/services of an ecosystem and how to deliver it. In addition, using the variation points applied on the infrastructure, i.e., storage, service delivery, and service execution, the platform provider can specify for which purposes hardware and software resources are used in the ecosystem.

On the top of Figure 6.5, third-Party development openness and intellectual property are applicable when the third-party providers publish on the store or when the users download the third-party developments from the store. The type of user and com-

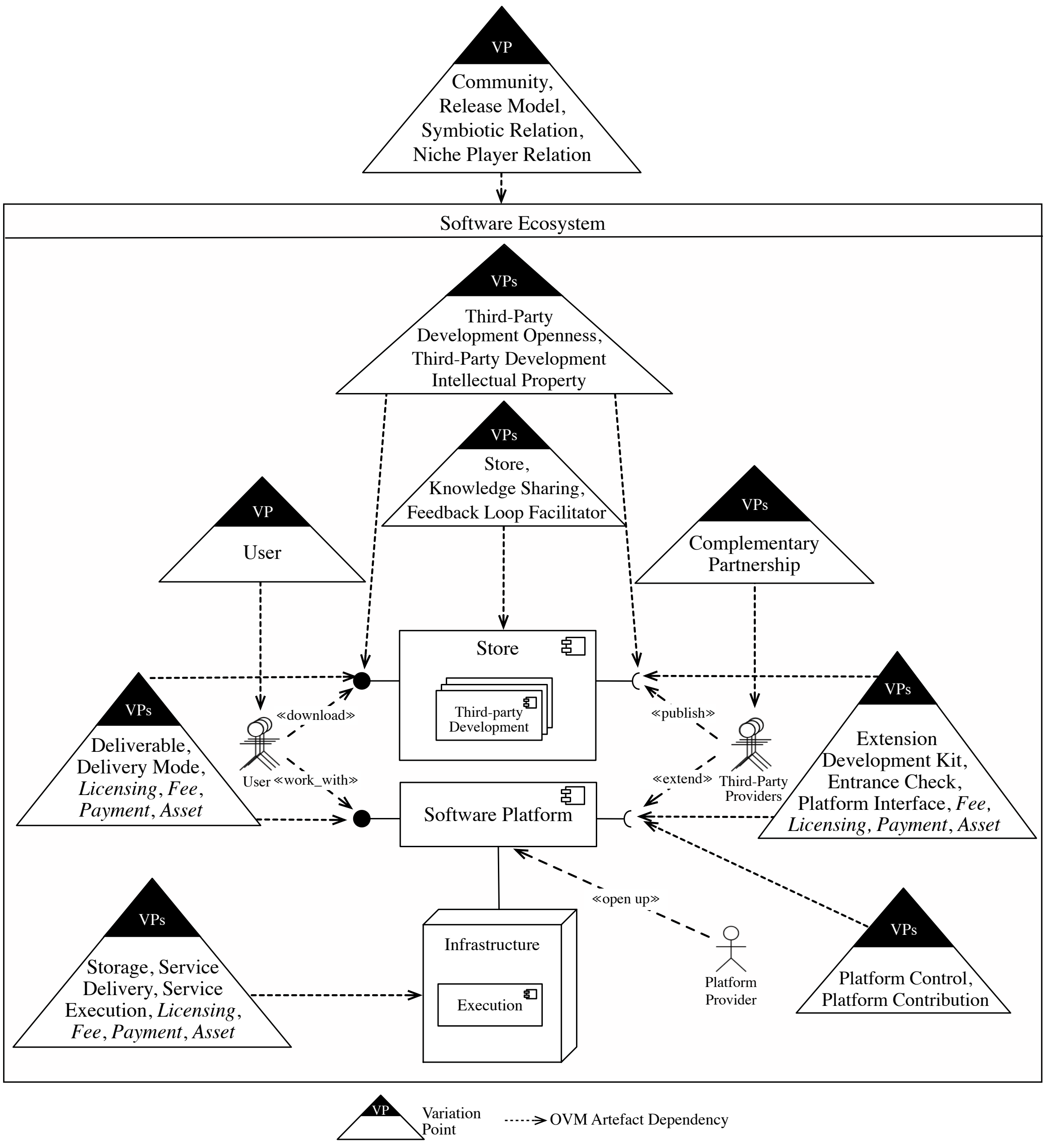


Fig. 6.5 Relations of the Variation Points to Ecosystem Architecture

plementary partnership are respectively applied on the users and third-party providers. A store can be differently specified by using the store, knowledge sharing, and feedback loop facilitator three variation points. By defining the type of store, i.e., App store, repository, or catalog, the platform provider can further specify relevant variants of feedback loop facilitator and knowledge sharing. For instance, deciding for a repository turns a ticket system and version control management to further relevant design choices.

Furthermore, the variation points on the right of Figure 6.5 are applicable to the interfaces on the third-party providers' side. This includes the extension development kit, entrance check, and platform interfaces. Using these variation points and the platform openness policies (i.e., platform control and platform contribution), a platform provider enables the third-party providers to develop software on the basis of the platform. In addition, using these variation points, the platform provider can realize openness policies at the technical level. For instance, the entrance check is related to the store and platform interfaces. Such entrance check usually appears in form of a review process to ensure the quality of third-party developments.

Moreover, different applications of *licensing*, *fee*, *payment*, and *asset* can vary the ecosystem architecture in different ways since these variation points are applicable at several places in the architecture. These include the interfaces related to the users, third-party providers, and the infrastructure. Choices regarding the fee define whether and how the users are charged for platform usage, third-party applications, or for publishing on the store. Different fees might need to be paid using a different payment method. Introducing ecosystem-specific assets affects platform usage and third-party applications. This highly influences third-party providers' work with respect to the choices of an extension development kit.

Last but not least, the specifications of community, niche player relation, release model, and symbiotic relation variation points affect the whole ecosystem because they are the high-level characterization of the interactions among the human actors and their interactions with the architectural elements.

6.1.5 Relations to Architectural Commonalities

In Chapter 5, we presented the architectural commonalities of software ecosystems. The core part of the architectural commonalities were 24 primary features and their sub-features. Our study showed that these were the most addressed architectural features of software ecosystems in literature. This knowledge has been complemented with the variability model identified in this chapter that is aligned with our goal to obtain a knowledge base for software ecosystems.

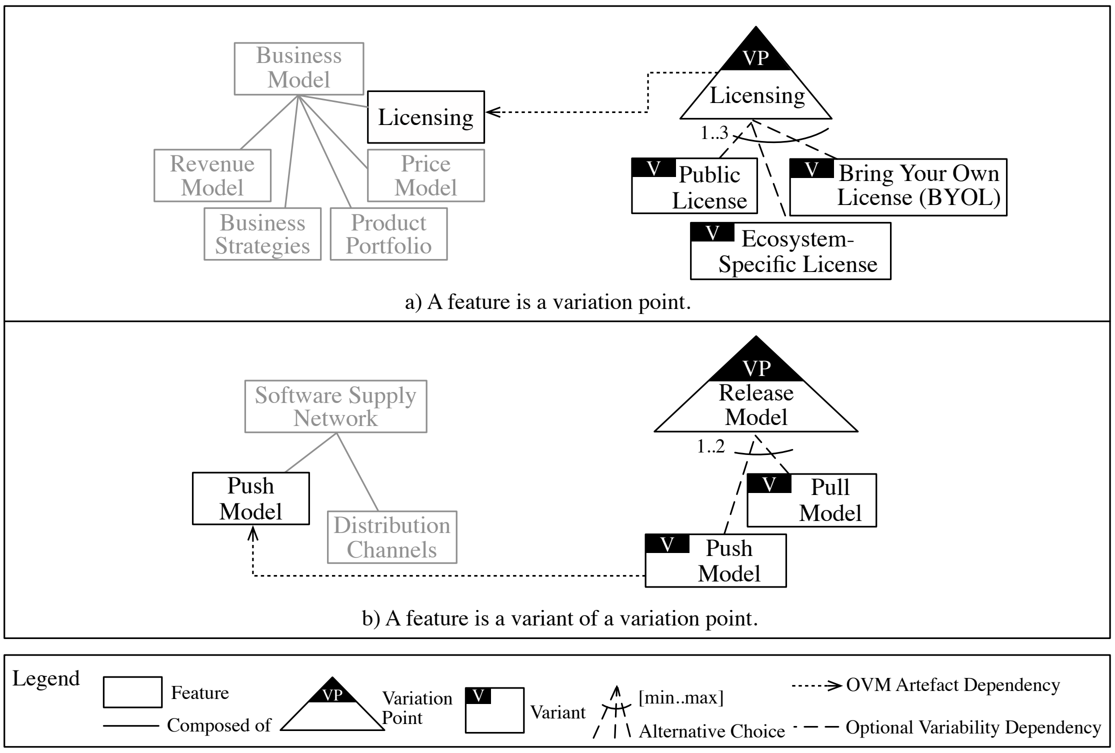


Fig. 6.6 Two Types of Relations between Architectural Commonalities and Variabilities

Specifically our study of variability reveals two types of relations between the variabilities and the primary features:

(a) A feature is a variation point. Now, we know its variants.

(b) A feature is a variant of a variation point. Now, we know the other variants and how to deal with them during the decision-making.

The first type of the relation denotes that a feature can be realized in different ways specified by a variability. Among the primary (sub-)features, *licensing*, *knowledge sharing*, *API management*, *store*, and *openness policies* are further refined by the knowledge of the variabilities. Figure 6.6 shows an example of this situation. Licensing has been identified as a sub-feature. The variability *licensing* complements this information by introducing three variants of licensing that are mostly used in ecosystems, i.e., *public licensing*, *ecosystem-specific license*, and *BYOL*.

The second relation type refers to a situation, where the variability model provides some complementary information on possible alternative features for a primary feature, which can be used in a specific application domain or context. *Rating*, *ranking*, *reviewing*, and *push model* possess this type of relation with the variabilities. For instance, the model of the primary features reveals that *push model* is used for software supply networks in software ecosystems as depicted in Figure 6.6. The knowledge of variabilities indicates that an alternative design decision for *push model* would be to use a *pull model*.

6.2 Service Provision Scenarios in Software Ecosystems

In the previous sections, we identified a variability model for architectural design decisions of software ecosystems. As mentioned in Section 6.1.2, the variability model can be realized very differently by platform providers. Aggregation of different realizations of the variabilities can result in very different ecosystems. To provide a general understanding of how software ecosystems may work as a result of different design decisions, we identify alternative service provision scenarios in software ecosystems. In this context, a scenario provides an overview of the sequence of key interactions between the actors that lead to provide software services and products in the ecosystems. We identify three scenarios on the basis of expert interviews that we conduct in Section 6.2.1. In Section 6.2.2, we present the scenarios and elaborate on their variants.

6.2.1 Expert Interview

This section describes the setup of ten semi-structured expert interviews that we have conducted from May 2016 to July 2016. A *semi-structured interview* is a combination of a predefined questionnaire, to extract specific knowledge from interviewees, and open-ended questions, to obtain unforeseen information on the basis of discussions during the interview. Furthermore, a person is considered as *expert* if the person has a high level of knowledge or skill in a specific field that is related to the subject of the interview [HA05, Sea99].

We conducted the interviews with the team of CRC 901 On-The-Fly Computing. CRC 901 On-The-Fly Computing is a research project that concerns the design and development of on-the-fly computing markets. The interviews have been done with 11 sub-projects (the number of the sub-projects belongs to the time that the interviews have been performed). Each interview is performed in a group of three to four persons including two interviewers. The interviewees are grouped based on their working areas. In total, 22 persons are interviewed. Each interview session has taken 90 to 120 minutes time.

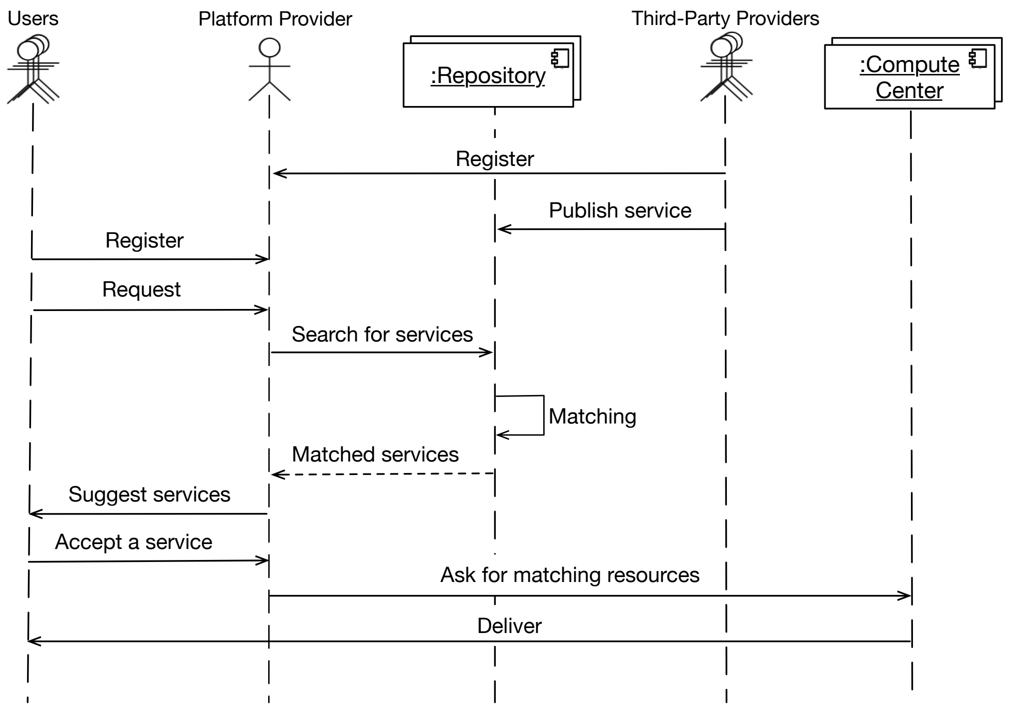


Fig. 6.7 Service Provision in Open Software Ecosystems

The interviewees are mainly researchers, who have depth knowledge of the on-the-fly computing markets. We have chosen the interviewees based on their experience and the amount of time being active in the project. First, we develop a questionnaire to collect information on the actors of the on-the-fly computing markets, e.g., how the third-party providers can participate in the markets. The questionnaire can be found in Appendix A. Additionally, we start open-ended discussions on service provision scenarios by asking how the actors might alternatively interact with each other.

6.2.2 Service Provision Scenarios

In this section, we elaborate on three scenarios that are the result of conducting the interviews in the previous section. We call the three scenarios *Open ecosystem*, *in-house ecosystem*, and *semi-open controlled ecosystem*. In addition, we discuss the variants of each scenario that is a slightly different realization of the scenario. While this is not a complete list of all possible scenarios of software ecosystems, the interviews show these scenarios dominate the ways how the ecosystems are realized.

Open Software Ecosystem

An open ecosystem describes a situation when the users and third-party providers freely enter the ecosystem and use the ecosystem resources. Figure 6.7 depicts the general scenario of service provision in open ecosystems. Both the users and third-party providers should *register* into the ecosystem before entering it. The registration is handled via a registration portal that is provided by the platform provider for this purpose. An example is when the users are asked to create an account on a website before being able to download the services. Furthermore, the third-party providers *publish* their services on a *repository*, which belongs to the platform provider. After entering the ecosystem, the users can submit service requests. Upon users' *request*, the platform provider *searches for the services* among the available ones on the repository. Afterward, using a *matching* function, the most suitable services are identified. In Section 5.3, we discussed that service matching is a part of the recommendation system. A list of *matched services* is sent to the platform provider. The platform provider *suggests the services* to the users. Once a user *accepts a service*, the platform provider *asks for resources* that are provided by compute centers to handle *delivery* of the service.

Several variants of the scenario described above are considerable. In the following, we elaborate on some of these variants. In some open ecosystems, repositories are directly accessible by the users. In this case, platform providers play the role of regulators by providing licenses and fostering the repositories. Furthermore, service delivery can be in terms of delivering the results of execution to the users or downloading the services and execute them on their local devices by the users. In addition, as mentioned in Section 6.1, compute centers can be external resources provided by suppliers that handle the task of service delivery. In addition, Moreover, although it is not very common, some platform providers allow the users and third-party providers to enter the ecosystem without any registration.

In-House Software Ecosystem

An in-house ecosystem describes a closed ecosystem of interacting actors, where the third-party providers are qualified by the platform provider to enter the ecosystem. Thereby, the platform provider forms a network of carefully selected third-party providers. Such strategic selection of third-party providers is needed when the services are a very domain-specific functionality related to certain use cases or industrial areas.

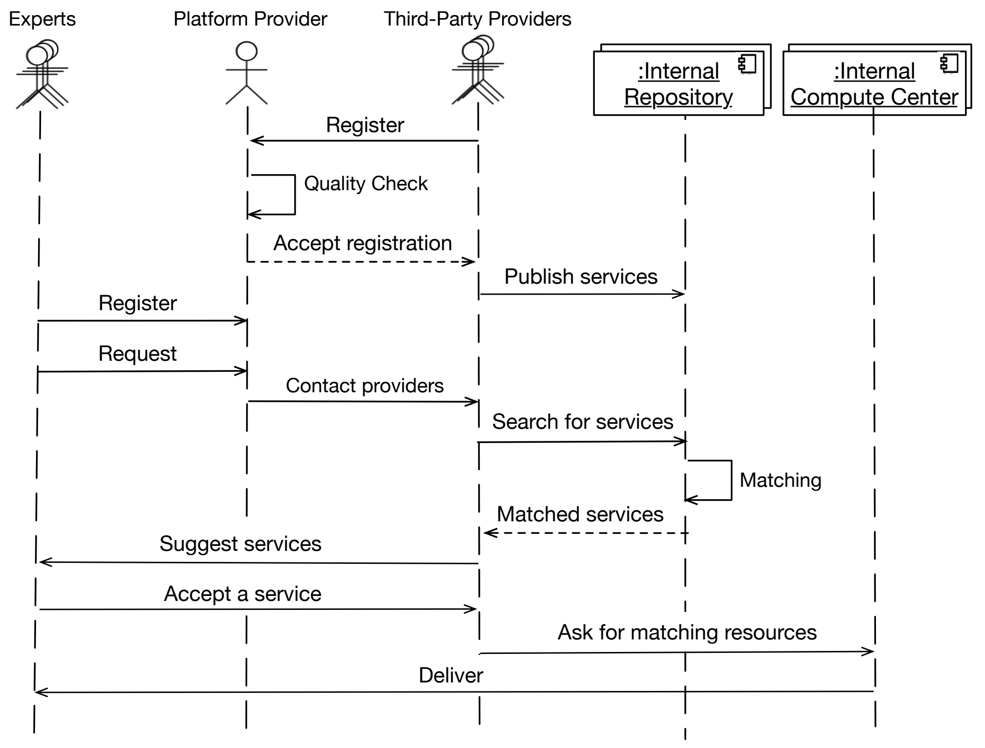


Fig. 6.8 Service Provision in In-House Software Ecosystems

In addition, the users, in this case, are domain experts, who seek professional service provision in their field of activity.

Figure 6.8 demonstrates the main interactions between the actors during the service provision in in-house ecosystems. Once the third-party providers decide to *register* with the platform provider, the platform provider considers the registration by performing *quality check*. A difference to the open ecosystem is that the third-party provider publishes the services on an internal repository that is not necessarily shared with the platform provider. Once the experts *register* in the ecosystem and *request* services, the platform provider *contact the third-party providers*, who provides services relevant to the requests. The task of service provision follows by finding, suggesting, and delivering services to the experts, which, in the case of the in-house ecosystems, are handled by the third-party providers.

A variant of the in-house ecosystem scenario is when the experts need to pass some quality checks. Such experts are usually enterprises, which require enhanced software for their customers. An example of quality checks is when the platform provider expects that the enterprises own a certain amount of the market share. Furthermore, for security reasons, the experts might want to execute the services on their local deceives. In this case, the compute centers reside on the experts' side. Often consulting services provided by the third-party providers help the experts with the local installation of the services. Additionally, the internal repository may belong to the platform provider if the third-party providers define the third-party development openness as open (cf. Section 6.1).

Semi-Open Controlled Software Ecosystem

While the scenarios discussed above are two extreme situations, where an ecosystem is completely closed or open, the semi-open controlled ecosystem falls in between these two. A large number of ecosystems are semi-open controlled, where platform providers use different levels of access control policies to achieve a combination of limited and open access for different actors of the ecosystems [JBSL12].

Given that a semi-open controlled ecosystem can be a transformation of an open ecosystem if more constraining rules are applied or a transformation of an in-house ecosystem when more enabling rules are applied, the transformation can be characterized in two different directions: a) Considering enabling / constraining rules for the third-party providers and users and b) introducing different levels of institutional environment in the ecosystem.

On one hand, by defining enabling/constraining rules for the third-party providers and users, all the providers and all the users are affected in the same way by the rules. This causes that the ecosystem as a whole becomes more open or closed depending on the type of the imposed rules. Although a wide range of rules can contribute to making the ecosystem opener or closer, the quality of openness remains the same for all the providers and the users.

On the other hand, by introducing different levels of institutional environment in the ecosystem, new dimensions of complexity are added to the ecosystem, which essentially affects the range and complexity of openness policies. Different levels of the institutional environment are created when the third-party providers are grouped and each group experiences a different degree of openness in the ecosystem. For instance, partner directories are a good example of categorizing partners based on different levels of privilege and promotion in the ecosystem.

In summary, the identified scenarios mainly differ in terms of openness for the third-party providers and users. This shows how determinant the openness variation points (VP9-VP12) for the design of software ecosystems are. In addition, in the open ecosystem scenario, the third-party providers work very decoupled from the platform providers while, in the in-house ecosystem scenario, the third-party providers and platform provider handle service provision in cooperation with each other.

6.3 Summary and Scientific Contributions

In this chapter, we identified architectural variation points and their variants for software ecosystems by extracting the knowledge from the literature and from the fragmentary information that is available on existing ecosystems by using a taxonomy development research method. We clarified the variabilities dependency constraints and the relations between the variation points and the design artifacts of software ecosystems. Furthermore, we provided an overview of alternative service provision scenarios and their variants that we obtained by performing semi-structured expert interviews. This knowledge enables platform providers to systematically deal with the architectural variabilities while creating software ecosystems. Furthermore, custom designs can be created on the basis of the variabilities. To summarize, the scientific contributions of this chapter are as follows:

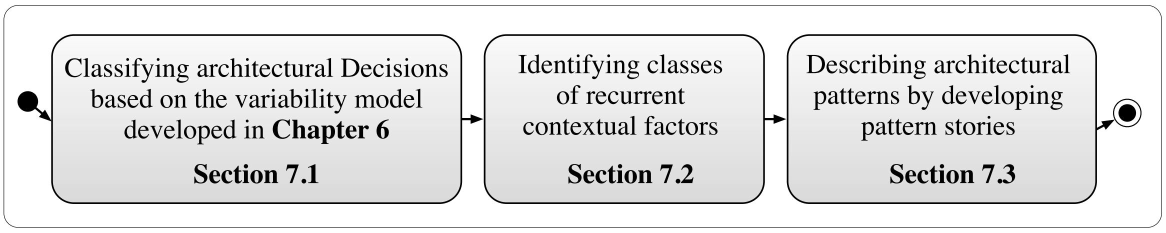
- This chapter presents a variability model for variable architectural design decisions of software ecosystems.
- The variability model is the result of the examination of a diverse range of software ecosystems from social, business, application, and infrastructure viewpoints.
- The knowledge of literature and practice is systematically consolidated into an integrated model.
- The variability model captures the dependency constraints between the architectural design decisions.
- We identified three major service provision scenarios called open software ecosystem, in-house software ecosystem, and semi-open controlled software ecosystem.

Chapter 7

Architectural Patterns of Software Ecosystems

In this chapter, we present three architectural patterns namely *resale software ecosystem (RSE), partner-based software ecosystem (PSE), and open source software-based ecosystem (OSE)*. The patterns provide an answer for RQ5 discussed in Chapter 4, i.e., *what are the main architectural designs of software ecosystems and what is their linkage to business objectives and quality attributes of ecosystem health?*

The architectural patterns are a part of *the methodical knowledge* developed in this thesis. The goal of the methodical knowledge is to provide methods and techniques to design ecosystem architectures. As mentioned in Chapter 2, the term *ecosystem architecture* refers to a part of an enterprise architecture that denotes the inclusion of third-party providers in the architecture.



Classifying architectural Decisions based on the variability model developed in Chapter 6
Section 7.1
Identifying classes of recurrent contextual factors
Section 7.2
Describing architectural patterns by developing pattern stories
Section 7.3

Fig. 7.1 Outline of Pattern Mining Process

Figure 7.1 shows the overall process of pattern mining. In Section 7.1, we classify architectural design decisions of existing ecosystems based on the variability model developed in Chapter 6. This results in identifying three groups of recurrent architectural design decisions and the ecosystems realizing those groups of decisions. Afterward, we identify the relations between the groups of the decisions and platform providers' contexts by investigating the contexts of platform providers, who belong to one group of ecosystems. Thereby, we identify recurrent contextual factors in Section 7.2. In Section 7.3, we develop stories for the patterns to clarify the relations between the architectural decisions and quality attributes of ecosystem health.

Pattern relations are the topic of Section 7.4. In Section 7.5, further results of our investigation are presented. Section 7.6 summarizes our findings. This chapter is based on our previous publications [JZE+18, JZK+18].

7.1 A Classification of Architectural Design Decisions

In this section, we aim for classifying architectural design decisions of software ecosystems. For this purpose, we use the data analysis and clustering technique namely *KJ method* [Kaw91] that helps us organize and group the architectural knowledge that we extract from the existing ecosystems. Other methods can be used as well to achieve classifications. A well-known example is *multi voting.* We do not use this technique because it is relevant for a situation when the act of classifying is performed by a group of stakeholders using voting techniques. In our work, we improve the classification, after its development, by surveying other academic researchers in several consecutive iterations.

In 1951, Jiro Kawakita developed this method for the field of psychology. Since then, it has been widely applied in other fields including computer science to identify patterns in a qualitative way [IYM17]. In addition, the method is widely accepted in the computer science pattern community organized by the Hillside Group[1] that hosts the pattern conference series called Pattern Languages of Programs (PLoP) [SKTI16].

Using the KJ method, we identify a classification of architectural design decisions of software ecosystems and the ecosystems that realize classes of decisions. Our approach consists of three main steps, i.e., 1) *element mining*, 2) *labeling*, and 3) *grouping* as shown in Figure 7.3. In the following, Section 7.1.1 describes our element mining and labeling activities (steps 1 and 2) that concern the investigation of existing ecosystems. In Section 7.1.2, grouping (step 3) is performed, where three final groups of architectural design decisions are identified.

[1]`https://hillside.net`. Last Access: January 10, 2022.

7.1.1 Investigation of 111 Existing Software Ecosystems

To identify architectural patterns of software ecosystems, we initially investigate existing ecosystems. This has been achieved by performing *element mining* and *labeling*.

Element Mining: Collection of a List of Ecosystems

The first step of pattern mining is *element mining* that is the act of collecting objects of study from possible sources[Kaw91]. The basis of our investigation is the ecosystems from real-world. Therefore, the objects of our study are existing ecosystems.

At the top of Figure 7.3, the activities conducted during the element mining are shown, which ultimately result in collecting a list of 111 software ecosystems. Initially, we define our search terms to find the ecosystems. To capture as many ecosystems in as diverse application domains as possible, we use a taxonomy for software ecosystem introduced by Bosch [Bos09] to define our search terms. Table 7.1 shows the taxonomy. The author classifies open software platforms into three types, i.e., desktop, web, and mobile whereas, each type appears in three dimensions, i.e., operating system, application, and end-user programming. Examples of our search terms that are inspired by the taxonomy are "top operating systems", "top mobile platforms", and "top web platforms". Furthermore, we use the list of ecosystems developed in Section 6.1.1 to extend our search terms.

Table 7.1 Software Ecosystem Taxonomy Proposed by Bosch [Bos09]

Platform / Category	Desktop	Web	Mobile
End-User Programming	Excel Mathematics	Yahoo! Pipes, Microsoft PopFly	None so far
Application	Ms Office	Salesforce, eBay, Amazon, Ning	None so far
Operating System	Ms Windows, Linux, Apple OS X	Google AppEngine, Yahoo Developer	Palm, Android, iPhone

In the next step, we use the search terms to perform web searches. Each web search results in a list of potential ecosystems. To add a potential ecosystem to our list, we consider whether it conforms to the definition of software ecosystem presented in Section 2.2.1. For instance, we check whether there is a platform that is a basis for the involvement and contribution of different actors. Figure 7.2 depicts an excerpt of the ecosystems from our final list. For better readability, the figure groups the ecosystems based on their application domains. The figure shows the common terms for the third-party developments inside each application domain. For example, the

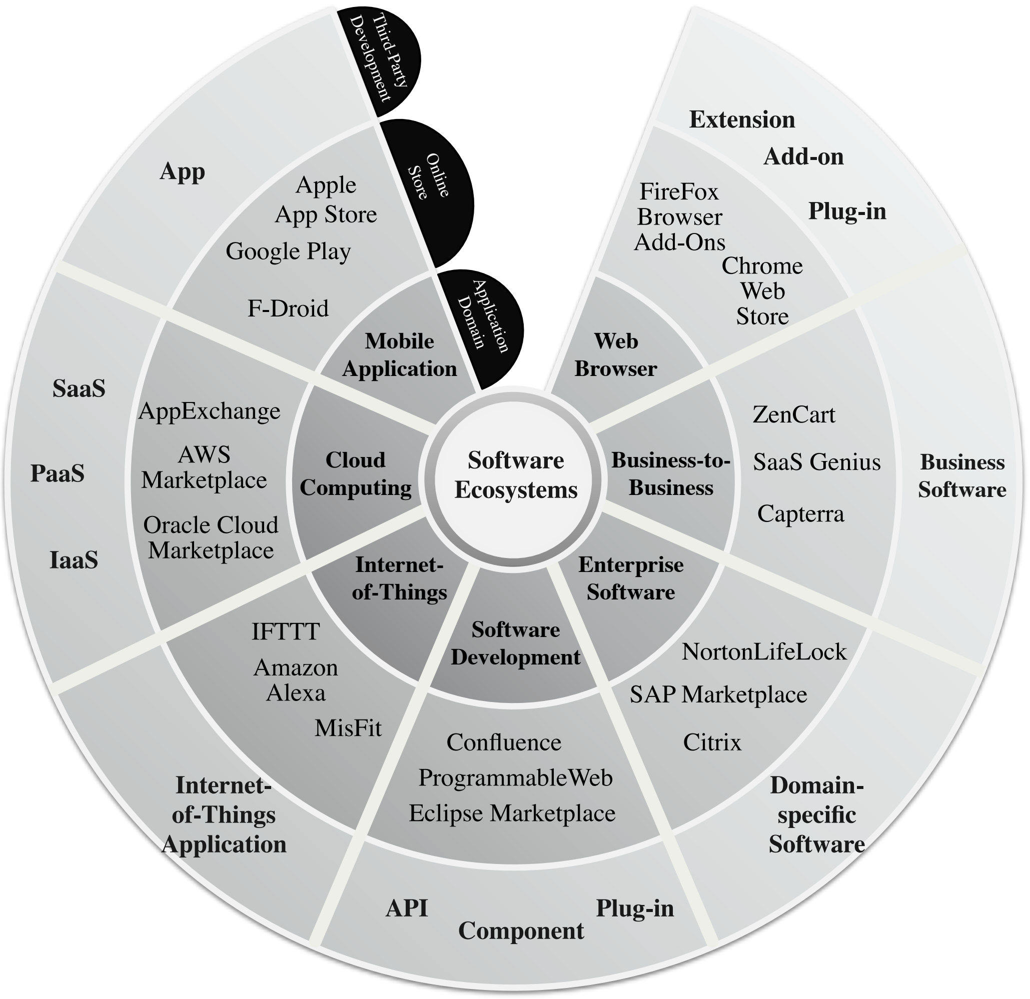


Fig. 7.2 Excerpt of Software Ecosystems Investigated during the Pattern Mining

term *App* refers to the applications developed for portable devices such as smartphones whereas *SaaS*, *PaaS*, and *IaaS* refer to cloud computing services that are software and hardware computing resources provided remotely. During the web searches, we use a search refinement feature provided by the Google search engine called *people also search for* [PAS12]. Using it, similar search results are suggested. This helps us to identify further open platforms that are used to provide services in the same or similar application domains. For example, by searching for *Salesforce*, we can add *SAP* and *Amazon Web Services* to our list. Similar to *Salesforce*, these are open platforms for cloud computing service provision. A complete list of the ecosystems can be found in Appendix C.

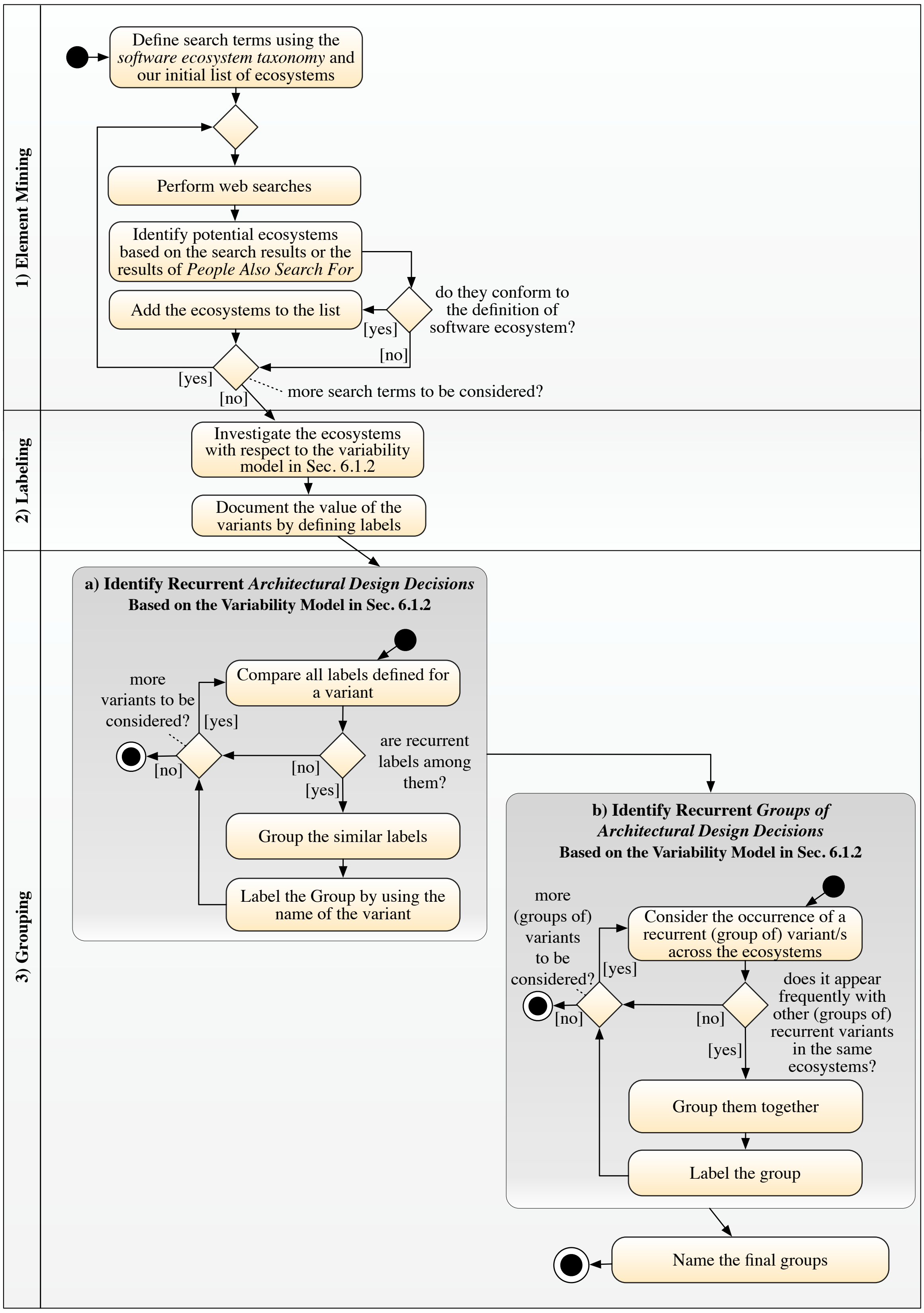

1) Element Mining
Define search terms using the *software ecosystem taxonomy* and our initial list of ecosystems
Perform web searches
Identify potential ecosystems based on the search results or the results of *People Also Search For*
do they conform to the definition of software ecosystem?
[yes]
[no]
Add the ecosystems to the list
[yes]
[no]
more search terms to be considered?
2) Labeling
Investigate the ecosystems with respect to the variability model in Sec. 6.1.2
Document the value of the variants by defining labels
3) Grouping
a) Identify Recurrent *Architectural Design Decisions* Based on the Variability Model in Sec. 6.1.2
Compare all labels defined for a variant
are recurrent labels among them?
more variants to be considered?
[yes]
[no]
[no]
[yes]
Group the similar labels
Label the Group by using the name of the variant
b) Identify Recurrent *Groups of Architectural Design Decisions* Based on the Variability Model in Sec. 6.1.2
Consider the occurrence of a recurrent (group of) variant/s across the ecosystems
does it appear frequently with other (groups of) recurrent variants in the same ecosystems?
more (groups of) variants to be considered?
[yes]
[no]
[no]
[yes]
Group them together
Label the group
Name the final groups

Fig. 7.3 Process of Classifying Architectural Design Decisions using the KJ Method

Labeling: Identification of Architectural Design Decisions based on the SecoArc Variability Model

Labeling concerns the investigation of objects of study. Specifically, first, the objects are examined. Afterward, the relevant information is extracted by introducing *labels*. A label is a self-explanatory and concisely formulated term that represents a concept related to the objects [Ray97].

At this stage, we examine the ecosystems of our list in order to identify architectural design decisions realized in those ecosystems. For this purpose, we use the variability model presented in Section 6.1.2 since the variability model provides a set of variable architectural design decisions distributed among the main architectural aspects, i.e., social and organizational, business and management, and application and infrastructure. Figure 7.3 refers to the activities that we perform during the labeling. We consider how the variants of the variability model are realized in the ecosystems. We document this information by defining new labels. Thereby, each label is an instance of a variant and, therefore, an architectural design decision. Table 7.2 shows part of the results of labeling for six ecosystems that have been investigated with respect VP18 of the variability model. In the table, the labels are the concrete values of the variants for each

Table 7.2 Examples of Ecosystems Investigated during Labeling of the KJ Method (“–” means a variant is not realized in an ecosystem.)

VP18: Feedback Loop Facilitator / **SECO**	**V18.1: Rating and Reviewing**	**V18.2: Ranking**	**V18.3: Market Analytics**	**V18.4: Version Control Management**	**V18.5: Ticket System**
SAP	–	–	Store Partner Cockpit: User statistics, e.g., on page views[1]	–	–
Apple	*Star rating, Reviewing*	*Recommended / Essentials / Bestseller / Newly added lists*	–	–	–
Firefox	*Star rating, Reviewing*	R*ecommended / Popular lists*	–	Using *GitHub*	*Bug tracking using Bugzilla, Firefox Developer Tools*[2]
Inkscape	–	–	–	*Using GitLab*	*CxxTest Testing framework, Gitlab Bug Tracker*
Amazon Alexa	*Star rating, Reviewing*	*Featured third-party*	–	–	–
Microsoft Azure	–	–	*The Azure Marketplace as a selling tool*[3]	–	–

ecosystem. We defined these labels by obtaining information from the obtaining the publicly available sources on the ecosystems. The labels related to V18.1-V18.3 have been defined by referring to the online stores while the labels V18.4-V18.5 originate from the official website of the platform providers. Not all variants are realized in each ecosystem, in this case, we defined no labels.

7.1.2 Classes of Recurrent Architectural Design Decisions

During the group step, to identify frequently applied architectural design decisions, we identify the decisions (labels) that reoccur among the ecosystems of our list. Afterward, recurrent groups of architectural design decisions are identified by specifying recurrent groups of labels.

Grouping: Identification of Groups of Design Decisions

During grouping, a classification of previously defined labels is achieved. In a bottom-up conceptualization, the labels are grouped and the groups are classified into larger groups. This process is continued until distinctive final groups of labels are specified. This is when the groups cannot be further grouped together. The general guideline to achieve a suitable abstraction is to reduce the number of groups to less than ten [Ray97].

To identify patterns of ecosystem designs, our goal is to consider whether there are groups of architectural design decisions that are frequently realized together in practice. For this purpose, we use the labels that are created in the labeling step to check whether recurrent groups of architectural design decisions exist among them. This specifically means we group the labels in two steps as shown in Figure 7.3: First, by *identifying the recurrent design decisions* and then by *identifying the recurrent groups of design decisions* across the ecosystems. In the following, we elaborate on these steps.

a) Identifying Recurrent Architectural Design Decisions We identify recurrent design decisions of the ecosystems of our list by first identifying, and then, grouping the similar labels of the variants. Initially, we compare the labels of each variant. If we detect similarity among the labels, we group them and name the group of labels by using the name of the variant. We repeat this process for all the variants. At the end of this process, we achieve several groups of labels while each group comprises a set of values for a common variant. For instance, some of the ecosystems realize a ticket system in different ways, e.g., *Firefox* and *Inkscape* in Table 7.2. Firefox realizes a ticket system using *Bugzilla* and Inkscape uses *GitLab.* We group Bugzilla and GitLab and name the group *ticket system.* Thereby, we consider *ticket system* as a recurrent design decision.

b) Identifying Recurrent Groups of Architectural Design Decisions: We identify recurrent groups of design decisions by identifying groups of labels that appear together across the ecosystems. In the context of our variability model, this means the ecosystems provide similar values for the groups of variants. For this purpose, first, we consider the groups of labels identified in the previous step and if there are two or more groups that are realized together by the ecosystems, we group them and label them accordingly. In the case that we have identified a group in the past, we notice the repetition of this occurrence.

Likewise, we continue to classify the groups of variants into larger groups if we notice that a combination of groups reoccurs across the ecosystems. Each time we define a new label when we group the variants. For example, we notice that some ecosystems demand third-party providers to pay entrance fees. The entrance fees come in terms of one-time payments, periodic payments, and membership payments. We cluster these labels into one group, called *entrance fee.* Afterward, we notice that while a group of platform providers demands an entrance fee, they develop a mutualism symbiotic relationship with other providers by developing shared software products with them. Therefore, we group *V13.3: Entrance fee* and *V4.1: Mutualism* into one group and label the group by *platform providers providing commercial ecosystems and partnering with other providers.*

We proceed with the button-up grouping process described above until no new groups can be identified. In several iterations of labeling, comparing the labels, and grouping the labels, we achieve three final groups of architectural design decisions. Table 7.3 demonstrates the final groups that form the basis of the architectural patterns presented in Section 7.3. Each final group comprises a set of variants. On the left side of the table, it is shown to which variation points the variants belong. Each cell refers to one or more recurrent variants. For explanatory reasons, some labels are shown on the cells that provide more information on how the variants are realized in the ecosystems. In case that a variation point has no common value in a final group, the em dash sign ("–") is used in the cell. We name the final groups as *Resale Software Ecosystem*, *Partner-Based Ecosystem*, and *Open Source Software (OSS)-Based Ecosystem.* The names refer to the most distinctive characteristics of the ecosystems of

Table 7.3 Classes of Recurrent Architectural Design Decisions
("–" means a variation point has no common variant among the ecosystems.)

Variation Points (VPs)	**Final Group 1: Resale Software Ecosystem**	**Final Group 2: Partner-Based Software Ecosystem**	**Final Group 3: OSS-Based Software Ecosystem**
VP1: User Who are the users of third-party developments?	**V1.1: Naive** - *End-users, e.g., users of mobile Apps*	**V1.2: Expert** - *Enterprises*	**V1.3: Developer** - *Developers of open-source software*
VP2: Niche Player Relation What are the relations between niche players?	**V2.1: Metropolis Model** - *Platform providers being the keystones*	**V2.2: Onion Model** - *Layers of strategic partners*	**V2.3: Lone Wolf** - *Major contributions by foundations*
VP3: Release Model Who initiates a release of the platform?	**V3.1: Push-Oriented** - *Regular release by providers*	**V3.2: Pull-Oriented** - *Update-requests by users*	**V3.2: Pull-Oriented** - *Update-requests by users*
VP4: Symbiotic Relation What are co-existence relationships?	**V4.3: Competition** - *Competing providers*	**V4.1: Mutualism** - *Developments shared by providers*	**V4.2: Commensalism** - *Developers benefiting each other's developments*
VP6: Knowledge Sharing Which software features enable knowledge sharing?	**V6.1: Q&A Forum** - *Developer / user / idea forums* **V6.2: Documentation Framework** - *Developer manuals. Wikis*	**V6.1: Q&A Forum** - *Partner portal* **V6.2: Documentation Framework** - *Partner portal: Documentation of software frameworks*	**V6.1: Q&A Forum** - *Public developer forum, e.g., StackOverflow* **V6.2: Documentation Framework** - *Manuals, wiki, public resources from FOSS community*
VP7: Complementary Partnership Who are the third-party providers?	**V7.1: Strategic Partner** - *Hardware / software suppliers, strategic partners* **V7.2: Independent Developer** - *Mass number of independent developers*	**V7.1: Strategic Partner** - *Hardware / software suppliers, strategic partners, system integrator, etc.*	**V7.1: Strategic Partner** - *Foundation members, Strategic partners* **V7.2: Independent Developer** - *High number of independent developers*
VP8: Licensing What are ownership and redistribution rights of software?	–	**V8.3: Bring Your Own License (BYOL)** - *Allowing companies to sell software under their own licenses*	**V8.1: Public License** - *free software / open-source software licenses, e.g., GPL and CPL*
VP9: Platform Control Is access to the platform allowed?	–	**V9.2: Closed Platform**	**V9.1: Equity Ownership**
VP10: Platform Contribution Is contribution to the platform allowed?	**V10.1: Platform Development** **V10.2: Closed Contribution**	**V10.1: Platform Development** -	**V10.1: Platform Development** -
VP11: Third-Party Development Intellectual Property Are the developments reused in other ecosystems?	**V11.2: Restricted Reuse**	**V11.2: Restricted Reuse**	**V11.1: Granted License**
VP12: Third-Party Development Openness Are the developments open-source?	–	**V12.2: Closed Source**	**V12.1: Open Source**
VP13: Fee What are costs of participating in the ecosystem?	**V13.3: Entrance Fee** - *One time / periodic payment*	**V13.1: Platform Fee** - Payments for platform editions. Monetized APIs **V13.2: Documentation Fee** - *Payments for access to documentation* **V13.3: Entrance Fee** - *Membership in partner programs / different payments for different partners*	*No Fee*
VP16: Entrance Check What security and policy checking should be performed upon entering the ecosystem?	**V16.1: Leaks and Bugs** - *Checked using entrance review process* **V16.2: Policy Violation**	**V16.1: Leaks and Bugs** - *See Testing* **V16.2: Policy Violation** **V16.3: Quality Check** - *Entrance requirements*	**V16.1: Leaks and Bugs** - *See Testing* **V16.2: Policy Violation**
VP17: Store How is software distributed?	**V17.1: App Store** - *Stores of applications*	**V17.3: Catalog** - *Lists of developments*	**V17.2: Repository** - *Repositories of source code*
VP18: Feedback Loop Facilitator Which software features enable a feedback loop between the users and providers?	**V18.1: Rating and Reviewing** - *Binary rating, Scale rating (stars, sliders), Reviewing* **V18.2: Ranking** - *Featured / popular / new developments* **V18.5: Ticket System** - *Bug tracking*	**V18.3: Market Analytics** - *User statistics, CRM, Marketing planers*	**V18.4: Version Control Management** - *Tools from the FOSS community, e.g., SVN, Git, Mercurial, Perforce, etc.* **V18.5: Ticket System** - *Issue tracking from the FOSS community used, e.g., using Jira*
VP19: Extension Development Kit Which software features enable providers to extend the platform?	**V19.1: API Management** - *SDK, API Documentation, Code. Multi development lines* **V19.2: IDE** - *Ecosystem-specific IDEs, e.g., Xcode, Android Studio, AWS tool kit* **V19.5: Testing** - *Crowd / Unit testing, Code review*	**V19.1: API Management** - *SDK, API Documentation* **V19.5: Testing** - *Acceptance / System / Smoke Integration Testing*	**V19.1: API Management** - *SDK, API Documentation, Source code* **V19.2: IDE** - *IDEs from the FOSS community, e.g., Eclipse* **V19.2: Programming Language** - *Certain flexibility to choose among alternatives* **V19.5: Testing** - *Unit / Integration / System testing. Tools from the FOSS*
V25: Service Execution	**V25.3: Data Center** - *To fetch the data for service provision*	–	–

each group, which we acquired during the pattern mining. Notably, during the process of pattern mining, we encountered new architectural variation points and variants, which we added to the variability model.

7.2 Classes of Recurrent Contextual Factors

In this section, we investigate whether there is a relation between the final groups of architectural design decisions presented in Table 7.3 and the context of platform providers. To answer this question, we use a contextual model introduced by Kruchten [Kru13]. The author uses the term *context* and argues that software development practices become unique mainly due to different contexts of the organizations running those practices. Eight key contextual factors are introduced, i.e., size, stable architecture, business model (commerciality), team distribution, rate of change, age of system, criticality, and governance. The model is not specific to software ecosystems while it has been generally defined for software development practices, which need to be agile and at the same time should scale up. However, it provides us with a solid basis to investigate the context of platform providers. Furthermore, in our study, we need to cope with the limitation of being outside of the organizations. Therefore, among all the contextual factors, we consider the ones, which can be determined using the publicly available data. With this respect, we consider *company size*, *domain criticality*, and *commerciality*. The literature on software ecosystems (e.g., [BPT$^+$14, JC13, APA14]) have frequently discussed the significance of these factors. For company size, we refer to the number of employees in an enterprise. Domain criticality determines whether software failure is dangerous to human lives. To decide on the criticality of a domain, we refer to its application domain. For instance, we consider enterprise software to be a non-critical domain whereas domains like safety and security are the critical ones. Additionally, commerciality defines the degree of protecting intellectual property.

Table 7.4 shows the common contextual factors. The value of each cell represents a shared value for a factor. A cell marked with "–" means there is no shared value among the platform providers regarding a contextual factor. Furthermore, since software ecosystems include third-party providers, company size as an intra-organizational factor does not suffice to decide on the size of an ecosystem [Bos09]. Therefore, we add *market size* as another contextual factor and measure it by referring to the number of third-party developments on the stores. As suggested by Duc et al. [DMHP14], in the case of ecosystems that are built around open-source software, the number of forks is an indicator to decide on its size. For the interpretation of size, we use the

Table 7.4 Classes of Recurrent Contextual Factors
("–" means the factor has no shared value among platform providers.)

Contextual Factor	Final Group 1: Resale Software Ecosystem	Final Group 2: Partner-Based Software Ecosystem	Final Group 3: OSS-Based Software Ecosystem
Company size Small (1-99 employees) Midsize (100-999 employees) Large (1000+ employees)	Large companies	–	–
Market size Small (1-99 extensions) Midsize (100-999 extensions) Large (1000+ extensions)	Midsize to large markets of third-party developments	–	Large markets of third-party developments
Domain criticality Failure in extensions dangerous to human lives	No criticality	Industrial and safety-critical applications	No criticality
Commerciality The degree of protecting intellectual property	–	Commercial software, Monetized APIs, Entrance fee	Free and open access to the platforms

scales provided by Gartner [SMB18], which is a well-reputed market observer firm. We identify the context of the patterns by identifying similar contextual factors inside each group of ecosystems.

7.3 Architectural Patterns of Software Ecosystems

In this section, we present three architectural patterns for software ecosystems called *resale software ecosystem (RSE)*, *partner-based software ecosystem (PSE)*, and *OSS-based software ecosystem (OSE)*. The architectural patterns are based on the identification of three groups of recurrent architectural design decisions in Section 7.1 and the recurrent contextual factors in Section 7.2.

The architectural patterns of this thesis follow the structure described by the metamodel in Figure 7.4. An *architectural pattern* is referred by a name and a high-level business objective. To choose a name for each pattern, we refer to the names of the final groups of ecosystems in Section 7.1. Furthermore, we characterize each pattern by using a high-level business objective based on our knowledge of the existing ecosystems that we obtained during the pattern mining. Furthermore, each pattern consists of a *landscape*. In the context of this thesis, we use the term *landscape* [RW12] to refer to an arrangement of human actors and software and hardware components that is a subset of ecosystem architecture. We derive the landscape from the architectural design decisions of the final groups (cf. Table 7.3). The landscape represents a part of ecosystem architecture that can help solve a problem at the managerial and governance level

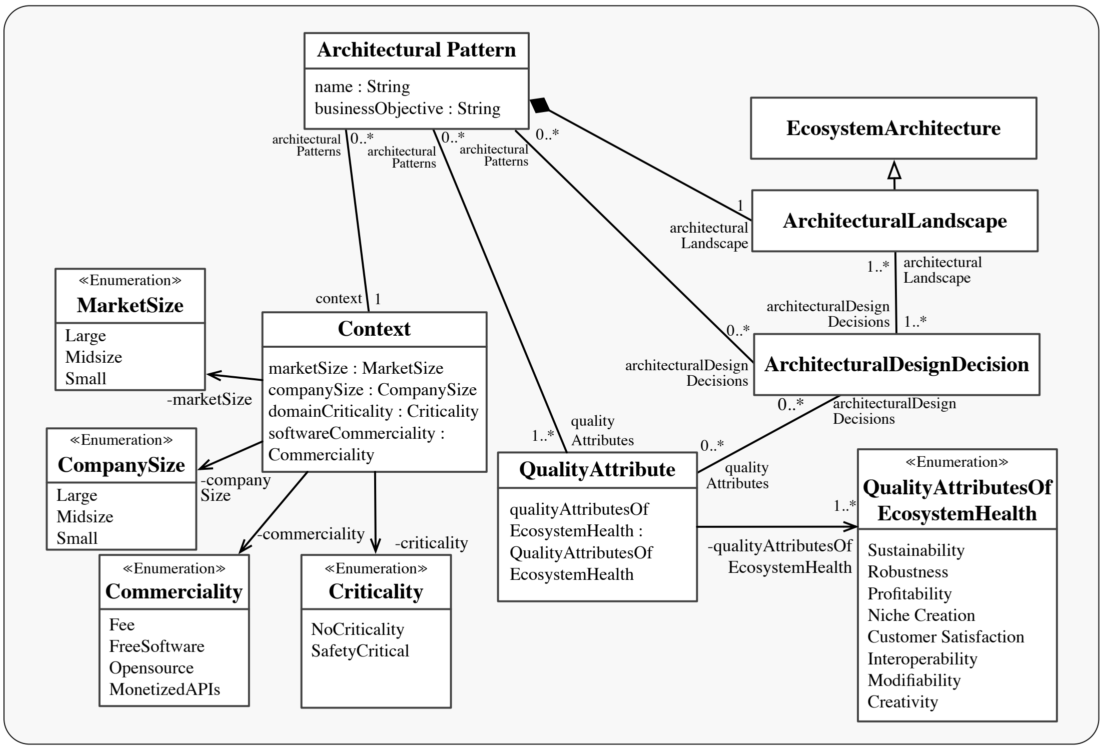


Fig. 7.4 Metamodel of the SecoArc Architectural Patterns

in software ecosystems. The term is inspired from *architectural landscape* introduced by Rozanski et al. [RW12, p. 254] to denote a high abstraction of the system that is particularly crucial for the communications between stakeholders with different viewpoints. Thereby, an architectural landscape portrays key architectural elements while concealing details. In addition, an architectural pattern addresses certain quality attributes of ecosystem health. We map the quality attributes to each pattern by referring to the definitions and examples for the quality attributes of ecosystem health given by Ben Hadj Salem Mhamdia [BHSM13]. In particular, we map the design decisions of the patterns that can help support a quality attribute according to those definitions. Finally, a pattern is related to a certain context. The context refers to the platform providers' contextual factors presented in Table 7.4).

We develop pattern stories [BHS07] for the architectural patterns by using a pattern template presented in Section 2.3.2. The knowledge of some sections of the pattern template can be directly derived from our earlier results in this chapter. These sections are the context, solution, examples, and known uses. Based on this knowledge, we develop the knowledge of the other sections, i.e., consequences, forces, and problem.

Firstly, we specify the consequences of each pattern. To identify the problems and forces, we refer to the challenges that the platform providers might faced, which led to the inclusion of those design decisions in the architecture. Having all sections defined, we develop a story for each pattern.

In the following, first, we clarify the relation between the pattern sections and the architectural patterns identified in this thesis. Following that, we present the pattern stories.

Context: In case of our patterns, context refers to the contextual factors derived in Section 7.2. We use thought bubbles to illustrate the contexts.

Problem: In the context of this thesis, it is a managerial problem that a platform provider would like to solve while designing an ecosystem.

Forces: Two main top-level forces faced by the platform providers to open up their platforms and create software ecosystems are **market growth** and **cost** [Bos09]. Further forces are related to the health of software ecosystems. To identify the quality attributes of ecosystem health relevant to the architectural patterns, we use the quality model of ecosystem health proposed by Ben Hadj Salem Mhamdia [BHSM13] as presented in Table 7.5.

Solution: The solution section of our patterns is the landscape that is based on the final groups of architectural design decisions. We visualize the landscapes using the notation of UML Component Diagrams [OMG17] as shown in Figures 7.5, 7.7, and 7.9.

Consequences: They refer to the architectural design decisions of the patterns and explain how they can help resolve the forces. At this point, the consequences are the quality attributes of ecosystem health that are addressed by the patterns.

Known Uses: Known uses of patterns are exemplary real-world software ecosystems, where we detected each pattern. In addition, we refer to the application domains in which service provisioning happens.

Related Patterns in Literature: We adapt this section by focusing on the relations between our patterns and other patterns introduced by the related work in literature. Relations between the architectural patterns of this thesis are discussed in a separate section (cf. Section 7.4).

Example: It is a description of a concrete ecosystem, which conforms to one of our patterns. An ecosystem conforms to one of the patterns if architectural design decisions of the ecosystem comply with the pattern, i.e., the architectural design decisions of the

Table 7.5 Quality Attributes of Ecosystem Health Addressed by Architectural Patterns (Based on Ben Hadj Salem Mhamdia's Quality Model [BHSM13])

Quality Attribute	Description
Sustainability	The ecosystem should be safe from external threats. Successfully confronting external threats has a long-term impact on ecosystem's success during its evolution.
Robustness	On the managerial level, it is the survival degree of the ecosystem's participants either in relation to other ecosystems or over time.
Profitability	The ability to increase revenue generation.
Niche Creation	The capacity to provide novel services by introducing meaningful and new diversity.
Customer Satisfaction	It refers to the external satisfaction of the service quality perceived by customers or users of the services.
Interoperability	From the perspective of enterprise architecture, the ecosystem should help an enterprise to minimize preparatory efforts to make a new relationship with other companies.
Modifiability	The degree to which the source code of software platform can be modified without introducing defects or degrading quality.
Creativity	The capability of an ecosystem to accommodate extensions with diverse characteristics like use case, programming language, execution environment.
Cost-Effectiveness	It refers to the capability of saving cost during establishment and governance of the ecosystem.

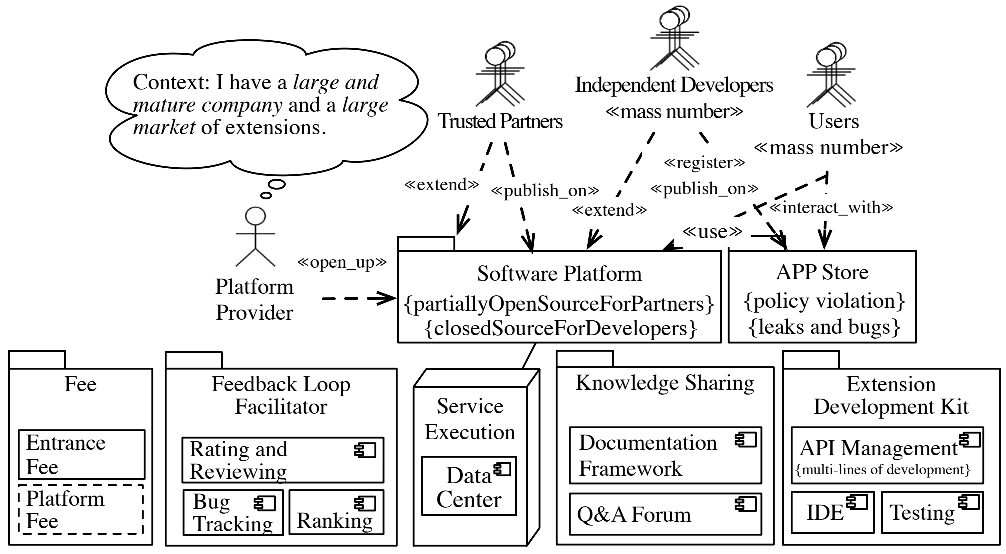
pattern are realized in the ecosystem. Furthermore, the types of human actors and their arrangement should match the pattern landscape.

7.3.1 Resale Software Ecosystem (RSE)

Resale software ecosystem (RSE) is an architectural pattern to establish control over a mass number of third-party providers by means of strategies, which help platform providers govern a large ecosystem around their platforms. Examples of the strategies are to define membership criteria, facilitate developer's independence, and enable discoverability of high quality third-party developments. The platform provider, in this case, is a large company. It includes a mass number of independent developers in process of software development. The independent developers work in completely separated teams. After the third-party developments are developed, they are sold several times to a mass number of users on an online store.

Resale Software Ecosystem (RSE):
Business Scalability

Landscape


Context: I have a *large and mature company* and a *large market* of extensions.
Platform Provider
Trusted Partners
Independent Developers «mass number»
Users «mass number»
«extend»
«publish_on»
«register»
«publish_on»
«extend»
«interact_with»
«use»
«open_up»
Software Platform {partiallyOpenSourceForPartners} {closedSourceForDevelopers}
APP Store {policy violation} {leaks and bugs}
Fee
Entrance Fee
Platform Fee
Feedback Loop Facilitator
Rating and Reviewing
Bug Tracking
Ranking
Service Execution
Data Center
Knowledge Sharing
Documentation Framework
Q&A Forum
Extension Development Kit
API Management {multi-lines of development}
IDE
Testing

Contextual Factors of Platform Provider

Large Company
Large Market of Third-Party Developments

Architectural Design Decisions

VP1 User: Naive
VP2 Niche Player Relation: Metropolis Model
VP3 Release Model: Push-oriented
VP4 Symbiotic Relation: Competition
VP6 Knowledge Sharing: Q&A Forum, Documentation Framework
VP7 Complementary Partnership: Independent Developer, Strategic Partner
VP10 Platform Contribution: Closed Contribution for Independent Developers / Platform Development for Partners
VP11 Third-Party Development Intellectual Property: Restricted Reuse
VP13 Fee: Entrance Fee, (Platform Fee)
VP16 Entrance Check: Leaks and Bugs, Policy Violation
VP17 Store: App Store
VP18 Feedback Loop Facilitator: Rating and Reviewing, Ranking, Bug Tracking
VP19 Extension Development Kit: API Management (multi-lines of development), IDE, Testing
VP25 Service Execution: Data Center

Relevant Quality Attributes of Ecosystem Health

Sustainability, Interoperability, Robustness

Fig. 7.5 Resale Software Ecosystem:
An Architectural Solution to Achieve Business Scalability

Resale Software Ecosystem Pattern Story

Figures 7.5 shows the RSE pattern that follows the metamodel introduced in Figure 7.4. In the following, the RSE pattern story based on this knowledge is presented.

Context. The platform provider owns a large enterprise and a large market of third-party developments. In addition, the platform and third-party developments are not safety-critical.

Problem. The platform provider wants to ensure the quality of third-party developments while opening the platform to a high number of independent providers. Thus, these questions arise: Firstly, **how to manage the ecosystem membership for the mass number of third-party providers?** Secondly, **how to enable the third-party providers' independence during software development and service provision?** Thirdly, **how to ensure discoverability of high quality developments among the high number of offers?**

Forces. In a large market of third-party developments, the users may receive low quality or malfunctioning software. This adversely affects user experience and consequently the platform provider's reputation. Therefore, **sustainability** is a force. Furthermore, organizational bureaucracy needs to be minimized to facilitate the integration of the high number of third-party contributions into the ecosystem, which turns **interoperability** into a relevant force. Another risk is that the third-party providers are not supported with suitable software features that ensure their independence. Thus, they may fail to extend the platform and abandon the ecosystem or start participating in a competitor's ecosystem, which turns **robustness** into a relevant force. With this respect, sustainability, interoperability, and robustness can be identified as forces behind the platform provider's urge to apply the RSE pattern while growing the ecosystem by including a mass number of third-party providers.

Solution. The architectural landscape of the RSE pattern is shown in Figure 7.5. **To manage membership of a high number of third-party providers, the providers need to register in the ecosystem and possibly pay fees.** The fees can vary, depending on the platform provider's strategies. Fees are set in terms of platform and entrance fees. Usually, the fees are not high to enable as many third-party providers as possible to join the ecosystem. Furthermore, accessing the platform's source code is constrained by defining openness policies. During the pattern mining, we noticed that a policy frequently applied by the existing ecosystems is to close the source code to the independent developers (VP10 Platform Contribution: Closed Contribution). However, partners can partially access the source code during the development of shared products and services (VP10 Platform Contribution: Platform Development for Partners). This is shown in Figure 7.5 by using tags on the software platform. After registering in the ecosystem, the third-party providers are obliged to follow a wide range of further policies that are regulated by the platform provider. Examples are the legal policies like using certain licenses and technical decisions like the choice of programming language and execution environment for services.

To support developers' independence, the developers are provided with a toolchain for the whole software life cycle. This includes an API management to access the platform's functionality, Integrated Development Environment (IDE), and testing features including emulators and simulators to imitate an execution environment. An example of existing toolchains is Xcode in the Apple ecosystem. The API management is set to distinguish between multi-lines of development of the platform. Different development lines result in different variations of the platform that the platform provider uses in the ecosystem to address the needs of different groups of target users. This often leads to the creation of a software product line. Furthermore, using Q&A forums, the users and developers trigger discussions on different topics and spread the knowledge throughout the ecosystem. Another concern is to support developers' independence is to provide infrastructure resources required for the service execution. This help the developer relieve the costs of providing infrastructure resources. Among the infrastructure resource, specially data centers are often provided to handle tasks related to the data storage.

To ensure discoverability of high quality third-party developments among the available offers, rating and reviewing, and ranking features are used. A ranking feature generates different lists of high quality third-party developments such as lists of popular/featured/newly published developments. In addition, the platform provider might proactively avoid low quality developments entering the ecosystem by conducting a review process before they are published on the store. Static code analysis is an example of such a review process to ensure malware-free developments. Furthermore, bug tracking is used both on the user and developer sides to share security and functional issues faced while using the developments. Apple Radar[2] is an example of bug trackers used in the Apple ecosystem.

[2]`https://developer.apple.com/bug-reporting`. Last Access: January 10, 2022.

Consequences. Testing frameworks and issue tracking features support the detection of malwares and misfunctions during software development that improve **sustainability**. In addition, using the App store and IDE, the complexity of licensing and software development in the ecosystem is reduced, which contributes to an enhanced **interoperability**. By executing services on a remote server, the computing tasks become independent of the end assets used by the users. While storing the data on the servers and remote service execution contribute to the data availability, it additionally contributes to interoperability at the ecosystem level. Furthermore, **robustness** is addressed by providing the features of feedback loop facilitator, i.e., rating, ranking, and reviewing, using which high-quality services and their providers can be known and further promoted in the ecosystem.

In general, by successfully accommodating the high number of third-party providers and ensuring the quality of third-party developments, the ecosystem can achieve **business scalability**. However, establishing a store and maintaining it impose extra **costs** on the platform provider. Therefore, we suggest the RSE pattern for the platform providers, who own a large market of high quality developments or own trusted partners so that they can keep the quality of service provision at an acceptable level.

known Uses. Operating system, mobile application, and web browser are typical examples of application domains. Further ecosystems applying *Resale Software Ecosystem* are Apple, Salesforce, and Esri.

Related Patterns in Literature. Other patterns related to the architectural management of large ecosystems can be applied together with the RSE pattern. Weiss and Noori [WN13] propose a pattern namely *Manage Complements* to systematically control the quality of third-party developments by using sandboxes. In addition, coding conventions are used to improve the quality of source code. Furthermore, Stocker et al. [SZZ$^+$18] present a pattern called *rate plan* for the commercialization of APIs while being used by clients. Rate plan can be used with other patterns such as *rate limit* to avoid excessive usage of APIs that harms the overall service quality. Regarding the variability model in this thesis, rate plan can be applied with the VP20: payment. Rate limit can affect the infrastructure variabilities.

Exemplary Software Ecosystem: Adobe

Adobe is a large software company and the provider of a product line of software platforms for graphical designs such as Dreamweaver, Photoshop, and InDesign. A large market of third-party developments (more than 2,500 developments) exists in the online store, which is called Adobe Exchange[3]. The developments are not safety-critical and often commercial.

Figure 7.6 presents the architecture of Adobe ecosystem. Trusted partners like *Esri*, *Microsoft*, and *Magneto* share products with Adobe. The source code of the platforms is partially open to the partners. In addition, a mass number of third-party developers, namely *producers*, develop software on top of the platforms. Entering the ecosystem is subject to an entrance fee for the producers. The producer can publish their developments on the store by successfully passing a review process that checks for malware and policy violations. Star rating and reviewing facilitate a feedback loop between the users and third-party providers. They are used to expose the quality of third-party developments. A ranking feature generates a list of popular third-party developments.

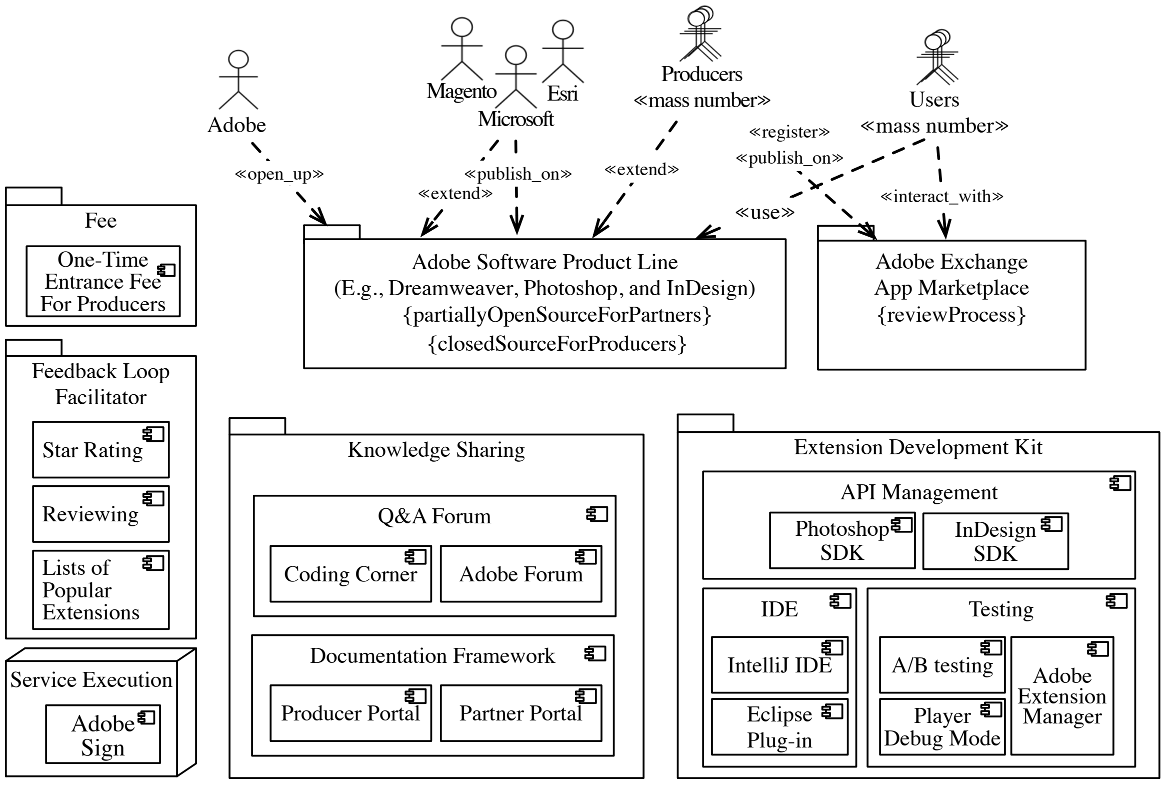


Fig. 7.6 RSE in the Adobe Ecosystem

The platforms are extendible using APIs. Platform SDKs, e.g., Adobe Photoshop CC and InDesign SDKs facilitate APIs management. The SDKs can be used by the

[3]www.adobeexchange.com. Last Access: January 10, 2022.

third-party providers to develop third-party software. Moreover, IntelliJ IDE and an Eclipse plug-in for are provided by Adobe to ease software development for the mass number of producers. Furthermore, software testing is facilitated by means of different testing frameworks such as *Extension Manager*, *A/B testing*, and *PlayerDebugMode*.

Adobe enables knowledge sharing by means of documentation frameworks and Q&A forums. Adobe Partner Portal[4] as well as Producer Portal[5] make necessary documentation available. Adobe Partner Portal is only accessible by the partners. In addition, Coding Corner[6] is a Q&A forum for topics related to software development for the producers. Adobe Forums[7] is a Q&A forum for the users and third-party providers to discuss open questions regarding the usage of the platforms and third-party developments.

7.3.2 Partner-Based Software Ecosystem (PSE)

Partner-based software ecosystem (PSE) is an architectural pattern to grow a commercial and complex software system in an industrial sector with the aid of third-party providers. The platform provider includes third-party providers in software development only by establishing partnerships with them. Using different customization of openness policies, the platform provider protects his/her intellectual property at different levels depending on the trustworthiness of partnerships. Third-party developments are often labeled as *tested* or *validated* on the store.

Partner-Based Software Ecosystem Pattern Story

In this section, we present the pattern story based on the knowledge of the PSE pattern shown in Figure 7.7.

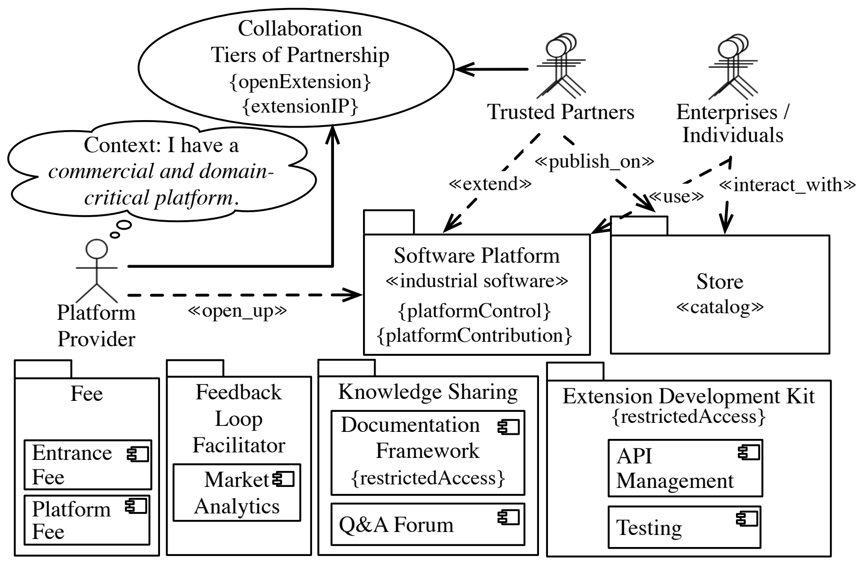
Context. High commerciality and criticality are the major characteristics of the software platform. The platform is complex (and often safety-critical) software for an industrial sector such as supply chains, aerospace, or healthcare.

Problem. The platform provider wants to grow the platform to a new industrial sector by including third-party providers. Subsequently, reasonable software and hardware resources, as well as extensive expertise in certain fields, are required to develop software

[4]`www.adobe.com/partners.html`. Last Access: January 10, 2022.
[5]`technologypartners.adobe.com/home.html`. Last Access: January 10, 2022.
[6]`forums.adobe.com/community/coding-corner`. Last Access: January 10, 2022.
[7]`forums.adobe.com`. Last Access: January 10, 2022.

Partner-Based Software Ecosystem (PSE):
Strategic Growth

Landscape



Contextual Factors of Platform Provider

High Criticality
High Commerciality

Architectural Design Decisions

- **VP1 User: Expert**
- **VP2 Niche Player Relation: Onion Model**
- **VP3 Release Model: Pull-oriented**
- **VP4 Symbiotic Relation: Mutualism**
- **VP6 Knowledge Sharing: Q&A Forum, Documentation Framework**
- **VP7 Complementary Partnership: Strategic Partner**
- **VP8 Licensing: BYOL**
- **VP9 Platform Control: Closed Platform**
- **VP10 Platform Contribution: Platform Development**
- **VP11 Third-Party Development Intellectual Property: Restricted Reuse**
- **VP12 Third-Party Development Openness: Closed Source**
- **VP13 Fee: Platform Fee, Documentation Fee, Entrance Fee**
- **VP16 Entrance Check: Leaks and Bugs, Policy Violation, Quality Check**
- **VP17 Store: Listing**
- **VP18 Feedback Loop Facilitator: Market Analytics**
- **VP19 Extension Development Kit: API Management, Testing**

Relevant Quality Attributes of Ecosystem Health

Profitability, Customer Satisfaction, Market Niche

Fig. 7.7 Partner-Based Software Ecosystem:
An Architectural Solution to Achieve Strategic Growth

solutions for customers in that sector. Given the platform's criticality, the question is: **How to ensure high quality of future third-party developments in the new sector?** Furthermore, considering the platform's commerciality, **how to regulate the platform openness to ensure protection of the intellectual property?**

Forces. It is too costly for the platform provider to supply all resources alone and too risky in terms of failing to deliver a certain degree of demanded quality. Therefore, **profitability** and **customer satisfaction** are the forces to create an ecosystem by involving partners. In addition, given the challenges of developing commercial software in a new industrial sector, the platform provider needs to reach a required market share that is often achieved in practice by fulfilling a niche market [MSJ$^+$16, KDKB14]. Therefore, we identify **niche creation** as a relevant force as well. In summary, profitability, customer satisfaction, and niche creation are the main forces to achieve a reliable ecosystem for the platform provider to cooperate with other providers to generate commercial and high quality services.

Solution. **To ensure high quality of third-party developments in the new sector, the platform provider establishes partnerships only with the providers, i.e., domain experts, who already possess the required resources and expertise in that sector**. The architectural landscape is presented in Figure 7.7. The platform provider and partners collaborate in developing and marketing joint solutions as well as conducting road map sessions. They pursue the same user segment and together compete for market success. Users are enterprises or individuals that require specific services and products for their needs. In the case of highly commercial third-party developments, acceptance tests are performed on users' behalf. Such third-party developments are distinguished on the store by the labels like *tested* or *validated.* In contrary to *Resale Software Ecosystem*, in *Partner-based Ecosystem*, rating and reviewing features do not play a critical role. The third-party developments are rather marketed using market analytics. This mainly transforms the store into a catalog. The market analytics features, e.g., customer relationship management (CRM) and repository mining, enable the partners to track user satisfaction in the ecosystem.

To regulate the platform openness to ensure the protection of the intellectual property, the platform provider specifies a range of openness policies and realizes them by defining partner programs and monetizing resources. Deciding on different customization of openness policies specifies the degree of openness. Boudreau [Bou10] introduces two major types of openness policies, i.e., *platform openness* and *complement openness.* Both types of openness are defined in a more fine-granular way: *Platform openness* comprises *platform control* and *platform contribution. Platform control* specifies the situation when third-party providers have equity ownership of the platform, and *platform contribution* is when the third-party providers only contribute to the development. Furthermore, *extension openness* is defined as *open extension* and *extension intellectual property (IP). Open extension* is the policy to grant licenses to the third-party providers, whereas *extension IP* is a policy to share the intellectual property with them.

An onion niche player relation is realized by forming various tiers of partnerships. Partner programs with tighter partnerships with the platform provider contain fewer entrance barriers and more openness. For instance, while platinum partners may have *platform contribution* but not *platform control*, they need to regularly prove a minimum amount of revenue generation in the sector. However, gold partners that have *platform control* are not subject to such a requirement. The collaboration between the platform provider and partners often begins with partner onboarding provided by the platform provider to help partners establish a common understanding of the ecosystem and the service provisioning. Further training and webinars are an inevitable part of a collaboration in order to communicate and share the knowledge in the ecosystem. The platform provider holds regular social events in terms of community gatherings to keep sharing the knowledge and pursue marketing goals.

Furthermore, the openness policies are realized by monetizing resources and demanding fees. An API management specifies a partner's access permission and corresponding fees. Furthermore, the partners need to pay entrance and platform usage fees on a periodic basis. Another facet of the monetization is documentation frameworks. Such frameworks are a part of the partner programs that are only accessible to the partners. However, Q&A forums are publicly accessible.

Consequences. By setting goals and selecting qualified providers as partners, the platform provider can address the **market niche** in the sector. Specifically, the platform provider and the partners are all the niche players in the ecosystem while the final services are the result of the integration of all players' value creation [KDKB14]. Using validation, verification, and consulting services, the platform provider ensures long-term **customer satisfaction**. Using external licenses by means of BYOL helps third-party providers enhance service delivery and decreasing time-to-market. In addition, by making a wide range of design decisions (e.g., costly documentation and API usage), the platform provider monetizes the ecosystem resources to enhance **profitability**.

In general, by extending the service provision to the new sector by the support of the partners, the platform provider succeeds to achieve **strategic growth** while saving costs of supplying new resources. However, applying the PSE pattern demands high knowledge of the market to choose the partners and the effort to establish a network of partners.

Known Uses. Examples of application domains, where we identified the PSE pattern, are cloud computing, industrial design and simulation software, and security. Exemplary ecosystems are Citrix, SAP, and IFTTT.

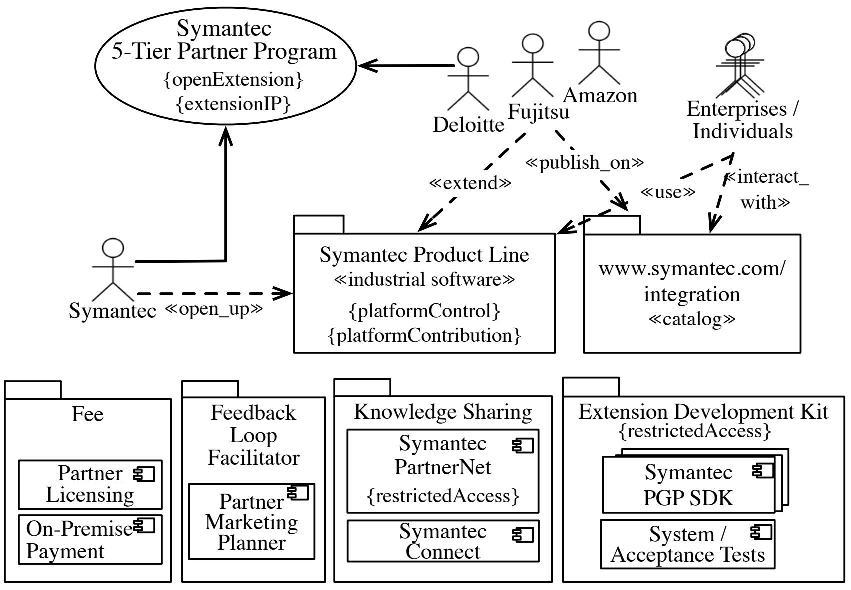
Related Patterns in Literature. Wei et al. [WFYZ20] propose a conceptual framework to include third-party providers as partners in an ecosystem in a two-stage process. The first stage describes how to form a *pool of resources* shared among the partners while the second stage is dedicated to the partner selection processes based on partners' collaborative capabilities. Although the authors do not denote their framework as a pattern, we consider it as a work related to the PSE pattern since it can be used to establish partner programs with several tiers of partnerships. A further related pattern to RSE is *service level agreement (SLA)* [SZZ$^+$18] (do not confuse with the SLA primary feature in Chapter 5). SLA facilitates a structured service level agreement for the API usage with measurable quality aspects.

Exemplary Software Ecosystem: Symantec

Symantec Corporation (today known as NortonLifeLock Inc.) was the provider of cyber security services such as secure email, web monitoring, and cloud security. Third-party developments are developed on the basis of platforms that are part of a software product line belong to Symantec. They are listed in a catalog on the web[8]. The platforms are closed source. Users are individuals and enterprises. Third-party developments are commercial. They can be highly critical for users, i.e., software failure can cause major financial and human-related harms. To use the platforms users have to pay a platform fee that follows an on-premise payment model.

Figure 7.8 presents the PSE pattern realized in the Symantec ecosystem in two ways: Figure 7.8a uses a graphical view to show the pattern, which is modeled by using the notation of a UML Component Diagram. Figure 7.8b refers to the context of the

[8] `www.symantec.com/integration`. Last Access: December 1, 2018.



(a) PSE in the Symantec Ecosystem, modeled using a UML Component Diagram

	Pattern Characteristics		**Symantec**
Context	**Domain Criticality**		Cyber security
	Commerciality		Commercial platforms & extensions, Costly licensing
Architectural Design Decisions	**Complementary Parnership**	**Trusted Partner**	Fujitsu, Deloitte, Amazon, etc. Symantec partner programs
	Platform Control	**Closed Source**	Closed source platforms
	Platform Contribution	**Closed Contribution**	No third-party contribution in the platforms
	Third-Party Development Intellectual Property	**Granted License**	Partner Licensing: Varying policies for different partner programs
	Third-Party Openness	**Closed Source**	Closed source third-party developments
	Fee	**Entrance Fee**	Partner Licensing
		Platform Fee	On-premise payment
	Feedback Loop Facilitator	**Market Analytics**	Planning, consulting, training Partner Marketing Planner tool
	Knowledge Sharing	**Documentation Framework**	PartnerNet including several partner portals
		Q&A Forums	Symantec Connect
	Extension Development Kit	**API Management**	Platform SDKs, e.g., Symantec PGP SDK
		Testing	System testing, Acceptance testing on user behalf

(b) Context and Decisions Made in the Symantec Ecosystem

Fig. 7.8 PSE in the Symantec Ecosystem

Symantec company and the decisions made as a part of the PSE pattern. To enhance readability, the table is reduced to the business and application aspects. Symantec provides a 5-tier partner programs to include different groups of third-party providers with different privileges in the ecosystem. Entering a partner program is subject to the fulfillment of certain portfolio requirements by third-party providers. Additionally, some partner programs with less privilege demand the third-party providers to pay an entrance fee. *Global Systems Integrator*[9] and *Technology Integration*[10] are examples of the partner programs.

Depending on a partner's tier, third-party developments are closed- or open-source. Norton is an example of acquisitions that are shared among Symantec and partners. In this case, platform control is granted, which means the partners own part of the source code of the software. Furthermore, the platform provider shares *third-party developments IP* with highly strategic partners such as Deloitte, Fujitsu, and Amazon. Thus, such partners can re-sell the common developments in their ecosystems. Third-party providers can decide on certain licenses. Different licenses are subject to different fees. While the permissions to re-sell the software can be given to certain partners, it does not automatically imply that the partners own the platform control. In this case, the functionality of the platforms is extendable using relevant SDKs offered for this purpose. For instance, Symantec PGP SDK gives access to the cryptographic functionality of the PGP platform. Testing features are offered to ensure service quality.

Knowledge sharing is enabled in the Symantec ecosystem by means of partner portals, Q&A forums, and market analytic features. Specifically, *PartnerNet*[11] is the partner portal, where partner tiers are managed. As a part of market analytics, *Partner Marketing Planner*[12] gives the partners a personalized marketing plan to enhance revenue generation. Furthermore, *Symantec Connect*[13] comprises a set of user and partner forums.

7.3.3 OSS-Based Software Ecosystem (OSE)

Open source software-based software ecosystem or shortly *OSS-based software ecosystem* (OSE) aids to attract developers of open-source software in order to cost-effectively create an ecosystem around an open-source platform. Several providers join together to form a foundation. The foundation plays the role of a platform provider. The

[9] `www.symantec.com/partners/programs/global-systems-integrator`. Last Access: December 1, 2018.
[10] `www.symantec.com/partners/programs/technology-integration-partners`. Last Access: December 1, 2018.
[11] `www.symantec.com/partners`. Last Access: December 1, 2018.
[12] `resource.elq.symantec.com/partner_resource_centre`. Last Access: December 1, 2018.
[13] `www.symantec.com/connect/`. Last Access: December 1, 2018.

members of the foundation can be individual volunteers or companies. The third-party providers are often non-commercially motivated developers. For example, they may want to gain a reputation within a community or to extend the platform for their own purposes [Han12]. However, commercially motivated companies can also be interested in using the platform. Third-party developments are in form of source code that is exchanged on a code repository. Additionally, another way to distribute the third-party developments is to offer an online store for the developments in form of executable software and add-ons. Furthermore, revenue generation is not generally high. However, by granting rights to the third-party providers to access the source code, the ecosystem becomes open for innovative developments.

OSS-Based Ecosystem Pattern Story

The following pattern story follows the OSE pattern demonstrated in Figure 7.9.

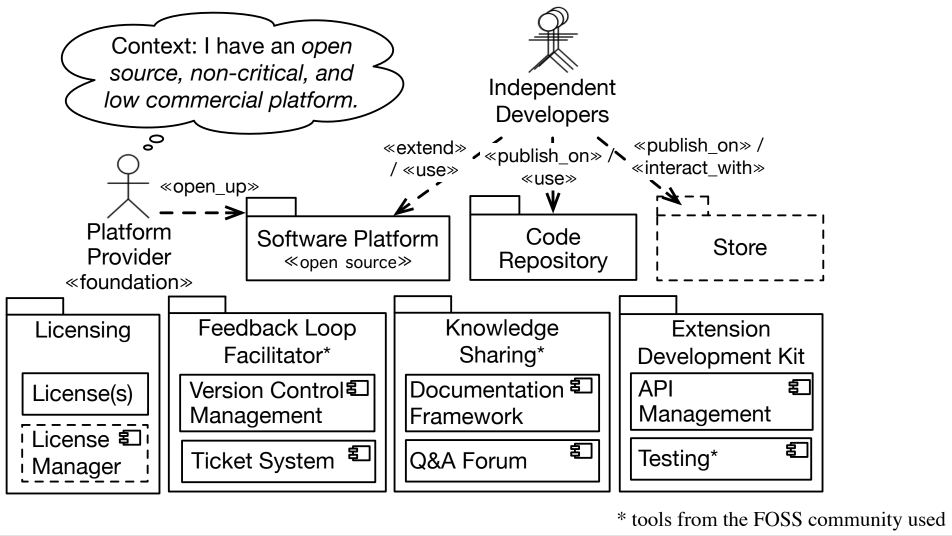
Context. A midsize to a large market of third-party developments resides on a code repository. In addition, the platform is open source and commerciality is low. Platform and third-party developments' functionalities are not critical to human lives.

Problem. The platform provider seeks long-term contributions from skilled developers in open-source software (OSS) communities. While ethical motivations are often the main drivers, the platform provider might be willing to inform him- / herself about recent market trends by growing a cost-effective platform in the direction of market trends. However, due to low commerciality, revenue is not generally returned to the ecosystem. Thereby, the questions are: **How to attract developers of open-source software to save costs of human expertise?** And, **how to save costs of software development?**

Forces. While the platform provider is interested in developing a software platform, which is strongly on the basis of third-party contributions, the platform needs to be modular and easy to modify, which makes **modifiability** a force. In addition, the interest to address market trends is a driver for the urge to develop innovative solutions [ZJD14]. Thus, **creativity** is another force. Moreover, as described in the problem section, **cost-effectiveness** becomes particularly important since revenue generation is barely a motivation to create the ecosystem. Thus, if the platform provider fails to include the developers, the platform will not grow. Therefore, we identify modifiability, creativity, and cost-effectiveness to be the main forces to develop

Open Source Software-Based Software Ecosystem (OSE): Innovation

Landscape



Context: I have an *open source, non-critical, and low commercial platform.*
Independent Developers
«extend» / «use»
«publish_on» / «use»
«publish_on» / «interact_with»
«open_up»
Platform Provider «foundation»
Software Platform «open source»
Code Repository
Store
Licensing
License(s)
License Manager
Feedback Loop Facilitator*
Version Control Management
Ticket System
Knowledge Sharing*
Documentation Framework
Q&A Forum
Extension Development Kit
API Management
Testing*
* tools from the FOSS community used

Contextual Factors of Platform Provider

Low Criticality
Low Commerciality

Architectural Design Decisions

- **VP1 User: Developer**
- **VP2 Niche Player Relation: Lone Wolf**
- **VP3 Release Model: Pull-oriented**
- **VP4 Symbiotic Relation: Commensalism**
- **VP6 Knowledge Sharing: Q&A Forum, Documentation Framework**
- **VP7 Complementary Partnership: Strategic Partner, Independent Developer**
- **VP8 Licensing: Public License**
- **VP9 Platform Control: Equity Ownership**
- **VP10 Platform Contribution: Platform Development**
- **VP11 Third-Party Development Intellectual Property: Granted License**
- **VP12 Third-Party Development Openness: Open Source**
- **VP16 Entrance Check: Leaks and Bugs, Policy Violation**
- **VP17 Store: Repository**
- **VP18 Feedback Loop Facilitator: Version Control Management, Ticket System**
- **VP19 Extension Development Kit: API Management, Testing, Choice of Programming Language**

Relevant Quality Attributes of Ecosystem Health

Creativity, Cost-Effectiveness, Modifiability

Fig. 7.9 OSS-Based Software Ecosystem:
An Architectural Solution for the Cost-Effective Enhancement of Innovation

a software platform by creating an ecosystem of third-party providers from open-source software communities around it.

Solution. **Attractiveness of open source platforms is highly associated with modifiability of software components in two different levels, i.e., governance and technical.**

At the governance level, this needs to be facilitated by giving third-party providers suitable ownership and decision rights to access the code. *License management* is one of the most used governance mechanisms in software ecosystems [AOJ17]. In general, a software license specifies usage permissions and conditions for such permissions. Thus, a license management becomes a way to control the protection of intellectual property. A well-balanced license, i.e., not too closed and not too open, is crucial for an ecosystem that is created around an open-source software platform because the license specifies the interplay between the capability of open source development and a business model [MI11]. Thus, an additional complexity is to find a suitable licensing that gives the developers rights to use the code while supporting the business model of another group of developers with commercial goals.

While deciding on a suitable licensing, the platform provider should consider conflict management and future access rights during ecosystem evolution. The choice of licensing needs to clearly answer the following questions without introducing any license conflict [SA12]: What are third-party providers' rights and obligations? Do the third-party developments already pertain to other licenses? Which part of the platform can be evolved or replaced? What are the dependencies between those parts and the rest of the platform?

Management of license conflicts can be done manually by staff, similar to *Eclipse Legal Process*, which is an examination procedure performed by Eclipse foundation to prevent publication of code with any license other than Eclipse Public License (EPL). Moreover, the platform provider might provide an automated environment to create and manage licenses including checking conflicts [MI11].

At the technical level, the ecosystem should provide third-party developers with software features that are necessary to access, reuse, and develop software components collaboratively. Figure 7.9 illustrates the architectural landscape. Version control management and ticket system facilitate a feedback loop between the developers. A version control management like Apache Subversion supports forking, branching, and merging the code. Additionally, a ticket system like Jira[14] helps to track issues and bugs.

To save costs of software, the platform provider uses a wide range of free and open source resources and tools on the web. Selenium[15] is an example of an open source testing framework that is widely used for web applications.

Consequences. Using multi-lines of development, third-party providers can fork, change, test, and deploy the source code without affecting the platform functionality. Furthermore, the version control management and ticket system can be used to track the recent changes applied directly to the source code. Thereby, **modifiability** is supported. Moreover, by eliminating the different types of entrance barriers, the third-party providers are given the freedom to enter the ecosystem and utilize the resources for creating third-party developments, which opens the ecosystem for **developers' creativity**. Generally, by attracting developers of open-source software and by relying on the resources of OSS communities, the platform provider can achieve the **cost-effective enhancement of innovation**. However, warranties and liability indemnity are not supported on the developers' behalf.

known Uses. Operating systems and software development are examples of the application domains for the OSE pattern. Further exemplary ecosystems are Apache Cordova, Ubuntu, and Zotero.

Related Patterns in Literature. A pattern language by Weiss [Wei18] provides an overview of engagement levels in open source businesses and identifies strategic decisions related to each level. The engagement levels start with *using* and *contributing*, and advance to *championing* and *collaborating*. Furthermore, two cross-cutting concerns, i.e., architecture and licensing, affect all levels. The pattern language captures fine-granular decisions that can be used to grow an open source business in a stepwise manner. For example, with regard to licensing, it suggests that a software owner, who possesses the full ownership of the code, can provide the same product under two different licenses, i.e., both open source and commercial (*Dual License*) or offer an enhanced version of the open-source software as commercial (*Dual Product*). In addition, *Public API* [Pub21] is concerned with the APIs used by an unknown number of API clients. It emphasizes providing detailed *API descriptions* and using security techniques for the purpose of access control.

[14]`www.atlassian.com/software/jira`. Last Access: January 10, 2022.
[15]`www.seleniumhq.org/`. Last Access: January 10, 2022.

Exemplary Software Ecosystem: Cloud Foundry

Cloud Foundry foundation is the provider of an ecosystem that conforms to the OSS-based software ecosystem pattern developed in this thesis. The foundation consists of several software companies, which contribute to the development of a platform. IBM, SAP, and Google are the exemplary members of the Cloud Foundry foundation. Cloud Foundry[16] that is the free and open-source software platform for cloud computing services. The services are non-critical and free to use. Third-party developers are other companies as well as individuals that are interested in extending the platform for their own needs. Figure 7.10 depicts the OSS-based ecosystem of Cloud Foundry. A GitHub repository comprises an under-development version of the platform. Furthermore, Cloud Foundry SDK[17] can be used to develop software on top of the platform. Additionally, the third-party developments can be published on the repository by forking the source code. At the time of this study, more than 4,000 forks have been created on the responsibility. A wide range of unit/integration/service quality testing features is provided on GitHub to test the scalability and performance of the services. Version control management is performed on the basis of GitHub services. Third-party providers can issue tickets for future improvement of the platform by using a ticket system provided by the platform provider called Pivotal Cloud Foundry Support Ticket System [Fou18].

In addition, The Foundry[18] is an online store that is used to distribute featured third-party services. The platform is under an Apache License. So, third-party providers have the right to use the software under the terms of this license. This is while the third-party developments are licensed based on the *Cloud Foundry Contributors' licenses*. With this respect, the foundation has the right to use the third-party developments but it does not own the right to change the source code.

The ecosystem supports knowledge sharing by means of documentation and Q&A forums. Cloud Foundry Docs[19] contains the technical documentation related to the software development on top of the platform. *Pivotal Knowledge Base*[20] is the Q&A forum for further discussions related to the usage of Cloud Foundry services. Moreover,

[16] `www.cloudfoundry.org`. Last Access: January 10, 2022.
[17] `github.com/cloudfoundry`. Last Access: January 10, 2022.
[18] `www.cloudfoundry.org/the-foundry`. Last Access: January 10, 2022.
[19] `docs.cloudfoundry.org/`. Last Access: January 10, 2022.
[20] `community.pivotal.io/s/communities`. Last Access: January 10, 2022.

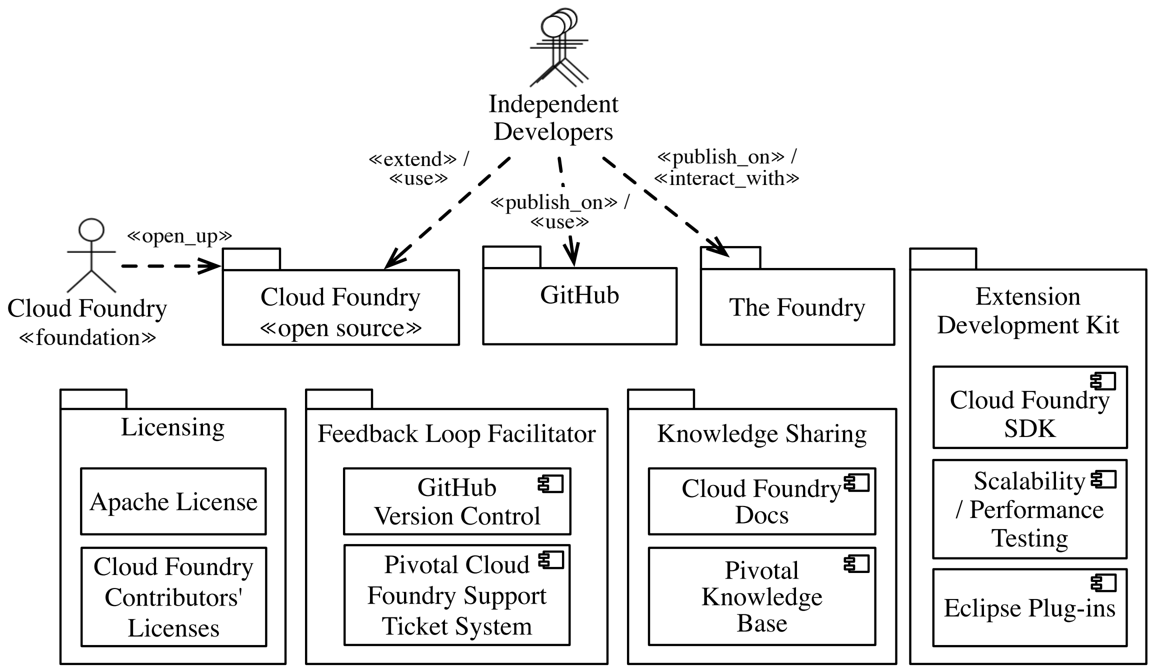

Independent Developers
«extend» / «use»
«publish_on» / «use»
«publish_on» / «interact_with»
«open_up»
Cloud Foundry «foundation»
Cloud Foundry «open source»
GitHub
The Foundry
Extension Development Kit
Cloud Foundry SDK
Scalability / Performance Testing
Eclipse Plug-ins
Licensing
Apache License
Cloud Foundry Contributors' Licenses
Feedback Loop Facilitator
GitHub Version Control
Pivotal Cloud Foundry Support Ticket System
Knowledge Sharing
Cloud Foundry Docs
Pivotal Knowledge Base

Fig. 7.10 OSE in the Cloud Foundry Ecosystem

Slack[21] and *Stack Overflow*[22] are the examples of public knowledge sharing services that are often used for topics related to the Cloud Foundry platform.

In a nutshell, each pattern characterizes a prominent strategic ecosystem development approach that is dominantly used in practice to gear the ecosystem functions to certain business objectives.

7.4 Pattern Relations

During the pattern mining activities conducted in Section 7.1, we noticed that the existing platform providers often create ecosystems of ecosystems while each of the ecosystems conforms to a pattern identified in this thesis. An *ecosystem of ecosystems* is an ecosystem that comprises other ecosystems [Jan20]. In this section, we discuss the relations between the three architectural patterns in ecosystems of ecosystems.

Figure 7.11 shows the relations between the architectural patterns. An ecosystem applying the PSE pattern can be a part of another ecosystem that applies the RSE pattern. Similarly, an ecosystem applying the OSE pattern can be a part of an ecosystem that applies the RSE pattern as well. The results of both situations are

[21] `slack.cloudfoundry.org`. Last Access: January 10, 2022.
[22] `stackoverflow.com`. Last Access: January 10, 2022.

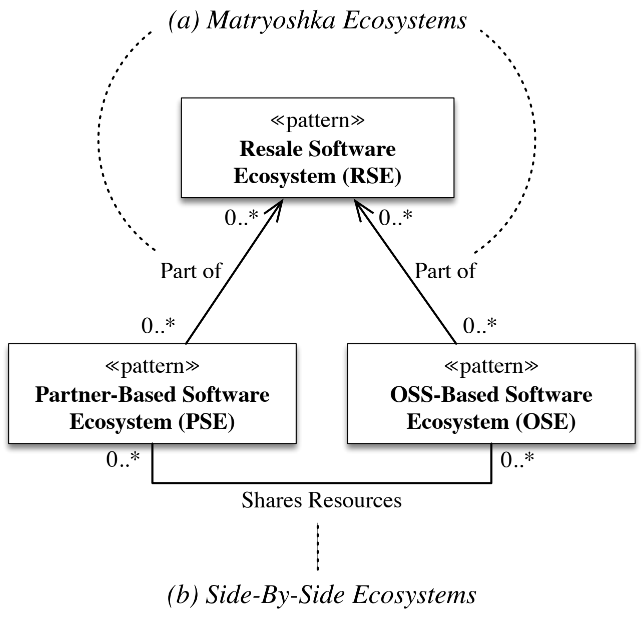

(a) Matryoshka Ecosystems
«pattern»
Resale Software Ecosystem (RSE)
0..*
0..*
Part of
Part of
0..*
0..*
«pattern»
Partner-Based Software Ecosystem (PSE)
«pattern»
OSS-Based Software Ecosystem (OSE)
0..*
0..*
Shares Resources
(b) Side-By-Side Ecosystems

Fig. 7.11 Pattern Relations

ecosystems that surround other ecosystems, i.e., ecosystems of ecosystems. We refer to this form of ecosystems of ecosystems as *matryoshka ecosystems* since an ecosystem completely surrounds another ecosystem. In addition, an ecosystem applying the PSE pattern and an ecosystem applying the OSE pattern can share some resources by both being the building blocks of a larger ecosystem. We refer to this situation as *side-by-side ecosystems*. In the following, we discuss the two situations resulting in ecosystems of ecosystems more in detail.

(a) Matryoshka Ecosystems

In the case of *matryoshka ecosystems*, a larger ecosystem consists of another ecosystem, which is smaller in terms of user-base, the market of third-party developments, or the number of third-party providers. The software platform is the common asset shared between the ecosystems. However, the ecosystems might have different architectural characteristics, e.g., openness policies, extension development kit, fees, etc. In the following, we discuss two common designs of matryoshka ecosystems.

- **An ecosystem applying the PSE Pattern is a part of another ecosystem applying the RSE Pattern**: Increasing growth of the partner community and store alongside each other is considered as a natural evolution of successful ecosystems [PSS+16]. Our study also shows that an ecosystem applying the PSE pattern can be surrounded by another ecosystem that applies the RSE pattern. This happens when the number of partners increases and the platform provider includes rating, reviewing, and raking features to improve extension discoverability. In addition, by demanding registration, instead of a direct partnership, the platform provider manages membership of the growing number of third-party developments. 5% of the ecosystems, e.g., Intuit QuickBooks[23], relate these two patterns.

- **An ecosystem applying the OSE Pattern is a part of another ecosystem applying the RSE Pattern**: An OSE ecosystem can be used to grow an open-source platform by means of the partners of the foundation and contributions from individuals whereas an RSE ecosystem is created to control the growth of the ecosystems at the level of the mass number of users, who are interested in ready-to-use software. The larger ecosystem is used to distribute high quality third-party developments among a large group of users by making them discoverable using rating, reviewing, and ranking features on a store. The smaller ecosystem is used to develop the core part of the platform on a code repository. In some cases, third-party developments on the store are free and open source as well, such as *Mozilla.org* and *LibreOffice.org*. In addition, the ecosystem may offer commercial third-party developments besides the free services, despite the platform itself being open-source. An example of this situation can be seen in *Eclipse Marketplace*[24] that mainly offers free and open-source plug-ins. However, commercial plug-ins can also be found in the marketplace.

(b) Side-by-Side Ecosystems

Side-by-side ecosystems describe a situation when a provider creates different ecosystems around different software platforms. In this situation, the provider is the owner of a software product line, where different ecosystems are built around each of the products.

Our results show that a frequently applied design in practice is the combination of the PSE and OSE patterns in the side-by-side ecosystems. In this case, one platform is free and open-source, which includes the base functionality and is used for investing in the open-source community, e.g., by making it available for enterprises, which are interested in using open-source software. The platform of the other ecosystem is similar to the fee platform in terms of having the base functionality, but it includes further complex features for advanced use-cases. In the ecosystem applying the PSE pattern, business services such as consulting, workshops, marketing resources are offered to

[23] `quickbooks.intuit.com/`. Last Access: January 10, 2022.
[24] `marketplace.eclipse.org`. Last Access: January 10, 2022.

the third-party providers. The users of the free and open-source platform are called *community* while the users of the commercial platform are referred to as *partners*.

Pivotal is an example of the platform providers providing side-by-side ecosystems. the provider of a *Partner-based Ecosystem* The ecosystem provided by Pivotal consists of two ecosystems. One is created around the Pivotal Web Services (PWS) that matches the PSE pattern whereas the ecosystem built around Cloud Foundry conforms to the OSE pattern. PWS and Cloud Foundry are both products of the same software family, i.e., Pivotal software.

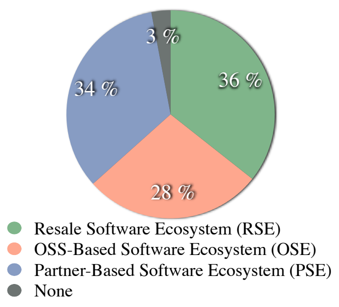
In summary, the relations between the patterns help us to obtain a better understanding of the relations between the ecosystems and how platform providers create ecosystems of ecosystems. We have discussed that the platform providers create ecosystems of ecosystems for different purposes while the main reasons are to reach a different group of users by popularizing a family of software platforms among enterprises (side-by-side ecosystems) or to develop a large user-base (matryoshka ecosystems). The list of the ecosystems applying a combination of the patterns can be found in Appendix C.

7.5 Analysis of Results

In the previous sections, we presented three architectural patterns for software ecosystems that were the result of investigating 111 existing ecosystems from a diverse range of application domains. We discussed the relations between the patterns and their relationship with the concept of ecosystems of ecosystems. In this section, we discuss our further findings with respect to a) the popularity of the patterns in practice and b) the distribution of the patterns among different application domains.

7.5.1 Pattern Popularity

Figure 7.12 shows the popularity of the patterns and their combinations, i.e., how often the patterns and their combinations are found in 111 ecosystems during our study. As shown in Figure 7.12a, the occurrences of the three patterns in our list are fairly close to each other, i.e., RSE (36%), PSE (34%), and OSE (28%), while RSE being slightly more popular. In addition, a small fraction of the ecosystems, i.e., 3%, is classified as *none* since we have not detected any of the three patterns in those ecosystems. We noticed that these are small ecosystems with small numbers of third-party developments. The third-party developments are usually distributed in the stores of other ecosystems that are larger in terms of market size.



(a) Popularity of Single Patterns

Ecosystems with any Combination of the Patterns	25 %
Matryoshka Ecosystems (OSE part of RSE)	14 %
Matryoshka Ecosystems (PSE part of RSE)	6 %
Side-By-Side Ecosystems (PSE and OSE)	5 %

The percentages are based on our study of 111 ecosystems.

(b) Popularity of Pattern Combinations

Fig. 7.12 Pattern Popularity

A combination of the patterns can be found in one-quarter of the ecosystems of our list as shown in Figure 7.12b. Notably, the number of matryoshka ecosystems surpasses the side-by-side ecosystems. This is while a main part of the matryoshka ecosystems applies a combination of the OSE and RSE patterns. One reason for this situation can be the rising popularity of open-source software platforms in recent years that makes such platforms attractive for third-party providers [MS20].

7.5.2 Domain Analysis

One key architectural challenge is to understand the characteristics of ecosystem architecture with respect to different application domains as discussed in Section 1.2. We analyze our list of ecosystems regarding their application domains. To identify prominent application domains, we refer to the definition of an application domain given In Section 2.4.1. Specifically, we identify the application domains in which the ecosystems have been created and the distribution of the patterns across the application domains as shown in Figure 7.13. In general, six dominant application domains have been identified, i.e., *OS and web browser*, *cloud computing*, *enterprise software*, *Internet-of-Things*, *software development*, and *graphic and simulation*. The term *others* refers to a group of application domains with less popularity in our list. Examples of such domains are shipping and logistic software and file-sharing. Furthermore, the highest numbers of ecosystems are created around enterprise software and cloud computing platforms. The ecosystems in these domains show to mostly match with the RSE and PSE patterns. The lowest numbers of ecosystems are found around Internet-of-Things platforms. A reason might be the novelty of this field and the fact that the maturity

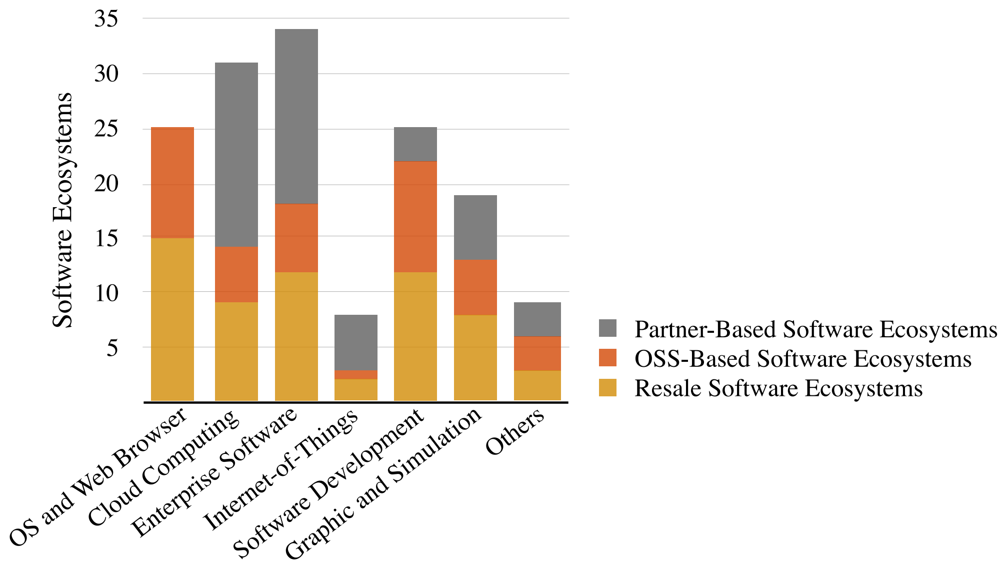


Fig. 7.13 Pattern Distribution Across Application Domains

of the software platforms is a key requirement to create software ecosystems around them in the first place [BB10a, JS17a]. The indicators of the platform maturity are a suitable amount of users/third-party providers, market share, etc.

In addition to the results discussed above, we used our study of the existing ecosystems to address current questions raised in the literature. As discussed in Section 3.2.1, some work in literature considers software ecosystems as a natural growth of software product lines. However, other work argues that software ecosystems can also be built around a single software product [BdA^{+}13]. To address this discussion, we examined our list of ecosystems with respect to the kinds of software platforms. Our results show that 67% of the ecosystems open a software product line to third-party providers, which is the majority in our list. Interestingly, almost half of this amount are the ecosystems that conform to the RSE pattern. In many cases like Mozilla and Adobe, the product lines include a family of different software products. In some other cases like WordPress and Docker, the product lines consist of similar software products that are different in terms of licensing or the number of functionalities offered by them, e.g., free and enterprise editions of a product. However, a considerable number of ecosystems in our list are built around single software products. Two examples are IFTTT and Dolibarr with up to 500 third-party developments.

7.6 Summary and Scientific Contributions

In this chapter, we presented three architectural patterns in order to facilitate the methodical development of software ecosystems. We called the architectural patterns *resale software ecosystem*, *partner-based software ecosystem*, and *open source software-based ecosystem*. We discussed that each architectural pattern supports different quality attributes of ecosystem health and business objectives by suggesting a different arrangement of human actors and choices of architectural design decisions.

The knowledge of the architectural patterns introduces platform providers the practice-proven reusable architectural designs. It helps them to decide when to apply any of these patterns, or how to transform their existing ecosystems by applying one or a combination of the patterns. In addition, this should help third-party providers to decide on the suitability of an ecosystem before entering it, by benchmarking existing ecosystems against the patterns, identifying key features of ecosystems, understanding enhanced and degraded quality attributes, and deciding on the suitability of the ecosystems with respect to their goals. Therefore, they will be able to save efforts of participating in inefficient ecosystems that miss critical features. In summary, the contributions of this chapter can be summarized as:

- We are the first to perform an investigation of more than 100 existing ecosystems.
- For the first time, we identified architectural patterns for software ecosystems.
- The patterns capture three ecosystem designs applied by the existing platform providers in practice.
- We developed the stories for the architectural patterns that describe when and how to apply the architectural patterns.
- We identified the relations between the patterns that result in the creation of ecosystems of ecosystems.

Chapter 8

A Modeling Framework to Design and Analyze Ecosystem Architecture Models

In this chapter, we develop a modeling framework for designing and analyzing ecosystem architectures in response to RQ6, RQ7, and RQ8 introduced in Chapter 4. The modeling framework consists of three design instruments: a) *a design process*, b) *a domain-specific modeling language*, and c) *an architectural analysis technique*. These instruments can be used to create models of ecosystem architectures and assess the suitability of the architectures with respect to the quality attributes of ecosystem health.

This chapter is structured as follows. Section 8.1 presents our language engineering approach. Section 8.2 introduces an illustrative example that we use throughout this chapter. Section 8.3 refers to a rule-based pattern matching approach that is the core of the architectural analysis technique. Section 8.4 proposes the design process for the stepwise development and analysis of ecosystem models. Section 8.5 presents the modeling language, followed by Section 8.6 that elaborates on the architectural analysis technique. Section 8.7 is dedicated to the implementation of the modeling framework in a tool. Section 8.8 discusses and outlines the scientific contributions of this chapter. Our findings are based on the publications [JSK$^+$20, SE20].

8.1 Language Engineering Approach

Modeling software ecosystems is a challenging and complex task since software ecosystems are variability-intensive systems with interrelated organizational, business, and technical aspects that should be considered at the design time [Man16]. As mentioned in Section 1.2, using general purpose languages like UML leads to laborious approaches. A modeling framework that integrates the knowledge of variabilities and spans across the interdisciplinary architectural aspects can ease the complexity of ecosystem development.

On the basis of the domain-specific knowledge identified in Chapters 5-7, we develop an ecosystem modeling framework that follows the principles of model-driven engineering (MDE) presented in Section 2.5. The modeling framework comprises a domain-specific modeling language, architectural analysis technique, and design process. The modeling language is defined at the meta-layer M2. For the definition of meta-layers in MDE, see Section 2.5. It is grounded on a *domain model* that is a metamodel (a.k.a abstract syntax and its semantics) including the key design elements of software ecosystems. The domain model is based on the knowledge of the architectural commonalities and variabilities that we identified in Chapters 5 and 6. Furthermore, we developed *visual notation* (a.k.a. concrete syntax) to enable the direct realization of the domain-specific knowledge in models at the M1 layer. These models can be a basis for the development of software ecosystems in real-world.

The methodical development of software ecosystems is facilitated by introducing a *design process* and *architectural analysis technique* that are derived from the knowledge of the architectural patterns presented in Chapter 7. The methodical knowledge resides on the meta-layer M2 since it consists of the guidance that can be used to create concrete ecosystem models. As discussed by SAUER [Sau11] in his thesis, meta-methods can be defined for the systematic development and tailoring of the architecture of software systems. In this context, a method comprises certain software development tasks that are specifically defined. During the workflow of the method, the tasks are performed as activities.

Notably, while the goal of the methodical knowledge is to enhance the systematic creation of software ecosystems, this knowledge can be applied in real-world to improve existing ecosystems as well. This gives rise to the necessity of an analysis technique to check the conformance of existing ecosystems in the M0 layer with the SecoArc framework in M2. The general idea is to be able to check suitability of the ecosystems in M0 by with respect to the framework. The ecosystem architectures can be modeled using the SecoArc visual notation. We propose an architectural analysis technique that can be used to assess the suitability of ecosystem designs by checking the conformance of the ecosystem models (M1) with respect to the SecoArc framework.

8.2 Illustrative Example

Citrix[1] is a provider of cloud computing platforms. The platforms are closed source. They provide server and desktop virtualization functionalities. Citrix cloud computing services can be run on the basis of the platforms. In addition, Citrix collaborate with strategic partners like Microsoft, Cisco, and Google. Third-party software services developed by the partners are publish on an online store namely Citrix Ready Marketplace[2]. While individuals can buy and use Citrix services, enterprises are a main group of users that work with the Citrix platforms and services.

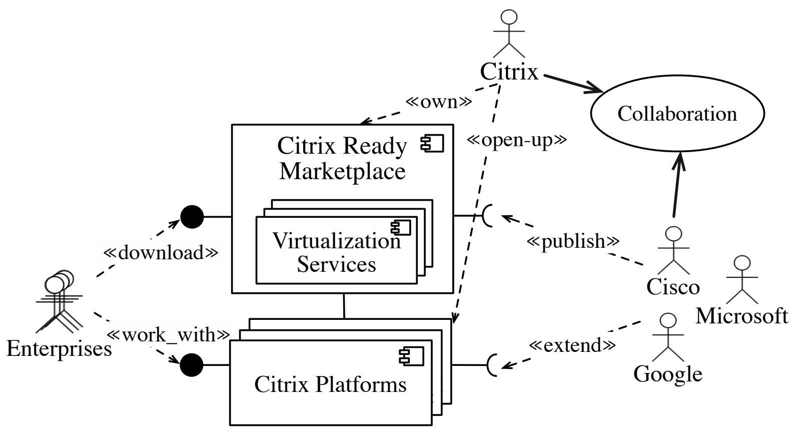


Fig. 8.1 Part of the Citrix Ecosystem

We use the Citrix ecosystem to illustrate the concepts of the SecoArc modeling framework. With this respect, firstly, we imagine a situation that the Citrix company introduced uses the SecoArc design process to scale it business by involving a mass number of independent developers. We discuss the SecoArc design process in the context of our example. After that, we design parts of the Citrix ecosystem that are relevant to the thesis' concept using the SecoArc modeling language. Finally, we analyze the exemplary ecosystem architecture bey means of the SecoArc architectural analysis techniques.

8.3 Rule-Based Pattern Matching

Existing software ecosystems that have been already created may possess architectural deficiencies that need to identified and improved. We enable analysis of the ecosystem

[1]`https://www.citrix.com`. Last Access:January 10, 2022.
[2]`https://citrixready.citrix.com`. Last Access:January 10, 2022.

Table 8.1 SecoArc Architectural Analysis Technique: Rule-Based Pattern Matching

Ecosystem Architecture			**Resale Software Ecosystem (RSE)**	**Partner-Based Ecosystem (PSE)**	**OSS-Based Ecosystem (OSE)**
Platform Provider's Organizational Context		**Business Objectives**	Business Scalability	Strategic Growth	Innovation
	Context- ? *Matching* ?	**Contextual Factors**	- Large Company - Large Market of Services	- High Commerciality - High Criticality	- Low Commerciality - Low Criticality
Design Decisions Made	*Decision-* ? *Matching* ?	**Design Decisions**	- Rating - Reviewing - Ranking - Testing Framework - Integrated Development Environment (IDE) - Issue (Bug) Tracking - Multi-Lines of Development - Documentation Framework - Service Execution	- Platform Fee - Monetized Documentation and APIs - Entrance Fee - Commercial Licensing - Testing Framework - Partner Program - Consulting Services - Market Analytics - Bring Your Own License (BYOL)	- Open Entrance - Open Platform - Open Publish - Free Platform - Free Licensing - Version Control Management - Choice of Programming Languages
		Quality Attributes	Interoperability Sustainability Robustness	Profitability Niche Creation Customer Satisfaction	Creativity Cost-Effectiveness Modifability

architecture by developing an architectural analysis technique based on the knowledge of the architectural patterns presented in Chapter 7. For this purpose, we transform the design decisions of each pattern to rules that should be checked by considering the architecture. The analysis applies a rule-based pattern matching to check whether the rules are applied.

The pattern matching approach consists of two parts, i.e., *context matching* and *decision matching* as shown in Table 8.1. To make the usage of the architectural analysis technique as effective as possible, we characterize each pattern by including the most critical design decisions that differentiate the patterns from the other ones.

During the context matching, the contextual parameters provided by the platform provider are compared with the contextual factors of the patterns. To this end, the analysis decides which patterns matches the platform provider's contextual factors the most.

The decision matching matches the decisions of the ecosystem architecture with the decisions of the patterns. Each decision is a rule. The tabular representation of the patterns on the right side of Table 8.1 shows that each pattern is characterized using a business objective, a set of design decisions, and a set of quality attributes. This means the patterns help platform providers with certain contextual factors address the quality attributes of ecosystem health. The concrete sets of design decisions in the middle determine the linkage between the ecosystem architecture and the quality attributes. During the pattern matching the rules are checked and the extent of matching is calculated. For instance, having *rating* in the architecture is a rule when the analysis calculates the matching of the architecture with the RSE pattern. The total number of fulfilled RSE-related rules reflects the extent that the architecture matches to the RSE pattern.

8.4 A Design Process

To provide platform providers with stepwise guidance to design concrete ecosystem architectures, we present a design process that captures the main procedure of designing and analyzing software ecosystems using the SecoArc framework. The design process is based on the knowledge of the architectural patterns that we identified in Chapter 7.

Figure 8.2 shows the SecoArc design process using the notation of UML Activity Diagrams. The process consists of the following steps : *Specifying context and goals*, *applying patterns*, *manually modeling the architecture*, *analyzing the architecture*, and *comparing architectures*. It starts with specifying the context and goals, where the platform providers determine the organizational characteristics and choose among the business objectives and/or quality attributes of ecosystem health. In the next step, the providers should design an ecosystem. This can be performed either by applying the architectural patterns or without a direct application of the patterns. If the architectural patterns are applied, the architecture can be tailored in another step by manually adding further elements. If the ecosystem architecture is going to be designed without applying the pattern, it needs to be first manually designed. In the next step, the architecture can be analyzed with respect to the business objectives and quality attributes specified initially. The platform providers decide whether further ecosystems should be designed and compared with each other.

The decision on one of the design approaches depends on the platform provider and specifically the willingness to follow the guidelines provided by the architectural patterns and whether an architecture already exists. While the architectural patterns include concrete guidelines on the problem, solution, forces, etc., an approach relying on the architectural analysis directly reveals to which extent an existing design matches the architectural patterns, which is more efficient in terms of time. However, the guidelines on how to apply the patterns are missing.

In the following, we elaborate on the process steps. In addition, we refer to the illustrative example introduced in Section 8.2 to show how a platform provider'

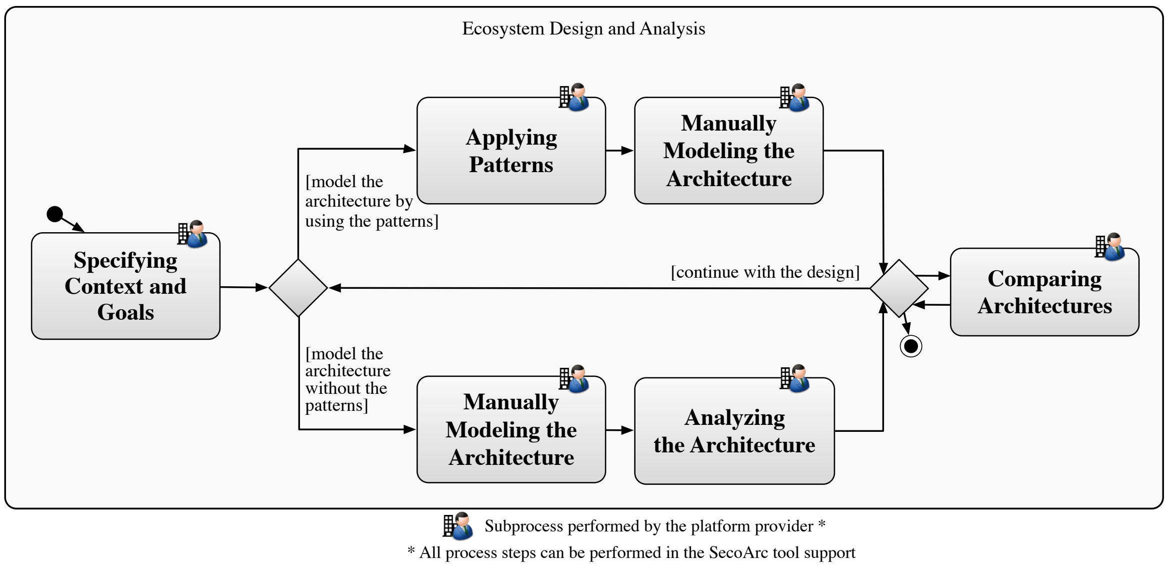

Ecosystem Design and Analysis
Specifying Context and Goals
[model the architecture by using the patterns]
Applying Patterns
Manually Modeling the Architecture
[continue with the design]
Comparing Architectures
[model the architecture without the patterns]
Manually Modeling the Architecture
Analyzing the Architecture
Subprocess performed by the platform provider *
* All process steps can be performed in the SecoArc tool support

Fig. 8.2 SecoArc Design Process: Ecosystem Design and Analysis

approach to enhance the architecture towards the specified goal can look like. Thereby, we imagine the Citrix company directly applies the architectural patterns. This approach includes *specifying context and goals* and *applying patterns* process steps in Figure 8.2. In addition, we consider an approach that Citrix manually models the architecture and afterward analyze it with respect to the goal, which comprises *specifying context and goals*, *manually modeling the architecture*, and *analyzing the architecture* process steps.

(I) Specifying Context and Goals:

Platform providers' organizational characteristics and business objectives define a major part of rules and regulations that constrain the architecture of software ecosystems [BPT$^+$14, JC13]. Therefore, designing an ecosystem begins with specifying these aspects. Here, the platform provider answers the question *who am I?* The SecoArc tool supports the context specification.

Figure 8.3 describes the process of specifying context and goals, first by specifying the organizational context. At this stage, concrete values for the contextual factors introduced in Section 7.2 should be provided.

In the next step, goals are defined. In this context, the goals are critical business objectives and quality attributes of ecosystem health that are supported by the framework. Hereby, the platform provider answers the question *what do I want to be?*

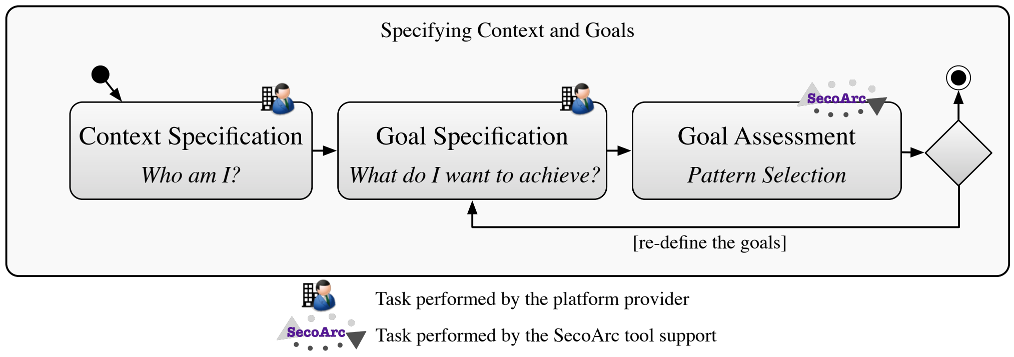

Specifying Context and Goals
Context Specification
Who am I?
Goal Specification
What do I want to achieve?
Goal Assessment
Pattern Selection
[re-define the goals]
SecoArc
Task performed by the platform provider
Task performed by the SecoArc tool support

Fig. 8.3 Context and Goal Specification Process

Afterward, by performing a goal assessment, the goals are assessed with respect to the given organizational context. Specifically, the goal assessment is the result of a context- and goal matching that are performed by means of the pattern matching presented in Section 8.3. During the goal assessment, a pattern recommendation based on the organizational context of the platform provider is performed that is the result of the context-matching. We refer to this pattern as *starting pattern*. Furthermore, the goals are expressed in terms of a *target pattern*. Using this knowledge, the platform provider can decide whether the business objectives and quality attributes should be reconsidered or not.

Example of Specifying Context and Goals First, Citrix context needs to be specified. Citrix is a large company that owns a large market of services. The company provides commercial software. While part of the services is not safety-critical. There are Citrix services that are used in health care and public sector domains that we consider critical because they require a high degree of reliability. The next activity is to specify the goals. As mentioned in Section 8.2, in our exemplary scenario, Citrix would like to scale the business by involving independent developers. Thereby, the goal is business scalability.

By performing the goal assessment using the SecoArc tool support, the context-matching presented in Section 8.3 is applied that results in the identification of RSE and PSE as starting patterns. Furthermore, the goal of business scalability is aligned with the RSE pattern. Thereby, RSE is the target pattern as well.

(II) Applying Patterns

Designing a software ecosystem using the knowledge of the architectural patterns depends on the results of the context and goal specification, which determines what the starting and target patterns are.

A pattern can directly be applied if the starting and target patterns are the same, i.e., the result of the context-matching is the same as the pattern that would be aligned with the platform provider's goal. Otherwise, a combination of the patterns needs to be applied.

Figure 8.4 shows the concrete process of applying the patterns. The general approach is to begin with the starting pattern as initial architecture and gradually integrate the components of the target pattern into the architecture. The process of integrating the target pattern depends on what the starting and target patterns are, which follows the knowledge of the pattern relations discussed in Section 7.4.

Both the OSE and PSE patterns are highly based on the openness policies whereas the OSE pattern generally characterizes an open ecosystem while PSE concerns highly protected ecosystems. Therefore, applying each of these patterns includes considering the openness policies and replacing components of the architecture with the components of the target pattern that help achieve the required openness.

Furthermore, the OSE and PSE patterns can be applied in two different ecosystems that are run by the same platform provider and share a high amount of resources. In this case, the platform provider should specify the shared resources among the ecosystems and the processes that the third-party providers need to follow in order to move between the ecosystems.

On the other hand, the goal of applying RSE is to gain business scalability. The pattern is relevant for mature ecosystems that have grown successfully to include a mass number of third-party providers. This concerns an evolutionary scenario, which requires the components of the RSE pattern to be added to the existing architecture. Figure 8.4 excludes the scenario, where the starting pattern is RSE and the target pattern is either PSE or OSE because, as already mentioned above, an ecosystem that applies the RSE pattern has already applied an OSS or PSE in the past.

Example of Applying Patterns The result of specifying context and goals in the previous process step was the PSE and RSE patterns as starting patterns, and the RSE pattern as target pattern. By carefully considering the Citrix ecosystem, we notice that a PSE pattern currently exists in the architecture. This is not the case regarding the RSE pattern. This can be easily seen since the third-party providers are

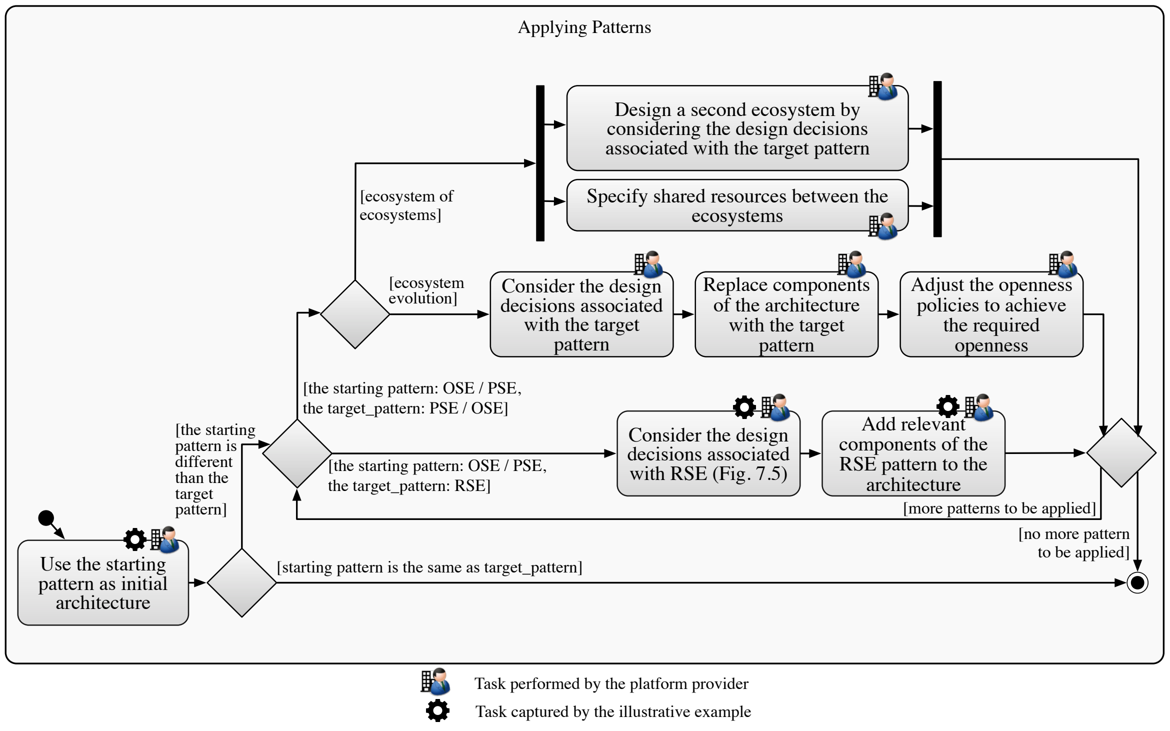

Applying Patterns
Design a second ecosystem by considering the design decisions associated with the target pattern
Specify shared resources between the ecosystems
[ecosystem of ecosystems]
[ecosystem evolution]
Consider the design decisions associated with the target pattern
Replace components of the architecture with the target pattern
Adjust the openness policies to achieve the required openness
[the starting pattern: OSE / PSE, the target_pattern: PSE / OSE]
[the starting pattern is different than the target pattern]
[the starting pattern: OSE / PSE, the target_pattern: RSE]
Consider the design decisions associated with RSE (Fig. 7.5)
Add relevant components of the RSE pattern to the architecture
[more patterns to be applied]
[no more pattern to be applied]
Use the starting pattern as initial architecture
[starting pattern is the same as target_pattern]
Task performed by the platform provider
Task captured by the illustrative example

Fig. 8.4 Pattern Application Process

only partners and the ecosystem is fairly closed, e.g., the source code of the platform and third-party developments are closed, fees are associated with using the services, and the services are tested. To apply a pattern, the goals should again be taken into consideration. PSE is the relevant starting pattern while the RSE still needs to be suitably applied in the architecture.

By clarifying the starting and target patterns, the process of applying the pattern can begin. The tasks that we describe in the following are marked in Figure 8.4 using a gear symbol. Initially, the Citrix company should use the PSE pattern to design an initial architecture. The target pattern remains RSE. Afterward, the design decisions associated with the target pattern should be applied in the architecture. At this point, the platform provider needs to refer to the definition of the target pattern presented in Chapter 7, i.e., RSE (cf. Figure 7.5).

In our exemplary scenario, Citrix should add the RSE pattern by identifying the decisions that are currently missing in the architecture. Table 8.2 shows the decisions concerning the PSE and RSE patterns that are (not-) realized in the architecture.

Platform fee is defined as a pay-as-you-go model that allows the partners to pay on-demand for the Citrix services. As a part of market analytics, Citrix mailing servers

Table 8.2 Examples of Design Decisions in the Citrix Ecosystem

	Variation Point	Variant	Citrix
Architectural Design Decisions	**Complementary Partnership**	**Trusted Partner**	Citrix strategic partners: Microsoft, Cisco, Google, etc. Citrix partner programs: Service providers, System Integrators, SaaS Referral Partner
	Platform Control	**Closed Platform**	Closed source platforms
	Platform Contribution	**Closed Contribution**	No direct contribution by partners
	Third-Party Development Intellectual Property	**Granted License**	Rights to distribute third-party developments by the partners
	Third-Party Development Openness	**Closed Source**	Closed source software services
	Fee	**Entrance Fee**	Periodic payment depending on a partner programm
		Platform Fee	On-demand pay-as-you-go for users
		Documentation Fee	Fees are associated with partner programs
	Store	**Catalog**	Citrix Ready Marketplace
	Entrance Check	**Leak and Bugs Tracking**	— (not realized)
		Policy Violation	— (not realized)
		Portfolio Requirements	A certain market share and annual revenue generation required
	Feedback Loop Facilitator	**Rating**	— (not realized)
		Reviewing	— (not realized)
		Ranking	— (not realized)
		Market Analytics	Citrix Marketing Concierge
	Knowledge Sharing	**Documentation Framework**	Citrix Partner Central, Citrix Product Documentation
		Q&A Forums	Citrix Discussions, Citrix User Group Community
	Extension Development Kit	**API Management**	API Gateway supporting multi-lines of development for different Citrix platforms
		IDE	— (not realized)
		Testing Framework	Citrix Quick Launch Tool, Citrix Workspace
	Service Execution	**Data Center**	Citrix Multi-Datacenter

support communication between the users and partners. Another example is Citrix Marketing Concierge, which is to manage email campaigns, webinars, and roadshows. In addition, knowledge sharing is enabled by Citrix Product Documentation[3] and Partner Central[4]. Using the partner central, the partners access different partner programs. Furthermore, Citrix Discussions[5] and User Group Community[6] are the forums respectively used by the partners and users. Citrix Multi-Datacenter[7] handle the tasks related to the orchestration of data when services are executed.

[3]`https://docs.citrix.com`. Last Access:January 10, 2022.
[4]`https://www.citrix.com/partnercentral`. Last Access:January 10, 2022.
[5]`https://discussions.citrix.com`. Last Access:January 10, 2022.
[6]`https://www.mycugc.org`. Last Access:January 10, 2022.
[7]`https://discussions.citrix.com`. Last Access:January 10, 2022.

By comparing the Citrix ecosystem with the knowledge of the RSE pattern, the design decisions that are missing can be identified and further considered to be added in the architecture. These decisions are marked as *not realized* in Table 8.2. For instance, currently, the store lacks a rating and reviewing features. Citrix may consider to provide indicators for the quality of services in the store by adding these features.

Manually Modeling the Architecture

An ecosystem architecture or parts of the architecture can be modeled manually. As shown in Figure 8.2, it might be the case that, after specifying context and goals, the architecture is manually designed without using the knowledge of the architectural patterns. Another case is when the architecture resulting from applying patterns is tailored by adding further architectural elements.

Using the SecoArc tool support, the ecosystem architecture can be manually modeled by a) using the SecoArc knowledge base and b) creating custom architectural elements. In Section 8.5, we introduce the *SecoArc modeling language* that facilitates the design of the SecoArc knowledge base, i.e., the architectural commonalities and variabilities, in models. In addition, the language allows the creation of custom elements that are not part of the SecoArc knowledge base.

(III) Analyzing the Architecture

Once the ecosystem architecture is modeled, it can be analyzed with respect to the quality attributes of ecosystem health and the business objectives addressed by the patterns. In general, the architectural analysis provides an answer to the platform provider's question, i.e., *where do I stand with this architecture?* We present this process step in Section 8.6, where we discuss further results relevant to the architectural analysis.

The results of the architectural analysis are not different than the initial goal assessment if the starting and target patterns were the same and the architecture is not changed later. However, if a combination of the patterns is applied, the analysis results reveal how the architecture is changed with respect to the fulfillment of business objectives and quality attributes.

(IV) Comparing Architectures

The platform provider can design and compare more than one architecture when further architectures need to be taken into account. To model more than one ecosystem, the platform provider needs to start the design process from the beginning. In this case, the organizational context may remain the same while new business objectives or quality attributes might need to be taken into account. Another possibility would be to design a new ecosystem for the previously defined business goals and quality attributes.

The subpart of the architectural comparison relies on the SecoArc architectural analysis technique. The perspectives introduced in Section 8.6.1 facilitate the comparison of more than one ecosystem.

8.5 Ecosystem Modeling

In this section, we present a domain-specific modeling language to design ecosystem architecture. As discussed in Section 2.2.2, the term *ecosystem architecture* denotes some parts of an enterprise architecture that are concerned with the platform providers' strategy of opening a platform to external parties.

The objectives of the SecoArc modeling language are a) to facilitate explicit expression of the knowledge of architectural commonalities and variabilities developed in Chapters 5 and 6, and b) to provide a basis for architectural analysis with respect to the quality attributes of ecosystem health. The modeling language is applicable during design of ecosystem architecture shown in Section 8.4.

The modeling language is grounded on a domain model and visual notation. We present the domain model in Section 8.5.1, followed by introducing the visual notation in Section 8.5.2.

8.5.1 A Domain Model for Software Ecosystems

Figure 8.5 presents the SecoArc domain model for software ecosystems using the notation of UML Class Diagrams. The cardinalities of aggregations and compositions in the diagram are respectively *0..** and *1..**, otherwise it is mentioned. The domain model consists of three types of architectures that reflect the architectural aspects of software ecosystems. We present the constructs of these architectures in the following order: First, we discuss the social and business architecture. We present these aspects in one UML package due to the high dependencies between their social and business elements. Afterward, we present the application and infrastructure architectures. Finally, we discuss how the domain model facilitates tailoring the architecture by using custom business decisions and features. For enhanced readability, not all relations are shown in Figure 8.5.

Modeling of Organizational and Business Aspects

At the top of Figure 8.5, the `Social and Business Architecture` is shown. It captures the architectural elements that are required to specify the social characteristics of the ecosystems as well as the platform providers' business decisions. The root element is `SoftwareEcosystem`. The domain model defines that a software ecosystem includes at least four elements that are a `community`, `HumanActor`, `BasicSoftwareElement`, and `BusinessStrategy`. While the choice of `BasicSoftwareElement` determines the types of software platform, store, and third-party developments, we consider it as a part of the business architecture because it is a fundamental strategic decision that is taken in the early design phases together with other business decisions. Accordingly, a `BasicSoftwareElement` can be a `SoftwarePlatform`, `Store`, and `Third-PartyDevelopment`.

While the `Community` possesses certain `SymbioticRelations`, it can consist of different `OnionLayers` that each contains further layers. The `KnowledgeSharing` featured used by the community. Additionally, a `Community` can be a FOSS community.

Using `BusinessStrategy`, business decisions can be defined, modified, or reused across the actors. `BusinessStrategy` shows which *BusinessDecisions* are applied by which `HumanActors` while interacting with `BasicSoftwareElements`. In particular, `BusinessActions` enable the specification of the variants of a `BusinessDecision` for different actors or to reuse a `BusinessDecision` for several actors.

A `HumanActor` can be a `User`, `ServiceProvider`, `Partner`, and `PlatformProvider`. A `PlatformProvider` has a certain `OrganizationalContext` that concerns the company size, market size, domain criticality of services, and commerciality. The provider follows a release model strategy like pull-oriented, push-oriented, or hybrid. `User` requires certain *Expertise* to use the services. A `ServiceProvider` can be characterized as `LoneWolf` ans `Inside` as well. An `Insider` is a platform provider's *Partner* or a *Supplier* of resources.

The domain model captures three significant groups of business decisions in software ecosystems, i.e., `Fee`, `QualityCheck`, and `OpennessPolicy`. Accordingly, `Fee` can be a `PlatformFee` to use the platform, `EntranceFee` to enter the ecosystem, or `ServiceFee` to use a `Third-PartyDevelopment`. A `ServiceFee` is defined by a `ServiceProvider` or `Partner`.

In addition, deciding on the platform openness is a critical decision-making task in software ecosystems that enables platform providers to specify the degree of openness versus protection [JBSL12, KDKB14, BHH15]. Using `OpennessPolicy` defined in the domain model, platform providers can specify which actors, and to which extent,

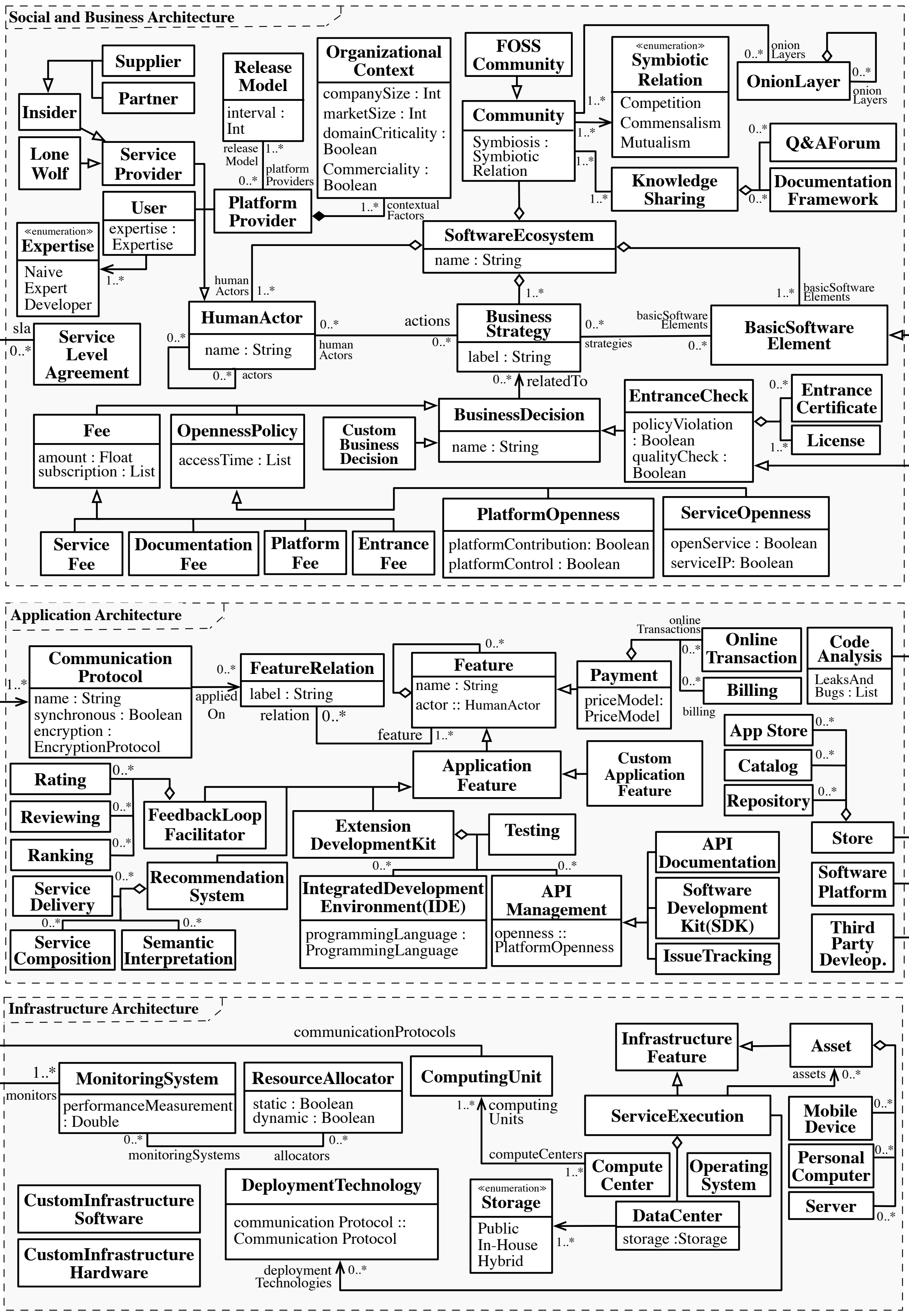


Fig. 8.5 SecoArc Domain Model for Software Ecosystems

can access or own a `SoftwarePlatform` or `Third-PartyDevelopment`. In particular, `PlatformOpenness` determines whether one can only contribute to the platform development or the equity ownership of the platform can be owned by the third-party providers. One way to regulate `PlatformOpenness` is where the third-party developers can develop new functions for the platform without having its ownership. An example of this situation can be seen in the ecosystem around the Mozilla Firefox web browser. Another `OpennessPolicy` is `ServiceOpenness` to grant a license to publish or to share intellectual property of services. For instance, in the Apple ecosystem, the developers receive a license to publish on the Apple App Store whereas, in the Cloud Foundry ecosystem, third-party services belong to the third-party providers and can be traded outside of the ecosystem.

Using `QualityCheck`, it can be determined whether `Third-PartyDevelopments` need to pass `StaticCodeAnalysis` before being published on `Store` or whether any of `HumanActors` should fulfill certain entrance requirements. Such requirements can be specified in models using `EntranceCertificate`.

Modeling of Application and Infrastructure Aspects

The application architecture supports business decisions by its application merits whereas the infrastructure architecture provides the application architecture with computing and deployment resources. Such resources are the software and hardware for the purpose of service execution. The domain model in Figure 8.5 shows the `Application-` and `Infrastructure Architecture` that respectively include `Application-` and `InfrastructureFeature`. In general, a `Feature` can be accessed by `HumanActor`, be a part of another `Feature`, or be in relation with another `Feature` in the same or another architecture.

Part of the domain model captures *built-in features*, i.e., the features specific to software ecosystems and open platforms. The built-in features originate from our study on variabilities of ecosystem architectures [JZEK17a, JPEK16a]. As depicted in Figure 8.5, the application-specific features are captured by means of `ExtensionDevelopmentKit` and `FeedbackLoopFacilitator`. `ExtensionDevelopmentKit` represents the software features that enable development on top of open platforms. This mainly includes `IDE`, `APIManagement`, and `Testing`. Android Studio is an example of `IDEs` used to develop Apps for Google Android. Android SDK is the `APIManagement` that facilitates accessing the platform APIs. Android Emulator is a `Testing` feature that is used to simulate a variety of hardware for testing purposes. Moreover, `FeedbackLoopFacilitator` makes user feedback operational in the ecosystem using `Rating`, `Reviewing`, and `Ranking`.

The feedback returned to the ecosystem through `Rating` and `Reviewing` is used to generate ranking lists and to improve services by service providers.

Furthermore, infrastructure-specific built-in features are grouped as `DeploymentT-echnology`, `ServiceExecution`, and `Asset` (cf. Figure 8.5). As the names suggest, they represent computing resources to deploy `Services` on a `DeploymentTechnology`, to execute them using `ComputingUnits`, and to deliver the execution results to an *Asset*. A `ComputingUnit` has an `OperationalEnvironment` and supports at least one `CommunicationProtocol`.

Modeling of Custom Business Decisions and Features

By introducing the domain model, we aimed for facilitating the modeling of ecosystem-specific concepts in models. However, this can be a limitation while specifying an architecture that includes elements, which are not included in the domain model. Thus, we introduce further language constructs that can facilitate tailoring the architecture. These constructs are `CustomBusinessDecision`, `CustomApplicationFeature`, `CustomInfrastructureSoftware`, and `CustomInfrastructureHardware`. These constructs can be instantiated in the models of ecosystem architecture to introduce custom business decisions or software/infrastructure features. The `CustomInfrastructureSof-tware` and `CustomInfrastructureHardware` can respectively be used to model features that provide hardware and software computing resources.

8.5.2 Visual Notation to Design Ecosystem Architecture

We develop visual notation to ease the application of the domain concepts in models. This helps address the efficiency and learnability of the SecoArc framework. Note that the domain concepts can be modeled by means of languages such as UML and ArchiMate as well. Such languages mostly need to become suitable for software ecosystems by using extension mechanisms such as stereotypes. For instance, we modeled several examples of ecosystem architectures using UML Component Diagrams in Section 7.3.

In the rest of this section, we present the SecoArc visual notation by means of an example. The example is base on the scenario of the Citrix ecosystem introduced in Section 8.2. Figure 8.6 shows how a business decision related to applying a fee in the Citrix ecosystem architecture is modeled. For readability purposes, only parts of the models are shown. A complete list of the visual notation can be found in the SecoArc specification [Jaz21]. At the bottom of Figure 8.6, a part of the Citrix ecosystem is shown that has been modeled using the SecoArc visual notation. The model demonstrates that `Enterprise 1` uses a service namely `Microsoft Skype for Business Server 2015`, which is developed and published on `Cirtix Ready Marketplace` by `Microsoft`. `Cirtix Ready Marketplace` uses `service discovery` such as search functions and categories. In addition, the store uses `access control`. Using the access control, partners can login to the store. To use the service, the `Enterprise 1` needs to pay a `service fee` that is modeled in the architecture using the SecoArc visual notation, i.e., S€. The service fees can be paid by performing the `online transactions` in the store.

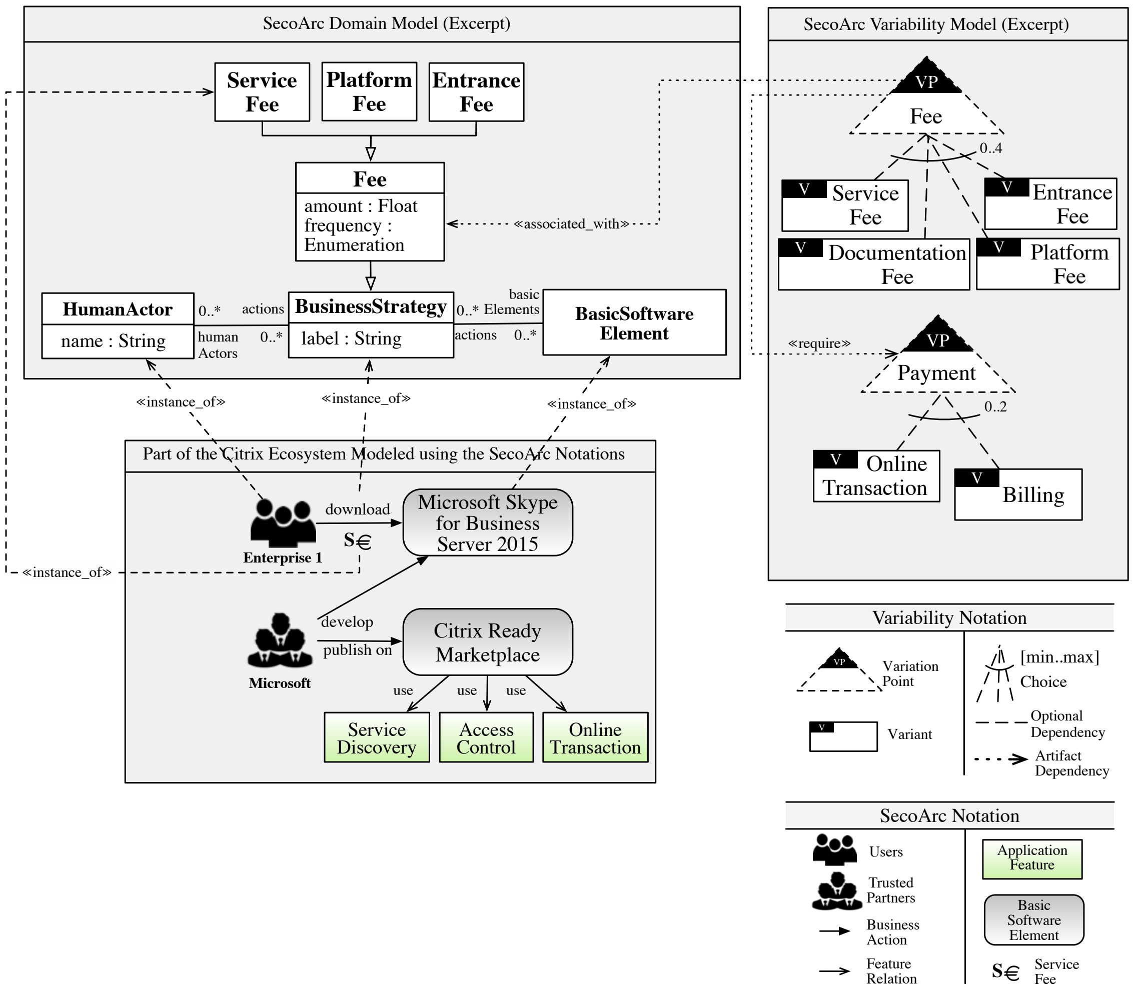


Fig. 8.6 Business Decision Service Fee Realized in the Architecture (For improved readability, models are partially shown.)

In addition, Figure 8.6 depicts the relations between the architecture, the SecoArc domain model, and the SecoArc variability model to show how the interrelations between the thesis' concepts. The Model of the Citrix ecosystem is on the M1 layer as discussed in Section 8.1. It is an instance of the SecoArc domain model that belongs to M2. The SecoArc domain model relates to the SecoArc variability model. The figure indicates that, through these relations, a new constraint emerges. The constraint implies *having fees requires payment functions to be realized in the architecture.* Specifically, this constraint becomes relevant since a *service fee* is realized in the Citrix architecture. According to the knowledge of variabilities discussed in Chapter 6, `fee` is a typical variation point in software ecosystems. The SecoArc domain model indicates this variability using inheritance relationships between `fee` and its three variants, i.e., `service fee`, `platform fee`, and `entrance fee`. In the variability model, the `require` relation between the variation points specifies a dependency. We specified such dependencies in the SecoArc tool support by using Object Constraint Language (OCL) and Xtend programming language. For more readability, the corresponding constraint is not shown in the figure, which specifies *if there is a kind of fee, then, there must be a payment feature in the architecture.* Since the Citrix ecosystem realizes `payment` by using an `online transaction` feature, the constraint is fulfilled by the architecture.

8.6 Architectural Analysis Technique

In this section, we present the SecoArc architectural analysis technique. The goal of the analysis technique is help platform providers a) assess the suitability of an ecosystem architecture with respect to the quality attributes of ecosystem health, b) compare alternative architectures, and c) use the practice-proven knowledge of existing ecosystems through the architectural decision-making.

The architectural analysis in the SecoArc framework relies on the pattern matching approach discussed in Section 8.3. The pattern matching consists of the context matching and decision matching. During the analysis, four types of outcomes are generated that are based on the results of the context matching and decision matching. We refer to each of these types as perspective because each of them reveals different insights into the ecosystem architecture.

Figure 8.7 illustrates the process of applying the architectural analysis that begins with analyzing the architecture using the pattern matching technique. The results of analysis are presented to the platform provider in terms of *pattern suggestion*

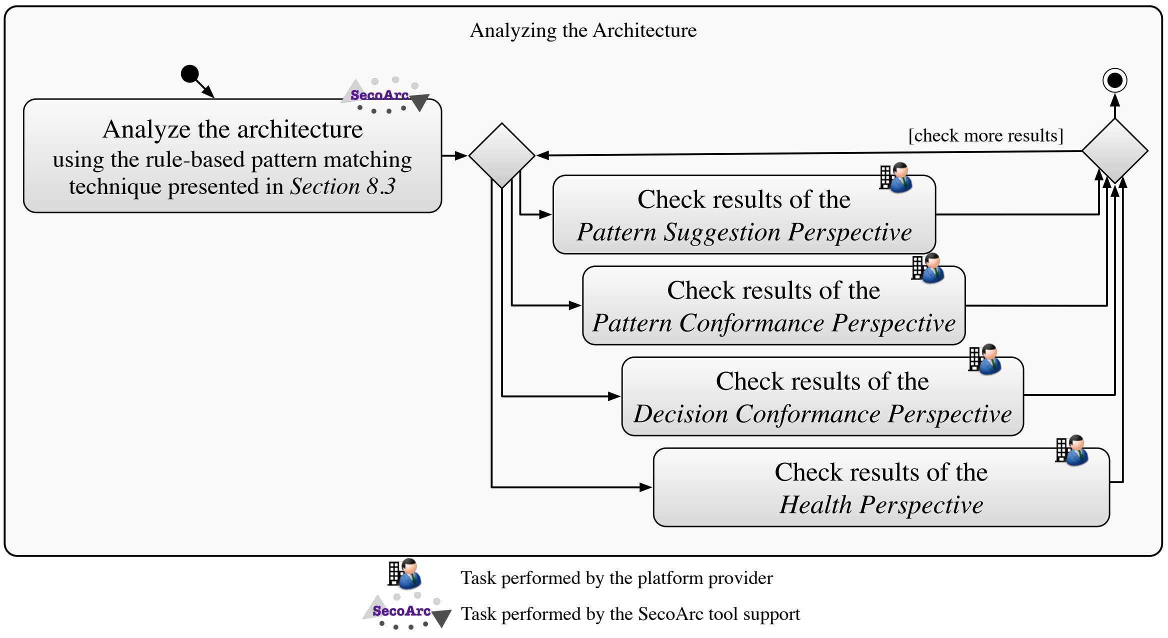

Analyzing the Architecture
SecoArc
Analyze the architecture using the rule-based pattern matching technique presented in *Section 8.3*
[check more results]
Check results of the *Pattern Suggestion Perspective*
Check results of the *Pattern Conformance Perspective*
Check results of the *Decision Conformance Perspective*
Check results of the *Health Perspective*
Task performed by the platform provider
SecoArc
Task performed by the SecoArc tool support

Fig. 8.7 Architectural Analysis Process

perspective, *pattern conformance perspective*, *decision conformance perspective*, and *health perspective*. We implemented the architectural analysis technique in the SecoArc tool support. The function of pattern matching is performed by the SecoArc tool support while the results of the architectural perspectives are checked by the platform provider. Platform providers should decide on relevant perspectives depending on their needs. In the following, we explain the four perspectives in detail.

8.6.1 Architectural Perspectives

In this section, we present different kinds of outcomes that architectural analysis performed using the SecoArc architectural analysis technique have. As depicted in Figure 8.8, the outcomes of the analysis are grouped into four categories that we refer to them as *architectural perspectives*. The reason is that each outcome gives different information concerning the architecture. *Pattern suggestion perspective* recommends a pattern that mostly matches the platform provider's contextual factors whereas *pattern conformance perspective* reveals the pattern to which the architecture mostly conforms. *Decision conformance perspective* shows the ecosystem-specific decisions that are currently realized / not realized in the architecture. Finally, *health perspective* shares information on how the architecture addresses the quality attributes of ecosystem health.

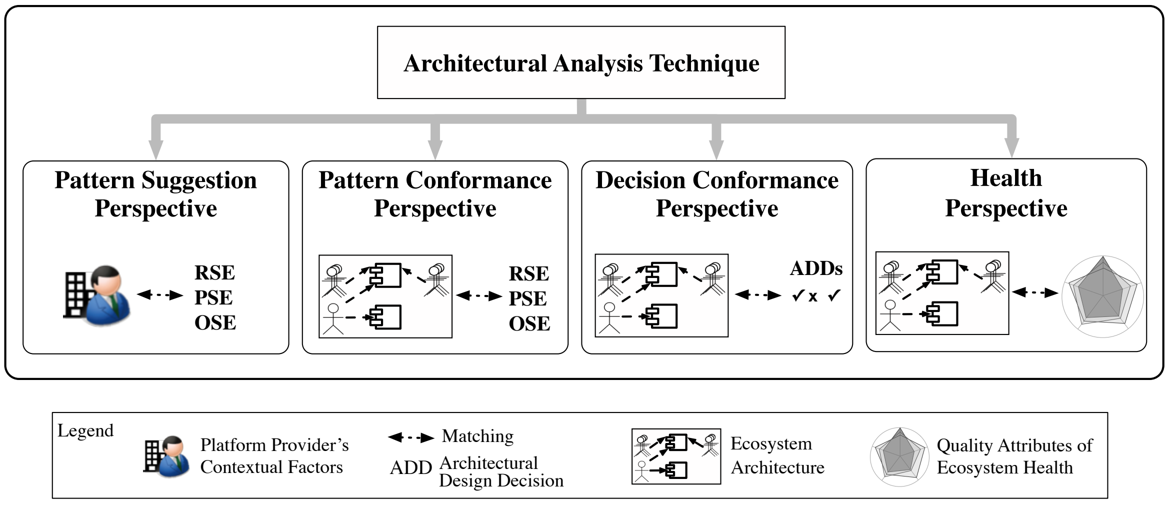

Architectural Analysis Technique
Pattern Suggestion Perspective
RSE
PSE
OSE
Pattern Conformance Perspective
RSE
PSE
OSE
Decision Conformance Perspective
ADDs
✓x ✓
Health Perspective
Legend
Platform Provider's Contextual Factors
Matching
ADD Architectural Design Decision
Ecosystem Architecture
Quality Attributes of Ecosystem Health

Fig. 8.8 Outcomes of SecoArc Architectural Analysis Technique

The architectural perspectives are the results of the context-matching and decision-matching presented in Section 8.3. Details related to the implementation of the architectural perspectives can be found in Section 8.7.

Pattern Suggestion Perspective

The pattern suggestion perspective is the result of the context matching. According to the value of contextual factors, the analysis suggests one or more patterns to be applied. The pattern suggestion is based on the knowledge of the existing ecosystems. It implies that existing platform providers with similar organizational contexts have frequently created similar ecosystem architectures, which we have extracted in terms of the three patterns.

Furthermore, the pattern suggestion perspective provides the platform provider with a list of existing ecosystems so that real-world examples of the suggested pattern can be found and further examined by the platform provider. In addition, the percentage of matching with the rest of the patterns is given to help the platform provider better understand the relations between the ecosystem architecture and the contextual factors and the rationale behind the pattern suggestion.

Pattern Conformance Perspective

The pattern conformance perspective shows to which extent the ecosystem architecture matches with each of the patterns. The extent is given in percentage. It implies the number of decisions matched as a result of performing the decision-matching. Additionally, similar to the pattern suggestion perspective, exemplary ecosystems are given to assist platform providers to follow the idea of each pattern by referring to the real-world examples. As mentioned earlier, the patterns are associated with a high-level business objectives, i.e., *business scalability*, *strategic growth*, and *innovation*. Using the knowledge of the pattern conformance perspective, platform providers can orient themselves through the process of decision-making for finding the right architecture for their high-level strategy.

Decision Conformance Perspective

The decision conformance perspective reveals the ecosystem-specific design decisions, which are defined as part of the patterns, are realized by the architecture. In particular, it is a detailed view of the results of decision-matching that includes the list of the architectural design decisions of each pattern and whether they have been realized in the architecture or not.

In addition, a *report of analysis* shows a complete view of the design decisions and consequences of current architectural decision-making. This information is based on the knowledge that we collected during the development of the architectural commonalities, variabilities, and patterns from the existing ecosystems and relevant literature in the previous chapters of the thesis.

Health Perspective

The health perspective shows the extent that the architecture addresses the quality attributes of ecosystem health, i.e., *creativity, profitability, sustainability*, and *inter-operability*. This information is generated by the decision-matching based on the fact that each set of decisions in the patterns are associated with a quality attribute. The knowledge of health facilitates the possibility to gear the architecture to desired quality attributes by opting for certain design decisions, which support those attributes.

In addition to the perspectives described above, the architectural analysis can be used to create a comparative view for more than one architecture. In the comparative view, results of the pattern conformance, decision conformance, and health perspectives for more than one architecture are provided.

Example of the Citrix Case

In Section 8.2, we introduced the exemplary scenario of Citrix and the goal to enhance

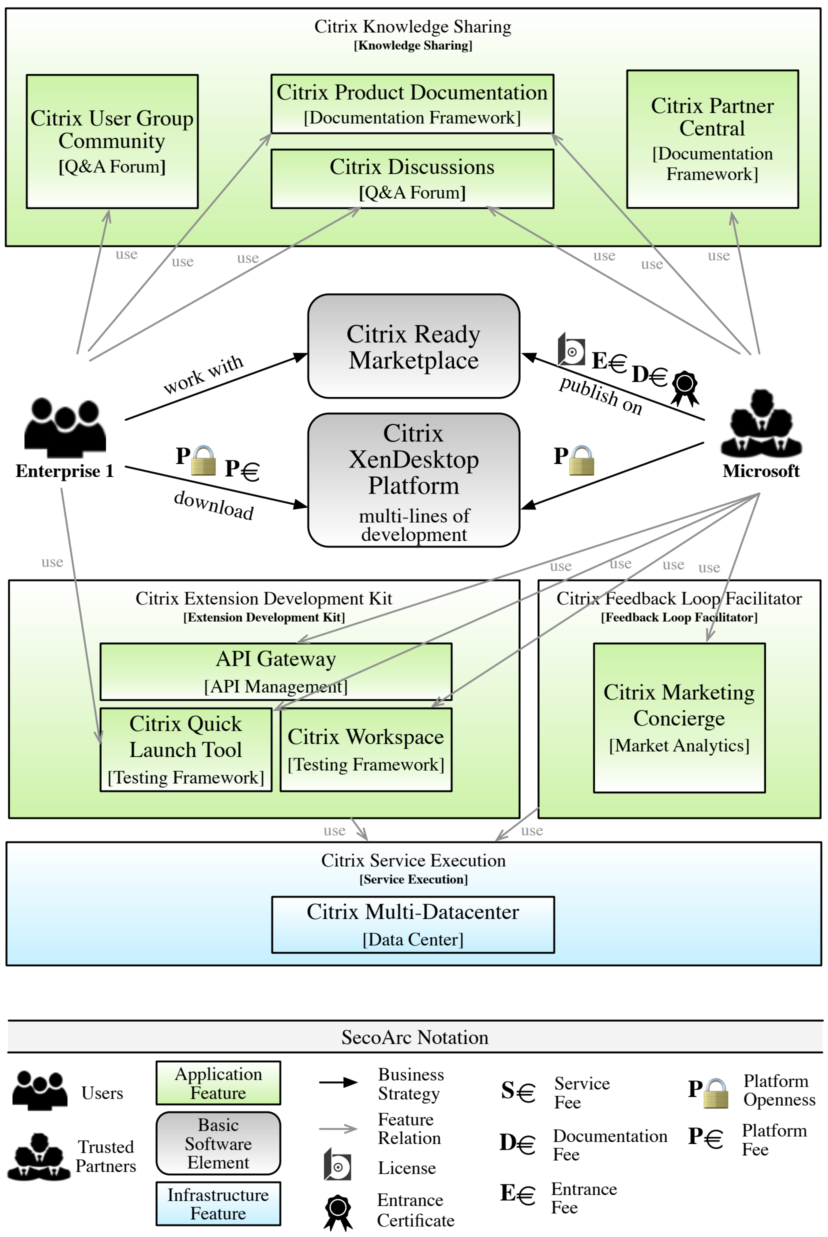


Fig. 8.9 Citrix Ecosystem Architecture
(For enhanced readability, only parts relevant to architectural analysis are shown and feature relations are depicted in gray.)

business scalability. In Section 8.4, we followed the SecoArc design process by applying the RSE pattern in order to include architectural design decisions aligned with the mentioned goal. In this section, we consider improving the architecture towards the goal by analyzing the architecture using the SecoArc architectural analysis technique. For this purpose, we model the ecosystem manually.

Figure 8.9 shows the Citrix ecosystem modeled using the SecoArc visual notation. This model is based on the architectural design decision that we identified earlier in this chapter (cf. Table 8.2). In the middle, two main actors, i.e., Enterprise 1 and Microsoft interact with the Citrix store and platform. The business decisions such as platform openness are specified on the business strategies between the actors and the basic software elements. The platform is closed source and Enterprise 1 has to pay a platform fee while using it. In a separate window in the SecoArc tool support called *properties view*, the attributes of a platform fee can be specified, e.g., amount of fee and the type of payment (this is not shown in Figure 8.9). Further business decisions concern Microsoft and the access to Citrix Ready Marketplace. For instance, to publish a service, Microsoft should become a partner and obtain an entrance certificate from Citrix, which is subject to fulfilling certain portfolio requirements such as publishing a certain number of services per year.

On the top of Figure 8.9, Citrix knowledge sharing features are modeled. Citrix Product Documentation and Citrix Discussion are used by Enterprise 1 and Microsoft while each of these actors has access to another knowledge sharing feature.

At the bottom of Figure 8.9, further application and infrastructure-related features are shown. As mentioned in Section 8.4, Microsoft obtains marketing data collected by Citrix using Marketing Concierge. As a partner, Microsoft can use features of the Citrix extension development kit during software development. Enterprise 1 can use Citrix Quick Lunch Tool for testing and troubleshooting purposes while using the Citrix platforms. Citrix Multi-Datacenter is responsible to provide the data required during computing tasks.

After modeling the architecture, we use the architectural analysis technique. In the following, we discuss the resulting architectural perspectives. In the pattern suggestion perspective, PSE and RSE patterns are recommended, which is based on the information that Citrix is a commercial and large software company with a large market of third-party developments. Figure 8.10a shows the results of the pattern conformance perspective. It shows that the Citrix ecosystem matches 71.7% with the PSE pattern and 44.4% with the RSE pattern. The degree of conformance to the PSE pattern shows that the platform provider already established a commercial and closed ecosystem that is aligned with the results of the pattern suggestion perspective. In addition, exemplary ecosystems are given. An example of RSE and PSE matryoshka ecosystem is AWS. In the decision conformance perspective

Furthermore, Figure 8.10b demonstrates the results of the decision conformance perspective. Rating, reviewing, ranking, issue tracking, and IDE can still be integrated into the architecture in order to address business scalability. The architectural implications resulting from the decision conformance perspective are essentially similar to the results of applying the RSE pattern discussed in Section 8.4.

The result of the health perspective is the extent to which the Citrix ecosystem conforms to the design decisions associated with the quality attributes in Figure 8.10. Note that for readability purposes, only relevant parts of the architectural analysis are shown in Figure 8.10. The architectural analysis provides matching results with respect to the OSE pattern as well. Due to being out of the scope of our illustrative example, the figure excludes these parts.

	Partner-Based Software Ecosystem (PSE)	Resale Software Ecosystem (RSE)
Citrix Ecosystem	71.4%	44.4%
Exemplary Ecosystems	Vsphere VMware ExtendSim	Esri Facebook Google Android
Exemplary RSE and PSE Matryoshka Ecosystem	Amazon Web Services (AWS)	

(a) SecoArc Pattern Conformance Perspective

	Partner-based Ecosystem: Strategic Growth							Resale Software Ecosystem: Business Scalability								
	Profitability							Sustainability						Interoperability		
	Platform Fee	Entrance Fee	Monetized Documentation	Monetized APIs	Commercial Licensing	BYOL	Closed Source Service	Rating	Reviewing	Ranking	Testing Framework	Issue Tracking (Bug Tracking)	Multi-development Lines	Documentation Framework	IDE	Service Execution
Citrix Ecosystem	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓	✓	✗	✓

Architectural Design Decision Realized (✓) / Not-realized (✗)

(b) SecoArc Decision Conformance Perspective

Fig. 8.10 Design and Analysis of the Citrix Ecosystem

8.7 Tool Support

Platform providers can design models of software ecosystems by using the SecoArc design process presented in Section 8.4 and the SecoArc tool support. Table 8.11 outlines the thesis's concepts that we have implemented in the tool. The tool is a prototypical implementation of the SecoArc modeling framework. The SecoArc modeling language, which is based on the knowledge of architectural commonalities and variabilities, has been implemented in the tool. However, our research results that exclusively concern the social aspect have not been implemented. The concepts shared between the social and other aspects (e.g., *reputation system* and *knowledge sharing* in Figure 5.2) have been implemented. The reason is that we have added the social aspect to the thesis's problem statement in the final phase of this PhD. Another reason has been different tooling requirements for modeling the social aspect than the other architectural aspects. Modeling the social aspects raises a need for simulation techniques that is out of the scope of this thesis. Furthermore, the SecoArc design process is not a part of the tool. The rationale behind this decision was to observe users of the framework while following the SecoArc design process on their own and independent of any assistance from the tool. Furthermore, the SecoArc architectural analysis, including the rule-based matching technique and architectural perspectives, has been implemented in the tool.

Fig. 8.11 Thesis's Concepts Implemented in SecoArc Tool Support

SecoArc Modeling Framework	**SecoArc Tool Support**
SecoArc Modeling Language (Based on the knowledge of the architectural commonalities and variabilities)	Implemented in the tool except for the elements concerning *the social aspect (cf. Figures 6.2 and 5.2)*
SecoArc Design Process (Based on the knowledge of the architectural patterns)	Not implemented in the tool
SecoArc Architectural Analysis Technique (Based on the knowledge of the architectural patterns)	Implemented in the tool

We developed the SecoArc tool support by using the Eclipse Modeling Framework (EMF) and the Sirius Framework. Further details can be found in [Jaz21]. The rationale behind this decision was to observe the user of the framework while following the design process independently and without the tool assistance.

Figure 8.12 demonstrates an overview of the architecture of the tool support using the notation of UML Component Diagrams. `Platform provider` is the human actor, who interacts with the tool using a `modeling workbench` to design and analyze the ecosystem architecture. The `modeling workbench` uses the components of the `SecoArc modeling language` and *SecoArc architectural analysis technique* to help the platform provider handle modeling and analysis tasks. The `SecoArc modeling language` includes a `metamodel` that represents the SecoArc abstract syntax. The metamodel holds the domain model of software ecosystems. The concrete syntax is a set of `visual notation` for modeling the ecosystem architecture. Furthermore, further semantics is expressed by a set of `constraints`. The *SecoArc architectural analysis technique* comprises three main components: `Context matching` provides design recommendations based on the platform provider's organizational context. `Decision matching` analyzes the suitability of a single architecture whereas `architecture comparator` compares more than one architecture competing with each other. Finally, a `report of analysis` is generated and provided to the `platform provider`. The rest of this section elaborates on each component.

In the following, we present the main components of the tool support.

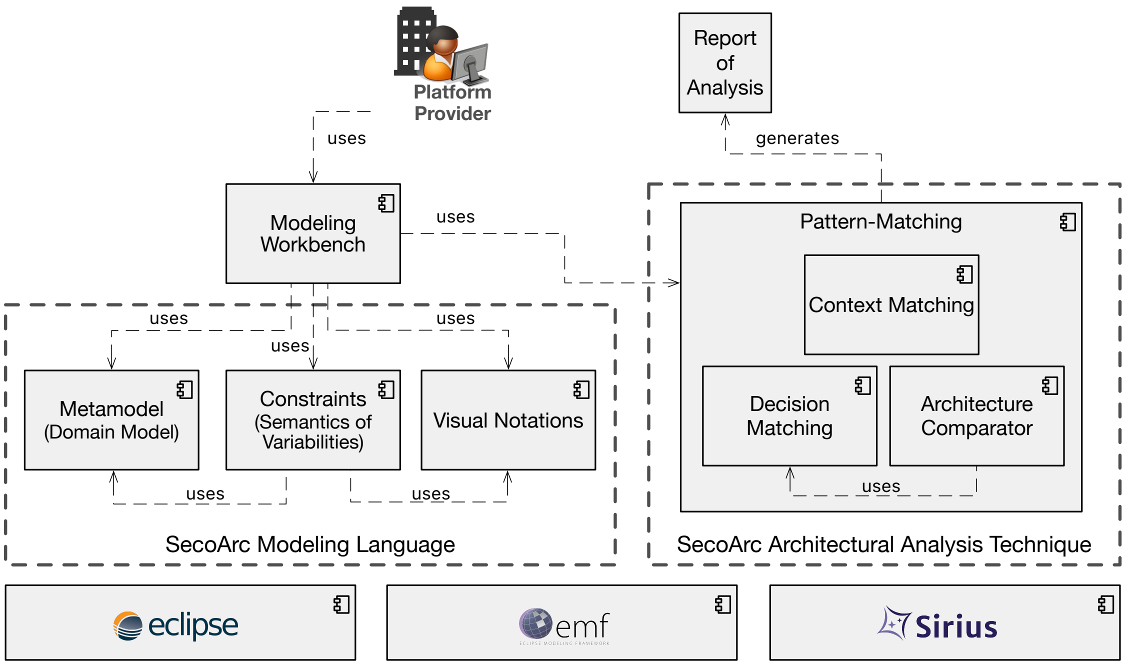


Fig. 8.12 Architecture of SecoArc Tool Support

Modeling Workbench

The *modeling workbench* is an integrated environment for modeling and analyzing architecture of software ecosystems. It provides access to the visual notation for creating models of ecosystem architecture. Afterward, the models can be analyzed by using the architectural analysis technique in the modeling workbench.

Metamodel

The abstract syntax of the modeling language is presented by a metamodel that comprises the domain knowledge of software ecosystems, which is captured by the domain model in Section 8.5. As already mentioned, the domain model includes the description of human actors and their relations. Besides, the architectural variabilities are captured in terms of four types of elements, i.e., the elements of organizational, business, application, and infrastructure architectures. The metamodel contains the semantics that is related to the well-formedness rules. An example for the well-formedness rules is *the ecosystem architecture should have at least one platform provider*.

Visual Notation

SecoArc introduces visual notation that form the concrete syntax of the modeling language. The notation can be used to design ecosystem architecture in the modeling workbench. While both visual and textual notation have their own advantages and disadvantages, the reason to choose visual notation for the SecoArc concrete syntax refers to the users of SecoArc that are platform providers, who are often in management positions. They would like to capture the main ideas efficiently without being confronted by overwhelming details. With this respect, visual notation surpasses textual representations in hiding unwanted complexity [WB15].

Constraints

Additional semantics of the language is defined by defining rules that are statically checked on the models. In SecoArc, these rules are implemented as OCL constraints. The constraints pertain to the structure of the ecosystem and should be obeyed by the architecture. We derive a major part of these constraints from the variability constraint dependencies that we identified defined in Section 6.1.3. For instance, *if the choice of service execution is defined as "remote execution on the cloud", then there must be infrastructure elements in the architecture that support this task or suppliers, who provide the execution resources.*

During architectural analysis, the pattern matching is performed that comprises the context-matching and decision-matching as mentioned in Section 8.3. In the following, we refer to the components of the architectural analysis technique shown in Figure 8.12 and explain how they handle the pattern matching tasks.

Context Matching

Each pattern is associated with a specific type of platform provider with certain organizational characteristics using four contextual factors, i.e., company size, market size, domain criticality, and commerciality. The `context matching` compares contextual parameters provided by the platform provider with the contextual factors of the patterns. This results in a *pattern suggestion* that shows which pattern would suit the platform provider's organizational characteristics the most. The platform provider's contextual factors can be defined in the properties view when the actor with the role of platform provider is selected.

Decision Matching

Each pattern is described using a concrete set of architectural design decisions. The `decision matching` check whether the decisions associated with the patterns are realized in the model of ecosystem architecture. The results of this matching are categorized into three perspectives as presented in Section 8.6.1, i.e., *pattern conformance*, *decision conformance*, and *health*. Here, according to the number of matched decisions, percentages of decision conformance and thereby the percentages of pattern conformance are calculated.

Architecture Comparator

The `architecture comparator` uses the results of `decision matching` to generate a comparative view for analysis of more than one architecture. In the comparative view, results of architectural analysis for more than one architecture in four architectural perspectives are shown. The architectural perspectives are pattern suggestion, pattern conformance, decision conformance, and health as described in Section 8.6.1.

In summary, we implemented the SecoArc framework in a tool to facilitate the application of the thesis's concepts that we have developed in the previous chapters. Using the components of the tool and a user-manual that we developed for this purpose, the SecoArc framework tool support aims at helping the platform providers to design software ecosystems and to increase the uptake of the concepts of the thesis.

8.8 Summary and Scientific Contributions

In previous chapters, we have developed the conceptual parts of the thesis's solution concepts, which include the domain knowledge of software ecosystems in terms of the architectural commonalities, variabilities, and patterns. In this chapter, we presented a modeling framework as well as a tool support on the basis of the previously developed concepts. The modeling framework comprises a *design process*, *modeling language*, and *architectural analysis technique*, to facilitate applying the domain knowledge in models.

The modeling framework is aimed at enhancing architectural decision-making by raising awareness about the strategic directions that the design decisions in the architecture are facilitating and by enabling platform providers to appraise their decisions on the basis of existing ecosystems so that they can accordingly orient themselves through the process of decision-making. In summary, the scientific contributions of this chapter are as follows:

- By using the design process, the platform providers are guided through the process of decision-making.
- Using the modeling language, designing the ecosystem architectures is facilitated. Unlike related work, the modeling language is not solely for modeling purposes, but it is the basis for architectural analysis.
- The architectural analysis technique enables the assessment of ecosystem architecture with respect to quality attributes of ecosystem health.
- By means of the analysis technique, our approach is the first to provide platform providers with concrete guidelines based on the knowledge of existing ecosystems.
- Our work provides an integrated tool support for modeling and analyzing software ecosystems.

Chapter 9

Validation

In the previous chapters, we developed the SecoArc architecture framework to facilitate designing and analyzing software ecosystems. In this chapter, we validate the framework with respect to our central validation question (RQ9 introduced in Chapter 4.2), i.e., *does SecoArc improve architectural decision-making to design software ecosystems?*

Our validation approach comprises two studies. After giving an overview of the validation approach in Section 9.1, we present a case study within the scope of on-the-fly computing [JSK$^+$20] in Section 9.2. In Section 9.3, we validate the framework in the context of existing ecosystems [JZEK17a]. We refer to threats to validity in Section 9.4, followed by a discussion and summary of the validation results in Section 9.5.

9.1 Validation Overview

The thesis's goal is *to improve architectural decision-making to design software ecosystems by providing architectural knowledge.* As discussed in Section 3.1, a solution should fulfill *functional suitability* [ISO12] regarding the goal. This means, we need to specify the degree to which the SecoArc framework provides functions to achieve the goal.

According to the design science methodology applied in this thesis [Wie14, Wie09], a validation aims at predicting how an artifact will interact with the problem context. Validation studies are experimental, where a prototypical implementation of the solution is assessed using a model of the problem context. The results of the validation are further used in the research to improve the artifact.

To specify functional suitability of the SecoArc framework, we assess the fulfillment of subcharacteristics of functional suitability considering the following criteria [ROP16]: To measure *functional completeness*, we consider *the number of requirements that have been developed in the framework* and *the ones that undergone the process of validation* comparing with *the number of requirements specified initially, i.e., R1-R9 specified in Section 3.1.* Moreover, to assess *functional correctness*, we consider *whether the results of applying the framework are satisfactory.* Functional appropriateness is concerned with whether the product fulfills usage objectives when it is used by the users. We validate the SecoArc framework with respect to two usage objectives that are *time and effort associated with using the framework.*

We validate the suitability of the SecoArc framework by performing two studies. In our first study, we validate *the knowledge base (i.e., the architectural commonalities and variabilities)* and *the methodical knowledge (i.e., the architectural patterns and modeling framework)* by means of a case study within the scope of on-the-fly computing. To validate whether the SecoArc framework improves architectural decision-making, an architect uses the SecoArc framework. The architect represents the platform provider's perspective and is responsible for architectural decision-making. Two ecosystems are designed, analyzed, and compared.

In another study, we validate the correctness of the variability model. Specifically, we consider whether existing ecosystems from different application domains can be described using the variability model. These are the ecosystems around the Salesforce, Apple, Amazon AWS, Eclipse, and Amazon Alexa platforms.

9.2 On-The-Fly Computing

On-the-fly computing is a paradigm for the automatic provision of individual IT services. It stems from a research project, which is developed within a Collaborative Research Center at Paderborn University. The IT services are composed of basic services that are provided by third-party providers in OTF markets. OTF markets are the place, where tasks related to the service life cycle are handled, e.g., publish of basic services, storage, search, composition, execution, and rating.

Given the complex architecture of OTF markets, a complex space of architectural variabilities is considerable. On-The-Fly Computing Proof-of-Concept (PoC) is a software platform for on-the-fly computing that is developed to realize one possible OTF market. It is developed by an architecture team called *PoC team.* We consider a scenario that the PoC team is going to open the platform to external service providers so that their third-party services will be composed on the basis of the PoC platform. The team is confronted with several variabilities that can result in different architectures. Subsequently, each architecture fulfills different requirements and quality attributes.

In this context, the variabilities are the design decisions that can be changed at the design time without violating core functional requirements.

Setting of the Case Study

The PoC platform with *60k+* lines of code is responsible to perform on-the-fly composition of software services. We performed the case study with eight representatives from the PoC team and two external service providers in January 2019. The PoC representatives include six domain experts responsible for conceptualizing single components like the chatbot, an architect responsible for designing and integrating the system components, and a developer. The domain experts have interdisciplinary backgrounds in business, economics, software engineering, networking, and infrastructure areas. Depending on their backgrounds, they have different perspectives regarding OTF markets, and thereby, holding different requirements.

In this case study, we first extracted the requirements of the PoC team. Then, the architect used the SecoArc framework to design, analyze, and compare two ecosystem architectures. Afterward, we conducted an interview with the architect to evaluate whether the framework was helpful to design the software ecosystems. This case study has been published in [JSK$^+$20].

9.2.1 Validation Questions

By performing a case study within the scope of on-the-fly computing, we pursue the goal of validating the SecoArc framework by involving the stakeholder relevant to the thesis's problem, i.e., a platform provider. For this purpose, we follow the goal-question-metric (GQM) approach [CR94]. Table 9.1 shows the goal of the thesis expressed using the GQM template. Based on the goal, we derive the validation questions (VQ1, VQ2) from the requirements (R1-R9 in Section 3.1). To judge the suitability of the domain knowledge provided by the SecoArc framework, we use metrics (M1-M7) that we define based on the criteria of functional suitability discussed in Section 9.1. In the following, we elaborate on the validation questions and the relation to the research questions (RQ1-RQ9) presented in Section 4.2.

Table 9.1 Goal Question Metric Applied in the Thesis

Questions	Goal / Metrics
	Goal
	Purpose: Improve
	Issue: architectural decision-making while designing
	Object: software ecosystems
	Viewpoint: from an ecosystem architect's viewpoint
Questions	**Metrics**
VQ1 Does the SecoArc knowledge base facilitate designing software ecosystems?	M01: Number of questions to functional suitability of the knowledge base answered positively by the architect (Architect's satisfaction) M02: Time required to model ecosystem architecture
VQ2 Does the SecoArc methodical knowledge facilitate designing software ecosystems?	M03: Number of process steps tested M04: Number of questions to functional suitability of the methodical knowledge answered positively by the architect (Architect's satisfaction) M05: Number of decisions with relation to the ecosystem health M06: Number of deficiencies detected and improvement suggestions M07: Time required to analyze the architecture

VQ1: Does the SecoArc knowledge base facilitate designing software ecosystems?

This validation question is directly concerned with the parts of the SecoArc framework that we developed in response to the thesis's main research question (RQ2). In addition, it refers to the further questions that resulted in developing the SecoArc knowledge base, i.e., RQ3 and RQ4 (cf. Figure 4.2). We answer this validation question by measuring the metrics M1 and M2 shown in Table 9.1. To measure the suitability of the SecoArc knowledge base (i.e., completeness, correctness, and appropriateness), we refer to the user opinion [MS94] that is the architect in this case. Furthermore, we validate the appropriateness of the knowledge base by measuring the time that is required to model ecosystems (M2).

VQ2: Does the SecoArc methodical knowledge facilitate designing software ecosystems?

The validation question generally refers to the answer given to RQ2 and it specifically casts doubt on the solutions for RQ5-RQ8. We answer this question by considering the metrics M3-M7 listed in Table 9.1. We measure the functional completeness of the SecoArc design process by referring to the number of the SecoArc process steps tested during the validation (M3). Similar to the knowledge base, we validate the functional suitability of the SecoArc methodical knowledge (i.e., completeness, correctness, and appropriateness) by referring to the architect's opinion (M4). This measurement is performed in the view of the previous validation results (M1, M3). By measuring the number of decisions relating to the ecosystem health and the number of deficiencies detected and improvement suggestions (M5, M6), we aim for assessing the suitability of the knowledge of the architectural patterns and architectural analysis technique. M7 is introduced to validate the efficiency of using the SecoArc methodical knowledge by means of the tool support.

9.2.2 Study 1: On-The-Fly Computing Proof-of-Concept

On-The-Fly Computing Proof-of-Concept (PoC)[1] is a market for provisioning machine learning (ML) services. The PoC platform was designed to perform on-the-fly composition of basic services. The basic services come from the ML libraries Weka[2], in Java, and Scikit-learn[3], in Python. With the help of the PoC, users are automatically provided with tailor-made ML services for typical classification problems without having any prior knowledge of ML.

An example scenario for the PoC functionality is when a user asks for a service for an automatic classification of animal pictures. There, the PoC learns a classification model that predicts if an unlabeled picture shows a dog or cat, e.g., based on labeled pictures of cats and dogs, Initially, in dialogue with a *chatbot*, the user submits her (non-)functional requirements and the training data. The chatbot broadcasts the requirements to *configurators*. Configurators are computer programs to find and compose basic services. They consider different combinations of basic services. Once suitable combinations are found, the results of the service compositions with different non-functional properties are offered to the user. Upon the user's acceptance, her personal service is composed and further deployed for execution in a *compute center.*

Elicitation of PoC Requirements

To open up the PoC platform to third-party providers, first, the ecosystem around the platform needed to be designed. Therefore, the first step of our validation was to elicit the requirements of the PoC ecosystem. These include architectural variabilities and relevant quality attributes that should be considered during the design.

[1]`sfb901.uni-paderborn.de/poc`, Last Access: March 20, January 10, 2022.
[2]`cs.waikato.ac.nz/ml/weka`, Last Access: January 10, 2022.
[3]`scikit-learn.org`, Last Access: January 10, 2022.

We conducted a *quality attribute workshop* (QAW) [BEL^{+}03] with the PoC team and two external service providers, which lasted for two hours. A QAW is a requirements engineering technique based on architecture tradeoff analysis method (ATAM) to identify architectural drivers and quality attributes of a software-intensive system by triggering discussions between different stakeholders.

Initially, we introduced the quality attributes of ecosystem health mentioned in Section 2.2.3 to the PoC team. Then, we asked them to identify the attributes that are critical for the PoC system from their viewpoints. Afterward, we asked them to describe the critical quality attributes by means of *raw scenarios*. Raw scenarios do not necessarily conform to the prescribed format of QAW. But, they are exemplary scenarios for system behavior under certain circumstances.

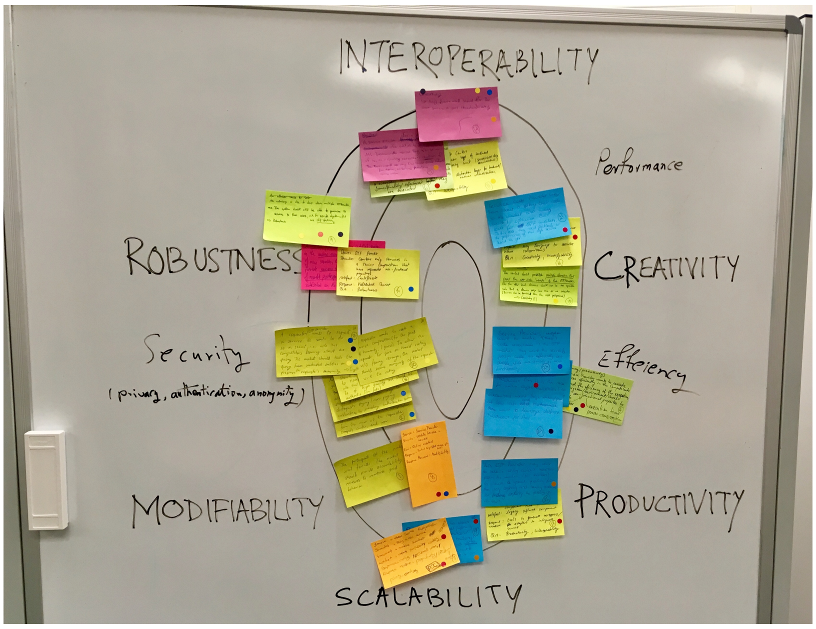


Fig. 9.1 Quality Attributes Identified in the Workshop

In the next stage, *formal quality attribute scenarios* were derived from the raw scenarios in four steps, i.e., generation, consolidation, prioritization, and refinement of detailed scenarios. In total, we collected 25 formal quality attribute scenarios for the PoC ecosystems. Throughout the workshop, the quality attributes and the scenarios relevant to them are organized on a spider web chart by different participants as shown in Figure 9.1. The participants are asked to vote on the scenarios. A scenario with more votes is shown to be more important to the PoC team. A list of the scenarios can be found in Appendix B.

In the next step, based on the scenarios, architectural variabilities, i.e., variation points and variants of the PoC ecosystem, are identified. Table 9.2 shows the variabilities in three groups of business, application, and infrastructure. These groups are based on the main architectural aspects of software ecosystems as introduced in Section 2.2. The variants are not mutually exclusive, i.e., several variants of the same variation point can be applied at the same time.

Table 9.2 Architectural Variabilities of the PoC Ecosystem

Business	**Entrance Quality Check (VP1)**	(**v1.1**) Entrance to the ecosystem is subject to no quality check. (**v1.2**) If necessary, using static code analysis, quality of third-party services should be checked. (**v1.3**) Third-party providers' qualification may need to be evaluated.
	Entrance Fee (VP2)	(**v2.1**) Entering the ecosystem is free. (**v2.2**) However, in some cases, the third-party providers need to pay an entrance fee.
	Service Fee (VP3)	The ecosystem supports (**v3.1**) free and (**v3.2**) paid third-party services.
	Access to Platform (VP4)	(**v4.1**) The platform's source code is open so far the platform quality is not threatened. (**v4.2**) The team might want to limit the access if necessary.
	Access to Third-party Services (VP5)	(**v5.1**) Basic services need to be open source. (**v5.2**) Protecting intellectual property of services becomes service providers' right.
Application	**Marketplace (VP6)**	(**v6.1**) While the default is to use available service repositories, (**v6.2**) the team is ready to establish own marketplace.
	Programming Language (VP7)	The ecosystem leverages two programming languages for third-party services, i.e., (**v7.1**) Java and (**v7.2**) Python.
Infrastructure	**Hardware Allocation (VP8)**	(**v8.1**) Compute centers dedicate fixed hardware resources to basic services. (**v8.2**) Compute centers dynamically allocate the resources at run-time based on resource availability and performance metrics.
	Distribution of Execution (VP9)	(**v9.1**) Execution of basic services is distributed on several compute centers. (**v9.2**) Each basic service is executed centralized by a compute center.

Variation Point (VP_i) / Variant ($v_{i,j}$)

While different ecosystem architectures can be built upon the variabilities, different architectures support different business objectives and quality attributes of ecosystem health. The PoC team wanted to decide on the most suitable architecture among the two alternative choices described below. The alternative ecosystem architectures can be matched to the main scenarios of software ecosystems that we identified in Section 6.2.2. Accordingly, the first architecture matches the descriptions of an open ecosystem while the second architecture concerns a semi-open controlled ecosystem. Therefore, we name the alternative architecture after the matched scenarios. Specifically, the PoC team is interested in considering the following two architectures:

Architecture #1: Open Ecosystem *The platform remains an independent open-source project so that the ecosystem grows by the direct contributions of service providers. All source codes should be free to use.*

Architecture #2: Semi-Open Controlled Ecosystem *If the number of service providers drastically increases, the ecosystem openness might cause degradation in the quality of services. In this case, the PoC team wants to prevent unleashed growth of the ecosystem by establishing a controlled software development and marketing environment.*

Despite different architectural variabilities, the team expressed that the following business vision should always be fulfilled:

Business Vision*: The ecosystem should support innovation while being sustainable in terms of confronting external threats that could have a long-term impact on the platform's success. Furthermore, the platform ownership should be managed by using the GNU General Public License (GPL).*

We provided the PoC architect with the results of the requirements engineering actions mentioned above that we have documented during the workshop. This includes the knowledge of the quality attributes, the scenarios, architectural variabilities, business vision, and the general definition of the two architectures.

Afterward, the architect designed and analyzed two ecosystems architectures using the SecoArc modeling workbench. As mentioned in Section 8.7, the modeling workbench is a part of the SecoArc framework tool support that can be used to design and analyze software ecosystems. The following sections are dedicated to details of the design and analysis of the PoC alternative architectures.

Organizational Context of the PoC

In this step, the organizational context of the PoC is described using the contextual factors. The factors remain the same for both *Architectures #1* and *#2*. Accordingly, the PoC platform is open, free-to-use, and not safety-critical. With <*100* employees and <*100* third-party services, the PoC is considered a small size organization with a medium-size market. These values are specified based on the ranges that we have introduced in 7.2.

Modeling Ecosystem Architecture #1

This section describes the process of modeling Architecture #1 based on the knowledge of the variabilities (cf. Table 9.2). To keep the ecosystem fully open, entering the ecosystem is free for the users and service providers (Table 9.2: **v2.1**). The third-party services are free (**v3.1**) and released under the GPL license (**v5.1**). To enable third-party providers' contribution to a full extent, the team will provide open code repositories (**v4.1**). The only entrance barrier for the service providers is to pass a static code analysis (**v1.2**).

Figure 9.2 shows the application and infrastructure features in relation to a part of the business architecture designed using the SecoArc notations. Currently, a microservice architecture is used, where the PoC `CustomApplicationFeatures`, i.e., the `Chatbot` and `Configurator`, are deployed on `Compute Center 1` and `Compute Center 2`.

To enable third-party contributions to a full extent, the team uses `Git` as an open code repository (**v4.1**) (**v6.1**). The default way to publish is to `commit` new services into the repository. Because the services are encapsulated inside the `DockerContainers`, `Providers of ML Services` are allowed to develop both `Basic Service in Java` and `Basic Service in Python` (**v7.1**) (**v7.2**). During the service provision, the execution of basic services is handled by several docker containers. Figure 9.2 portraits `DockerContainer 1` that consists of two `ComputingUnits`, called `Executor 1`, `Executor 2`. The executors `register` with the `Gateway` in advance (**v8.1**). The `Gateway` is a `ResourceAllocator` that allocates each basic service to multiple executors (**v9.1**).

Modeling Ecosystem Architecture #2

This section elaborates on the process of modeling Architecture #2 based on the knowledge of the variabilities (cf. Table 9.2). To tackle uncontrolled growth, several business decisions are to be taken into account. Figure 9.3 shows the business decisions designed using the SecoArc notations. Human actors, like `Providers of ML Services`, interact with the `BasicSoftwareElements` on the basis of `BusinessActions`. The symbols on the actions, e.g., $E_€$ and $S_€$, represent the business decisions that impact those actors.

Firstly, to create a controlled marketing environment, the team considers providing its own marketplace (**v6.2**). This way, the providers are forced to continuously improve their services in order to beat the increased market competition. In addition,

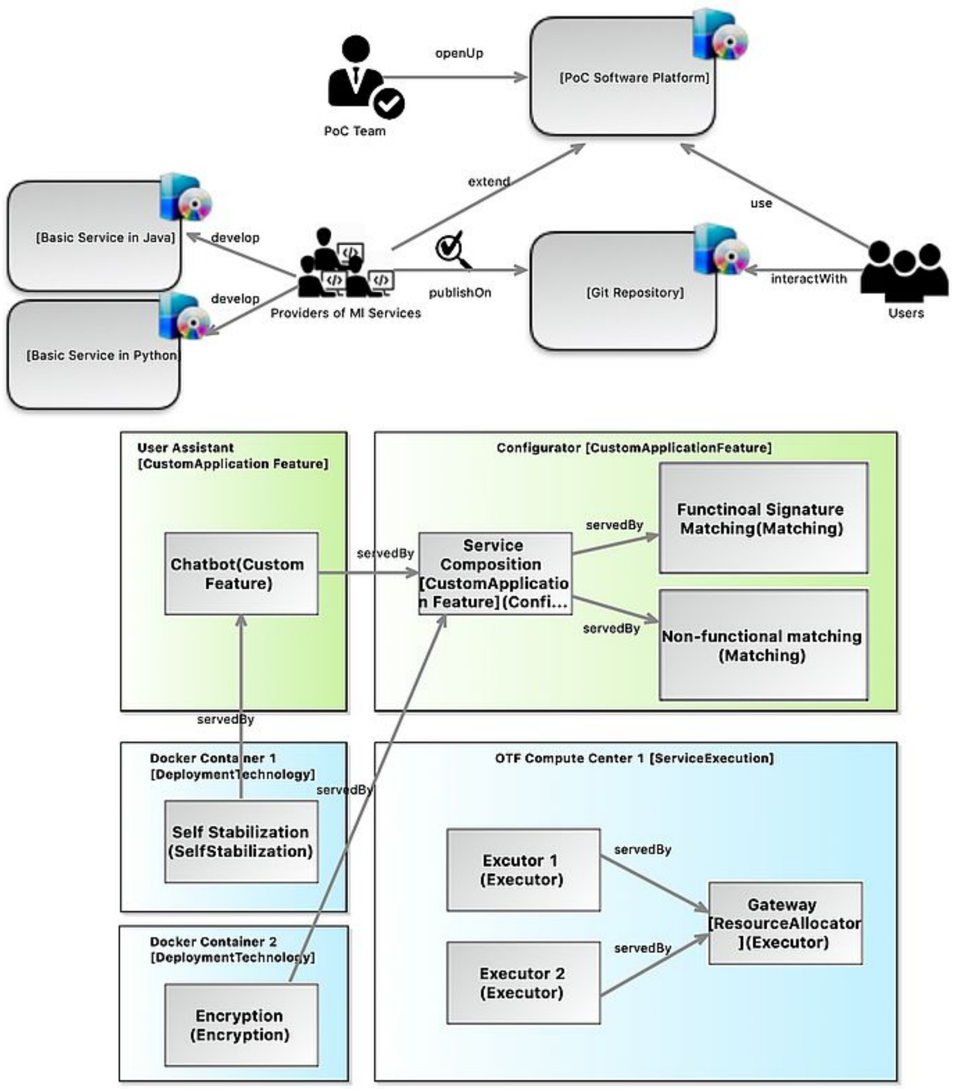

openUp
[PoC Software Platform]
PoC Team
extend
use
[Basic Service in Java]
develop
develop
Providers of MI Services
publishOn
[Git Repository]
interactWith
Users
[Basic Service in Python]
User Assistant [CustomApplication Feature]
Chatbot(Custom Feature)
servedBy
Configurator [CustomApplicationFeature]
Service Composition [CustomApplication Feature](Confi...
servedBy
Functinoal Signature Matching(Matching)
servedBy
Non-functional matching (Matching)
servedBy
Docker Container 1 [DeploymentTechnology]
servedBy
Self Stabilization (SelfStabilization)
OTF Compute Center 1 [ServiceExecution]
Excutor 1 (Executor)
servedBy
Gateway [ResourceAllocator](Executor)
servedBy
Executor 2 (Executor)
Docker Container 2 [DeploymentTechnology]
Encryption (Encryption)

Fig. 9.2 Architectures #1 modeled in the SecoArc Modeling Workbench (Screenshot) (More information on the SecoArc notation in Appendix D)

membership controls in terms of *1€* annual `EntranceFee` for the providers are to be set (**v2.2**) (shown by $E_€$ in Figure 9.3). Note that the notations in Figure 9.3 do not show attributes of the business decisions. However, the SecoArc modeling workbench has a *properties view* that allows to define them. According to the business vision, the PoC platform remains free and under the GPL license. Thus, `PlatformContribution` is `true` (**v4.1**).

Secondly, to ensure providing high quality services, the team considers partnering with other companies. In this case, `Partners` need to *"possess certain amount of annual revenue"*, which is specified using an `EntranceCertificate` in Figure 9.3. Moreover, the partners are allowed to monetize their services and the belonging documentation by defining price models (**v3.2**), closing the source code, or use their own licenses (BYOL) (**v5.2**). Thereby, two types of services are created for `Partners`, i.e., `Basic Service 2` and `3`, whereas `Basic Service 2` has `ServiceFee` and `ServiceOpenness` policies defined. The rest of business decisions remain the same as *Architecture #1*.

Upon deciding on an own marketplace by the PoC team, related software features of the marketplaces, e.g., rating, ranking, and reviewing, are included. To create a controlled development environment, the team considers providing the service providers with an IDE and testing environment. By a drastic increase in the number of service providers, some decisions in *Architecture #1* cause scalability issues: By static resource allocation (**v8.1**), the executors are permanently waiting for jobs, thus, wasting resources if they remain idle. Furthermore, by the distributed execution (**v9.1**), if many executors become incorporated in the execution of a service, the network payload will be tremendous. Therefore, the team decides that, for every basic service, a Docker container is generated that contains all the necessary environments as well as incorporated basic services (**v9.2**). A respective container is distributed in a compute center on demand and will be freed up after the execution (**v8.2**).

Architectural Analysis of Architecture #1 and Architecture #2

The PoC architect used the architectural analysis technique integrated in the SecoArc modeling workbench to evaluate suitability of *Architectures #1* and *#2*. This section presents the results of the architectural analysis. First, the results of the analysis for *Architecture #1* and *#2* are provided separately. In particular, the focus of these parts is the reports of analysis that are generated for *Architectures #1* and *#2* during the analysis. As mentioned in Section 8.6.1, a report of analysis is generated in the decision conformance perspective. It provides a detailed view of the design decisions

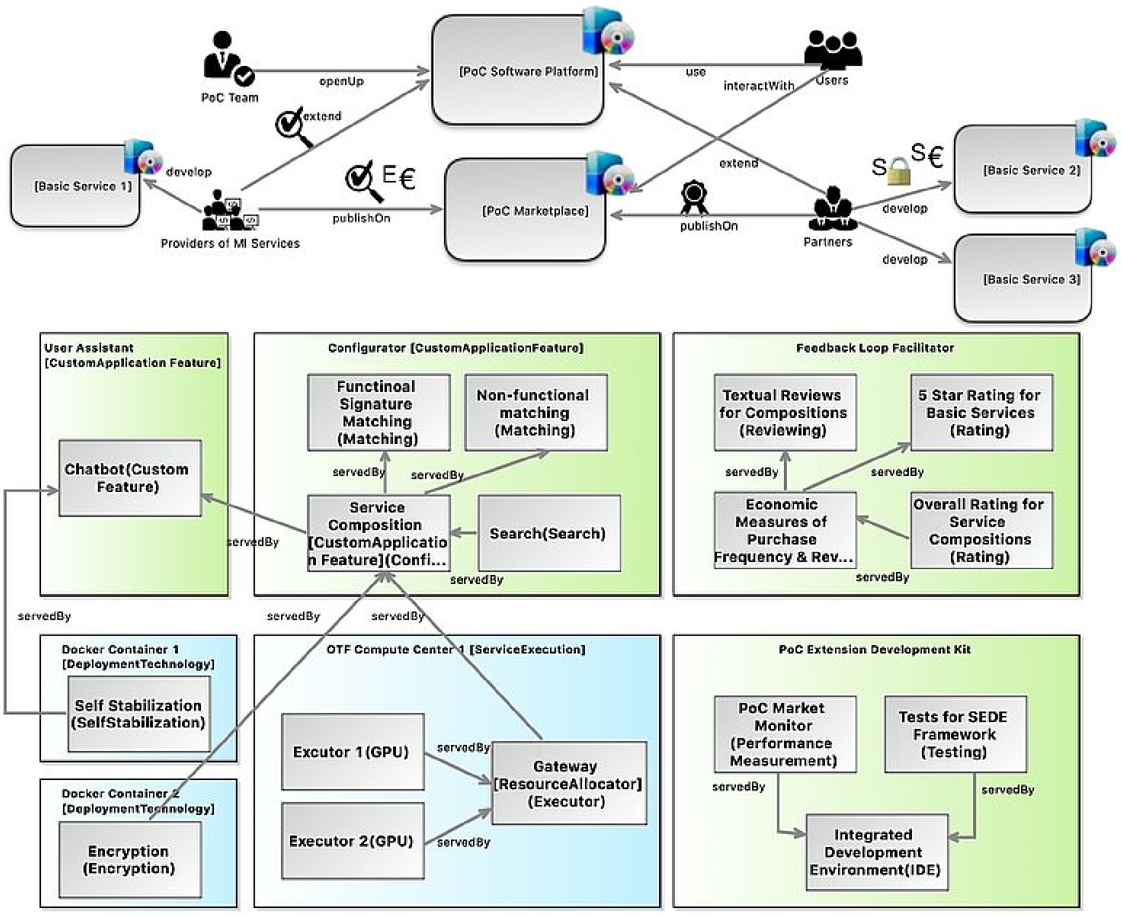
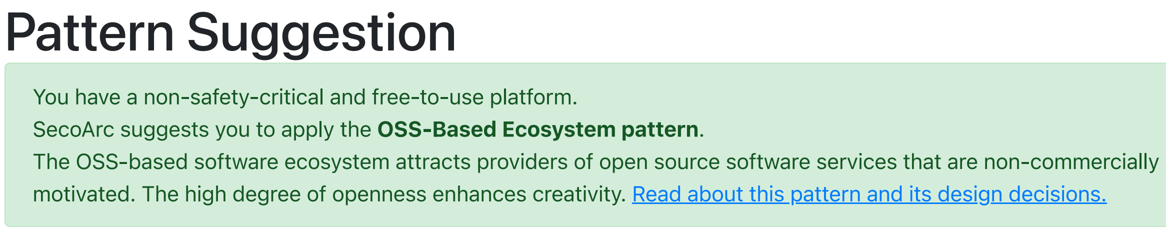

[PoC Software Platform]
PoC Team
openUp
extend
use
interactWith
Users
[Basic Service 1]
develop
Providers of MI Services
publishOn
[PoC Marketplace]
extend
publishOn
Partners
develop
develop
[Basic Service 2]
[Basic Service 3]
User Assistant [CustomApplication Feature]
Chatbot(Custom Feature)
Configurator [CustomApplicationFeature]
Functinoal Signature Matching (Matching)
Non-functional matching (Matching)
servedBy
servedBy
Service Composition [CustomApplication Feature](Confi...
Search(Search)
servedBy
servedBy
Feedback Loop Facilitator
Textual Reviews for Compositions (Reviewing)
5 Star Rating for Basic Services (Rating)
servedBy
servedBy
Economic Measures of Purchase Frequency & Rev...
Overall Rating for Service Compositions (Rating)
servedBy
servedBy
servedBy
servedBy
Docker Container 1 [DeploymentTechnology]
Self Stabilization (SelfStabilization)
Docker Container 2 [DeploymentTechnology]
Encryption (Encryption)
OTF Compute Center 1 [ServiceExecution]
Excutor 1(GPU)
Executor 2(GPU)
servedBy
servedBy
Gateway [ResourceAllocator] (Executor)
PoC Extension Development Kit
PoC Market Monitor (Performance Measurement)
Tests for SEDE Framework (Testing)
servedBy
servedBy
Integrated Development Environment(IDE)

Fig. 9.3 Architecture #2 modeled in the SecoArc Modeling Workbench (Screenshot) (More information on the SecoArc notation in Appendix D)

and consequences of current decision-making according to the domain knowledge of software ecosystems developed in this thesis.

Analysis of Architecture #1 Once the architect ran the architectural analysis for *Architecture #1*, a report of analysis was generated as shown in Figure 9.5. The decisions in green are the ones that are realized in the architecture. On contrary, the decisions in red are not realized. Furthermore, a brief description is given for each decision that addresses the advantages of a decision being realized or the disadvantages if the decision is not realized.

It can be easily recognized that *Architecture #1* matches to the OSS-based pattern the most. This is not a surprise as the business vision of *Architecture #1* is the key driver for the architectural design decisions that promote ecosystem openness. For


Pattern Suggestion

You have a non-safety-critical and free-to-use platform.
SecoArc suggests you to apply the **OSS-Based Ecosystem pattern**.
The OSS-based software ecosystem attracts providers of open source software services that are non-commercially motivated. The high degree of openness enhances creativity. Read about this pattern and its design decisions.

Fig. 9.4 SecoArc Pattern Suggestion Perspective for PoC (Screenshot)

example, *free licensing*, *open publish*, and *free platform* support the fact that all source code should be free to use as mentioned as a part of the business vision.

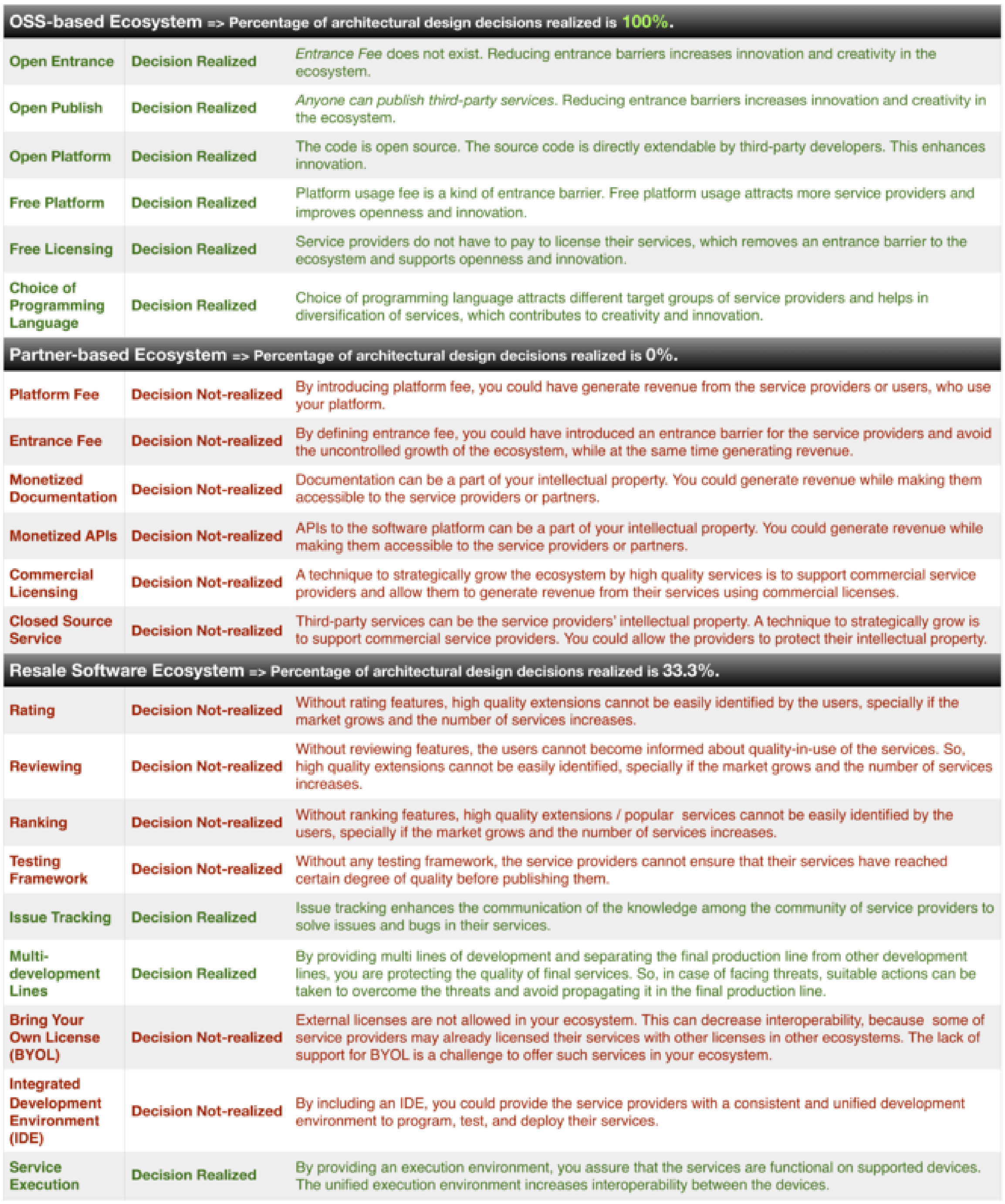
Some design decisions in *Architecture #1* can help the ecosystem grow. This can be seen in the report, where *issue tracking*, *multi-development lines*, and *service execution* are part of the RSE pattern, which aims at business scalability. For instance, by including the multi-lines of development, the PoC team can specify different groups of service providers based on providers' expertise or other requirements. Thereby, giving more reliable or trusted providers access to the development of final services.

Furthermore, several design decisions are not realized in the architecture. Importantly, none of the decisions of the PSE pattern is realized by *Architecture #1*. This is due to the fact that the PoC platform did not have any commercial aspect until now. However, by referring to the report, the team considered including some of the decisions such as rating, ranking, and platform fee to introduce a lightweight protection strategy to control the service quality.

Analysis of Architecture #2 Similar to *Architecture #1*, *Architecture #2* was modeled and analyzed by the architect. Figure 9.6 portrays the report of analysis for *Architecture #2*.

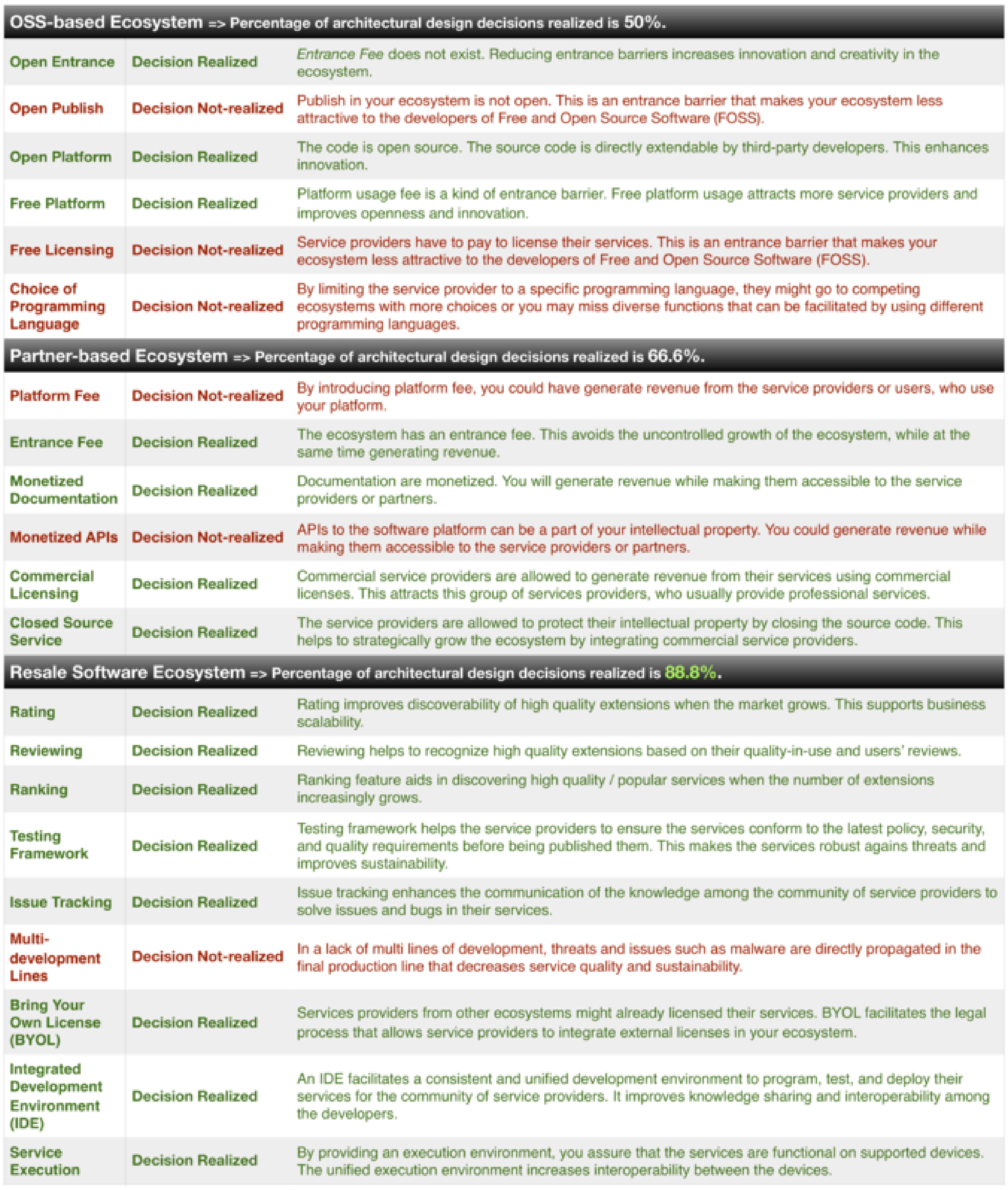
While the architecture partially matches all the patterns, it mostly realizes the decisions of the RSE pattern. This is because of the decisions that support the third-party providers' independence in provisioning software services in the ecosystem. For instance, *issue tracking* and *IDE* support the providers during software development while features like *rating* and *ranking* provide the third-party providers with feedback regarding the quality and market success of their services.

Furthermore, the PSE and OSS patterns respectively take the second and third places with respect to being realized by the architecture. Taking a closer look at the realized decisions shows that the partial fulfillment of the decision constraints in the



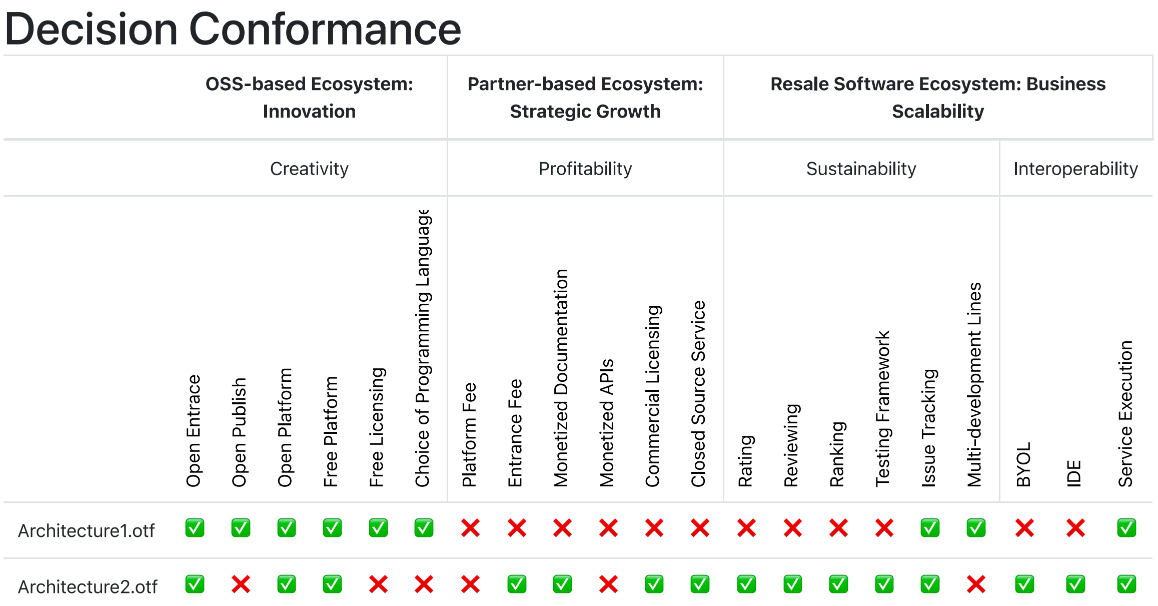
OSS-based Ecosystem => Percentage of architectural design decisions realized is 100%.		
Open Entrance	Decision Realized	*Entrance Fee* does not exist. Reducing entrance barriers increases innovation and creativity in the ecosystem.
Open Publish	Decision Realized	*Anyone can publish third-party services*. Reducing entrance barriers increases innovation and creativity in the ecosystem.
Open Platform	Decision Realized	The code is open source. The source code is directly extendable by third-party developers. This enhances innovation.
Free Platform	Decision Realized	Platform usage fee is a kind of entrance barrier. Free platform usage attracts more service providers and improves openness and innovation.
Free Licensing	Decision Realized	Service providers do not have to pay to license their services, which removes an entrance barrier to the ecosystem and supports openness and innovation.
Choice of Programming Language	Decision Realized	Choice of programming language attracts different target groups of service providers and helps in diversification of services, which contributes to creativity and innovation.
Partner-based Ecosystem => Percentage of architectural design decisions realized is 0%.		
Platform Fee	Decision Not-realized	By introducing platform fee, you could have generate revenue from the service providers or users, who use your platform.
Entrance Fee	Decision Not-realized	By defining entrance fee, you could have introduced an entrance barrier for the service providers and avoid the uncontrolled growth of the ecosystem, while at the same time generating revenue.
Monetized Documentation	Decision Not-realized	Documentation can be a part of your intellectual property. You could generate revenue while making them accessible to the service providers or partners.
Monetized APIs	Decision Not-realized	APIs to the software platform can be a part of your intellectual property. You could generate revenue while making them accessible to the service providers or partners.
Commercial Licensing	Decision Not-realized	A technique to strategically grow the ecosystem by high quality services is to support commercial service providers and allow them to generate revenue from their services using commercial licenses.
Closed Source Service	Decision Not-realized	Third-party services can be the service providers' intellectual property. A technique to strategically grow is to support commercial service providers. You could allow the providers to protect their intellectual property.
Resale Software Ecosystem => Percentage of architectural design decisions realized is 33.3%.		
Rating	Decision Not-realized	Without rating features, high quality extensions cannot be easily identified by the users, specially if the market grows and the number of services increases.
Reviewing	Decision Not-realized	Without reviewing features, the users cannot become informed about quality-in-use of the services. So, high quality extensions cannot be easily identified, specially if the market grows and the number of services increases.
Ranking	Decision Not-realized	Without ranking features, high quality extensions / popular services cannot be easily identified by the users, specially if the market grows and the number of services increases.
Testing Framework	Decision Not-realized	Without any testing framework, the service providers cannot ensure that their services have reached certain degree of quality before publishing them.
Issue Tracking	Decision Realized	Issue tracking enhances the communication of the knowledge among the community of service providers to solve issues and bugs in their services.
Multi-development Lines	Decision Realized	By providing multi lines of development and separating the final production line from other development lines, you are protecting the quality of final services. So, in case of facing threats, suitable actions can be taken to overcome the threats and avoid propagating it in the final production line.
Bring Your Own License (BYOL)	Decision Not-realized	External licenses are not allowed in your ecosystem. This can decrease interoperability, because some of service providers may already licensed their services with other licenses in other ecosystems. The lack of support for BYOL is a challenge to offer such services in your ecosystem.
Integrated Development Environment (IDE)	Decision Not-realized	By including an IDE, you could provide the service providers with a consistent and unified development environment to program, test, and deploy their services.
Service Execution	Decision Realized	By providing an execution environment, you assure that the services are functional on supported devices. The unified execution environment increases interoperability between the devices.

Fig. 9.5 Report of Analysis for *Architecture #1* (Screenshot)



OSS-based Ecosystem => Percentage of architectural design decisions realized is 50%.		
Open Entrance	Decision Realized	*Entrance Fee* does not exist. Reducing entrance barriers increases innovation and creativity in the ecosystem.
Open Publish	Decision Not-realized	Publish in your ecosystem is not open. This is an entrance barrier that makes your ecosystem less attractive to the developers of Free and Open Source Software (FOSS).
Open Platform	Decision Realized	The code is open source. The source code is directly extendable by third-party developers. This enhances innovation.
Free Platform	Decision Realized	Platform usage fee is a kind of entrance barrier. Free platform usage attracts more service providers and improves openness and innovation.
Free Licensing	Decision Not-realized	Service providers have to pay to license their services. This is an entrance barrier that makes your ecosystem less attractive to the developers of Free and Open Source Software (FOSS).
Choice of Programming Language	Decision Not-realized	By limiting the service provider to a specific programming language, they might go to competing ecosystems with more choices or you may miss diverse functions that can be facilitated by using different programming languages.
Partner-based Ecosystem => Percentage of architectural design decisions realized is 66.6%.		
Platform Fee	Decision Not-realized	By introducing platform fee, you could have generate revenue from the service providers or users, who use your platform.
Entrance Fee	Decision Realized	The ecosystem has an entrance fee. This avoids the uncontrolled growth of the ecosystem, while at the same time generating revenue.
Monetized Documentation	Decision Realized	Documentation are monetized. You will generate revenue while making them accessible to the service providers or partners.
Monetized APIs	Decision Not-realized	APIs to the software platform can be a part of your intellectual property. You could generate revenue while making them accessible to the service providers or partners.
Commercial Licensing	Decision Realized	Commercial service providers are allowed to generate revenue from their services using commercial licenses. This attracts this group of services providers, who usually provide professional services.
Closed Source Service	Decision Realized	The service providers are allowed to protect their intellectual property by closing the source code. This helps to strategically grow the ecosystem by integrating commercial service providers.
Resale Software Ecosystem => Percentage of architectural design decisions realized is 88.8%.		
Rating	Decision Realized	Rating improves discoverability of high quality extensions when the market grows. This supports business scalability.
Reviewing	Decision Realized	Reviewing helps to recognize high quality extensions based on their quality-in-use and users' reviews.
Ranking	Decision Realized	Ranking feature aids in discovering high quality / popular services when the number of extensions increasingly grows.
Testing Framework	Decision Realized	Testing framework helps the service providers to ensure the services conform to the latest policy, security, and quality requirements before being published them. This makes the services robust agains threats and improves sustainability.
Issue Tracking	Decision Realized	Issue tracking enhances the communication of the knowledge among the community of service providers to solve issues and bugs in their services.
Multi-development Lines	Decision Not-realized	In a lack of multi lines of development, threats and issues such as malware are directly propagated in the final production line that decreases service quality and sustainability.
Bring Your Own License (BYOL)	Decision Realized	Services providers from other ecosystems might already licensed their services. BYOL facilitates the legal process that allows service providers to integrate external licenses in your ecosystem.
Integrated Development Environment (IDE)	Decision Realized	An IDE facilitates a consistent and unified development environment to program, test, and deploy their services for the community of service providers. It improves knowledge sharing and interoperability among the developers.
Service Execution	Decision Realized	By providing an execution environment, you assure that the services are functional on supported devices. The unified execution environment increases interoperability between the devices.

Fig. 9.6 Report of Analysis for *Architecture #2* (Screenshot)



Decision Conformance

	OSS-based Ecosystem: Innovation						Partner-based Ecosystem: Strategic Growth						Resale Software Ecosystem: Business Scalability								
	Creativity						Profitability						Sustainability						Interoperability		
	Open Entrace	Open Publish	Open Platform	Free Platform	Free Licensing	Choice of Programming Language	Platform Fee	Entrance Fee	Monetized Documentation	Monetized APIs	Commercial Licensing	Closed Source Service	Rating	Reviewing	Ranking	Testing Framework	Issue Tracking	Multi-development Lines	BYOL	IDE	Service Execution
Architecture1.otf	✅	✅	✅	✅	✅	✅	❌	❌	❌	❌	❌	❌	❌	❌	❌	❌	✅	✅	❌	❌	✅
Architecture2.otf	✅	❌	✅	✅	❌	❌	❌	✅	✅	❌	✅	✅	✅	✅	✅	✅	✅	❌	✅	✅	✅

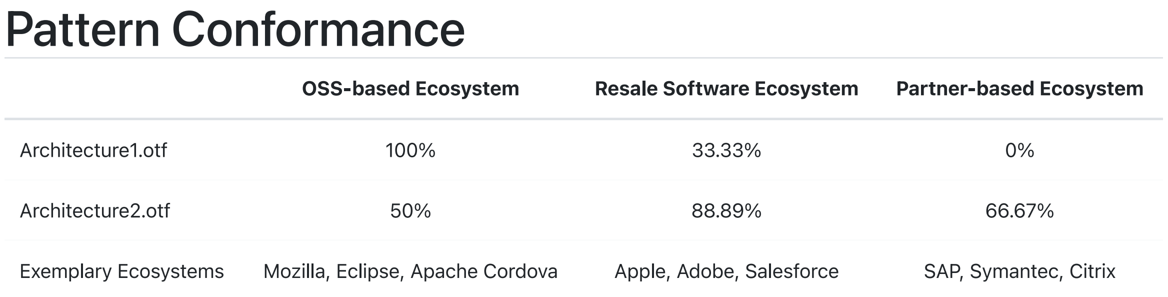
Fig. 9.7 SecoArc Decision Conformance Perspective:
A Comparison of *Architecture #1* and *Architecture #2*
(Screenshot from the SecoArc Tool Support)

both patterns relies on the openness policies chosen for *Architecture #2* that originate from the semi-open nature of the ecosystem.

Comparison of Architecture #1 and Architecture #2 The PoC architect performs a comparison of Architectures #1 and #2 by using the comparative view. In this view, the results of architectural analysis for more than one architecture are presented in a unified view in the tool support as described in Section 8.7

Table 9.7 depicts the results of the decision conformance perspective that shows how the architectures match with the decisions of the patterns. Accordingly, *Architectures #1* and *#2* respectively fulfill OSS-based Ecosystem (*100%*, *50%*), Partner-based Ecosystem (*0%*, *66.6%*), and Resale Software Ecosystem (*33.3%*, *88.8%*). This means, *Architecture #1* fulfills **innovation** the most, while *Architecture #2* enhances **business scalability** and **strategic growth**. Furthermore, the majority of decisions in *Architecture #1* contributes to **creativity** while **sustainability** and **interoperability** are partially supported. The main shift in *Architecture #2* is to include partners, who can help enhance revenue generation. Decisions like the entrance fee and commercial licenses support **profitability**.

Figure 9.8 summarizes the result of pattern conformance for the both cases. *Architecture #1* and *Architecture #2* respectively conform to the OSS-based ecosystem and resale software ecosystem patterns the most. The percentages show the extent to which the ecosystem architecture matches the decisions of the patterns. By referring to the exemplary ecosystems, the platform provider can orient themselves through the process of decision-making.



Pattern Conformance

	OSS-based Ecosystem	Resale Software Ecosystem	Partner-based Ecosystem
Architecture1.otf	100%	33.33%	0%
Architecture2.otf	50%	88.89%	66.67%
Exemplary Ecosystems	Mozilla, Eclipse, Apache Cordova	Apple, Adobe, Salesforce	SAP, Symantec, Citrix

Fig. 9.8 SecoArc Pattern Conformance Perspective:
A Comparison of *Architecture # 1* and *Architecture # 2* (Screenshot)

In summary, the comparative view shows that *Architecture #1* is a more suitable candidate with respect to the business vision presented in Section 9.2.2, as it fulfills innovation to a better extent (*100%* vs. *50%*). However, sustainability can be improved by considering the relevant design decisions, e.g., including rating and ranking. In addition, the analysis shows that existing ecosystems, with contextual factors similar to the PoC, conform to the OSS-based Ecosystem pattern, which emphasizes the suitability of this pattern for the PoC. Furthermore, exemplary ecosystems are provided so that the PoC team can additionally reflect on the suitability of each pattern by referring to the real-world instances of the architectural patterns.

Semi-Structured Interview

We conduct an interview with the PoC architect to assess the suitability of the SecoArc framework for designing software ecosystems. We design a questionnaire by referring to the sub-characteristics of functional suitability, i.e., *completeness*, *correctness*, and *appropriateness* [ISO12] and casting doubt on the design artifacts of SecoArc, i.e., the *design process*, *modeling framework*, and *architectural analysis technique*, as well as the *tool support.*

As shown in Table 9.3, the questionnaire consists of *17* questions. The interviewee responds to the questions in three scales, i.e., *No Partially Yes*, while reasoning about the responses. Using the semi-structured interview technique [HA05], we collect data on strong and weak aspects of SecoArc based on open conversations.

Table 9.3 Interview Questionnaire and Results

Modeling Language & Design Process	***Q1***	*Does the domain knowledge of SecoArc contribute to familiarizing you with critical design decisions of the ecosystem around your platform?*	+
	Q2	*Do the design process appropriately guide you through the ecosystem design?*	+
	Q3	*Are the business decisions, application and infrastructure modeling features complete in terms of being sufficient for modeling your ecosystem?*	+
	Q4	*Do you find the suggested business decisions and built-in application and infrastructure features correct (or rather contradictory or wrong)?*	+
	Q5	*Does grouping of features help you to better understand the design space?*	+
	Q6	*Are you able to appropriately express custom decisions of your ecosystem?*	+
	Q7	*Are the visual notation of business decisions and application and infrastructure features appropriate?*	+
	Q8	*Are the architectural models easy to change or reuse, so you can use them during the system evolution?*	o
	Q9	*Can you appropriately express the relationships between the three architectures?*	+
Architectural Analysis Technique	***Q10***	*Is the knowledge of existing ecosystems, embedded in the patterns, beneficial for your decision-making?*	+
	Q11	*Does the analysis help you to consider complementary business decisions and application and infrastructure features wrt. your business objectives?*	+
	Q12	*Does the analysis make you aware of the correctness of your design decisions? E.g., by revealing the contradictory decisions to your business objectives and quality attributes?*	+
	Q13	*Do you find the quality attributes and business objectives complete to address your needs?*	o
	Q14	*Is the comparison of alternative architectures beneficial to your decision-making?*	+
Tool Support	***Q15***	*Is the modeling workbench easy-to-use to design the ecosystem around your platform?*	+
	Q16	*Are the provided specification and guide materials satisfactory to start using the tool?*	+
	Q17	*Does the tool provide the functionality and features promised by the specification?*	+

No (-) / Partially (o) / Yes (+)

Analysis of Interview Results

In summary, the interviewee expresses that the the components of SecoArc implemented in the tool contribute to improved decision-making. Table 9.3 briefly refers to the responses. In the following, we report the results.

1) Modeling Language and Design Process: The architect approves that the modeling language of SecoArc is a domain-specific language. With the design process, they help with familiarizing oneself with the novel and complex architecture of the ecosystem around the platform. An good example is the set of business decisions to define platform and store openness that can now be directly specified in the architecture by using the framework. The built-in features provide a correct basis to explicitly design the ecosystem-specific design decisions of PoC. Furthermore, custom features can appropriately be added. This adds flexibility to the language because the custom features are treated the same way as the built-in features, e.g., to group them or relate them to the human actors. Another point of appropriateness is the abstraction provided by the visual notations and the central presentation of business decisions.

Although our approach is concerned with a modeling support at design time, there is an inherent need for consistency between the models and the code (Q8). A way to address this issue is to apply Aspect-Oriented Software Development to weave the decisions in the code, another way is to introduce traceability links [KZ10].

2) Architectural Analysis Technique: The architect finds it beneficial during the decision-making to have an estimation of the suitability of the PoC architecture based on the practice-proven knowledge of existing ecosystems and being able to compare different architectures. This also helps take further actions by including complementary decisions or re-considering the ones that threaten the business objectives and quality attributes.

Future research on extensive quality models for software ecosystems is required, to address more quality attributes, and their trade-offs. In addition, while the SecoArc framework captures relevant business objectives to software ecosystems, there is a need for methodological approaches that can be used to align the architecture with other business objectives that platform providers can have (Q13).

3) Tool Support: Availability of the tool and guide material impacts the uptake by enabling the architect to work with the ecosystem-related concepts right away. The tool is consistent with the functions described in the specification.

9.2.3 Validation Questions Revisited

In this section, we discuss the validation questions introduced in Section 9.2.1 based on the results of the case study described in Section 9.2.2.

VQ1: Does the SecoArc knowledge base facilitate designing software ecosystems?

Our experience from the case study as well as the results of the interview show the SecoArc framework facilitates the design of software ecosystems by enhancing modeling ecosystem architectures. During the interview, the architect positively replied to questions directly related to the correctness, appropriateness, and completeness of the SecoArc framework (M1). In particular, the domain knowledge contributed to making the critical decision points in the ecosystem architecture explicit. Furthermore, the openness policies that are a critical part of business decisions could be designed appropriately in the models by means of business strategies.

The availability of the tool and guide materials impacted the uptake. Our experience from the case study showed that training efforts were acceptable since the architect installed the tool on his own without our assistance by using the available links to the source code on a public repository (M2). He required 20 minutes to successfully install the tool prior to the experiment that we consider as a reasonable amount of time. The efforts included importing two sets of projects in the workspace: The modeling projects related to domain model and a Sirius specification project related to the visual notation. During this process, the architect used the SecoArc specification to setup the modeling environment. Furthermore, the architect used an Eclipse package provided by the Obeo Designer community[4] that had the Eclipse Modeling Framework and Sirius already installed in it. The architect also reported later that he did not face any difficulty during the installation. Furthermore, he read the SecoArc specification regarding the design and analysis functions provided by the tool, but not with respect to the semantics of the visual notation. Instead, he received a 20-minute guide from us, before the experiment started. In total, the architect required 1.5 hours to model two architectures. The architectures were modeled based on a blueprint of the PoC architecture that existed from the past in form of a UML Components Diagram.

VQ2: Does the SecoArc methodical knowledge facilitate designing software ecosystems?

The results of the interview show that the methodical knowledge provides useful information on the linkage of the architectural design decisions made in the architecture to the business objectives and quality attributes of ecosystem health.

12 out of 20 process steps are performed by the study. The 20 process steps comprise the SecoArc design process, goal and context specification, applying patterns, and architectural analysis (M3). The steps that are not tested by the validation study relate to the process of applying patterns. The reason is that the ecosystem architectures were designed freely and manually in the case study. The quality of the architectures were later analyzed using the architectural analysis technique. The interview showed that the analysis results help obtaining an improved understanding of the ecosystem health by getting informed about the supported and potentially degraded quality attributes (M4). The result of analysis demonstrated that, regarding *Architecture #1*, 9 out of 22

[4]`https://www.obeodesigner.com`, Last Access: January 10, 2022

and, regarding *Architecture #2*, 15 out of 22 design decisions related to the ecosystem health are applied (M5). This result shows that the framework provides relevant knowledge that can be used to address ecosystem health. The decisions that were not realized in the architecture were identified as potential deficiencies. In total, 13 improvement suggestions for *Architecture #1* and 7 improvement suggestions regarding *Architecture #2* were given (M6). The only decision that was not applied in any of the architecture was *platform fee*. Since other kinds of fees are applied in one of the architectures, we conclude that the fact that one decision is not applied is mainly related to the context and use cases in the case study.

The validation activities confirm the applicability of the SecoArc architectural analysis technique while a key part of the design enhancement performed in the case study relied on the results of architectural analysis. Using the architectural perspectives, the two ecosystem architectures could be compared in a quantifiable way, i.e., the extents to which the design decisions and quality attributes were fulfilled are provided. The architectural analysis took half an hour (M7). In general, the architect found the modeling workbench easy to use. The amount of time spent for analyzing the two architectures shows an acceptable degree of the efficiency to work with the framework.

Nevertheless, the SecoArc framework is exposed to certain limitations. In Section 9.4, we discuss the threats to validity of our solution concepts that we encounter while performing the case study.

9.3 Examination of Existing Ecosystems

We continue validating the suitability of the SecoArc framework. In section 9.2, we validated the framework by involving the user. In the second study, we assess the framework, and specifically the variability model, with respect to the real world ecosystems. The reason for choosing the variability model is its importance for the SecoArc framework by being the basis of the architectural patterns. Note that the validation study has been performed by means of an early version of the variability model. The early version included business, application, and infrastructure variation points. Therefore, the social and organizational aspect was not part of the study.

9.3.1 Validation Questions

We follow the goal-question-metric approach for the second validation study as shown in Table 9.4. In the following, we present the validation questions and metrics. These validation questions are concerned with RQ3, which is a part of our design science problem-solving approach to answer RQ2.

Table 9.4 Goal Question Metric Applied in the Thesis

	Goal	
	Purpose:	Analyze whether
	Issue:	the SecoArc variability model can capture
	Object:	architectural variabilities of software ecosystems in different application domains
	Viewpoint:	from an ecosystem platform provider's viewpoint
Questions	**Metrics**	
VC3 Does the variability model represent design decisions of the real world ecosystems?	M08: Numbers of concepts that can not be mapped to any of the variants M09: Numbers of concepts that can be mapped to more than one variant M10: Numbers of variants that can be mapped to more than one concept M11: Numbers of variants that is not realized in the ecosystems	
VC4 Does the variability model capture interdependencies between architectural design decisions?	M12: Number of dependencies instantiated by existing ecosystems	

VQ3: Does the variability model represent design decisions of the real world ecosystems?

This question casts doubt on whether the variability model is a true abstraction of the real world. To answer this question, we use the evaluation criteria introduced by Guizzardi et al. [GPVS05]. The criteria are shown as metrics in Table 9.4. The authors proposes four measurement criteria to decide on suitability of a domain abstraction with respect to the concepts in real world. A domain abstraction is a conceptualization of a given domain. To evaluate suitability of the domain abstraction, there needs to be an isomorphic relation between the abstraction and concrete models of that domain. If this relation is not held the models can not be precisely and uniquely mapped to the abstraction, e.g., there can be various abstractions for a concept in a model. In our study, the variability model represents the domain abstraction conceptualized in this thesis while the concepts extracted from the existing ecosystem represent the model.

VQ4: Does the variability model capture interdependencies between architectural design decisions?

We show several instantiations of the decision interdependencies by means of this validation study. This help us obtain further concrete implications regarding the interdependencies.

9.3.2 Study 2: Comparative Analysis of Salesforce, Apple, Amazon AWS, Eclipse, and Amazon Alexa Ecosystems

In this section, we present the results of examining five real-world ecosystems based on the variability model. For this examination, we chose five open platforms, i.e., *Salesforce* and *Amazon AWS* from the domains of enterprise application and cloud computing, *Apple iOS* from the mobile application domain, *Eclipse IDE* from the open-source development, and *Amazon Alexa* from the domain of IoT and connected devices. The criteria to choose these platforms were that they must be mature ecosystems and provide software in different application domains.

Our analysis provides concrete instances for the variants of the variability model as shown in Tables 9.5–9.7. Each cell of the tables provides an instance of a variant of the variability model. We consider each of these instances an architectural design decision because they represent the way that each ecosystem realizes a variant. We use "Not realized" when an ecosystem does not realize a variant. These design decisions have been taken by the platform providers. Each table is dedicated to one architectural aspect of software ecosystems, i.e., business, application, or infrastructure aspect.

In the following, we briefly introduce the software companies, which provide the platforms of our study:

Salesforce.com is a provider of customer relationship management and enterprise services. Salesforce and force.com are its main software platforms. On top of these platforms, a working environment including Salesforce proprietary services and third-party Apps is built. Independent developers publish third-party Apps on the Salesforce's store, namely, AppExchange[5].

Apple Inc. is the provider of mobile and desktop hardware assets (iPhone, MacBook, etc.) and operating systems (iOS and macOS). iOS acts as a platform for

[5] `appexchange.salesforce.com`, Last Access: January 10, 2022

Variation Point	Variant	Provider: salesforce.com Platform: Salesforce and force.com Store: AppExchange	Provider: Apple Inc. Platform: iOS Store: Apple App Store	Provider: Amazon.com Platform: AWS Store: AWS Marketplace	Provider: Eclipse Foundation Platform: Eclipse IDE Store: Eclipse Marketplace	Provider: Amazon.com Platform: Alexa Store: Alexa Skills Store
VP1: Complem-entary Partnership	**V1.1: Strategic Partner**	- Deloitte Digital Hub: A customer relationship management web-based App (Deloitte Digital) - Telco Sales-360: A preconfigured telecommunication solution (Tech Mahindra)	- Business and enterprise Apps supported by cloud services optimised for iOS (IBM) - Enterprise Next: A set of consulting services on Apple devices (Deloitte)	- Commercial vendors including IBM, Microsoft, SAP, 10gen, CA, Couchbase, Canonical - Open-source provision from Nginx, Drupal, etc.	- Eclipse Foundation: The core member-based decision makers - Several projects with different degrees of contribution	- Build-in services for several partners, e.g., Spotify, WeMo, Nest, Uber, etc.
	V1.2: Supplier	- Databases, Exadata, and Java platform (Oracle) - Servers with AMD processors (Dell) - Emailing system (MessageSystems). - Security assessment (Symantec) (KPMG) - CDN by a partner	- CDN to deliver App Store contents (Level 3) - Processors (Intel, historically from Samsung and TSMC) - Modems (Intel and Qualcomm depending on local telco provider) - Networking services (AT&T and Verizon)	- AWS Test Drive (Orbitera) - SaaS-based migration and Disaster Recovery solutions (CloudEndure) - Many telco partnerships	Not realized	- Chips (Intel, Conexant) - Voice processing hardware (Conexant)
	V1.3: Independent Developer	- Developers of Apps on Salesforce AppExchange	- Developers of Apps on Apple App Store	- ISVs - Cloud service providers - Consulting partners	- Eclipse committers - Individual projects (e.g., TopCased)	- Developers of Alexa Skills
VP2: Fee	**V2.1: Platform Fee**	- The platform fee varies with platform licenses	- Fees of buying Apple assets	- Cost of using AWS (e.g., pay per hour or per day)	Not realized (Open-source, non-profit, and free)	- Fees of buying Eco assets
	V2.2: Entrance Fee	- Annual fee to list Apps on the store - Different fees for different partnership tiers	- Annual fee to list Apps on the store - Different fees for independent developers and enterprises	- One time fee to list commercial services on the store - No fee for free services	Not realized (No fee to list plug-ins on the store)	Not realized (No fee to list skills on the store, No registration cost)
VP3: Openness	**V3.1: Open**	- force.com IDE - SDKs for iOS and Android - Aura UI Framework	- A few libraries like Open Source Reference Library	- AWS SDKs - Open Source Software projects for AWS	- Eclipse IDE - Eclipse projects (decided by developer)	- Skills Kit SDK for Node.js and Java
	V3.2: Closed	- Most of the platform frameworks	- Most of the platform frameworks	- The platform is mainly closed	- Based on developers' decision	- The platform is mainly closed
VP4: Licensing	**V4.1: Public Licensing**	- Developers are free to use public licenses for their source code	- Developers are free to use public licenses for their source code	- Externally licensed services (BYOL)	- Developers are free to use other public licenses	Not realized
	V4.2: Ecosystem-specific Licensing	- License Management App (LMA): a tool to enable developers to define licenses for their Apps	- Apple performs the licensing tasks - Developers provide metadata for their App	- Categories of licenses: Commercial software, free, and open Source	- Eclipse Public License - Eclipse Distribution License	- General Amazon Program Materials License Agreement

Table 9.5 Business-Related Design Decisions of Existing Ecosystems

Variation Point	**Variant**	**Provider: salesforce.com Platform: Salesforce and force.com Store: AppExchange**	**Provider: Apple Inc. Platform: iOS Store: Apple App Store**	**Provider: Amazon.com Platform: AWS Store: AWS Marketplace**	**Provider: Eclipse Foundation Platform: Eclipse IDE Store: Eclipse Marketplace**	**Provider: Amazon.com Platform: Alexa Store: Alexa Skills Store**
VP5: Deliverable	**V5.1: Software Product**	- CRM Apps	- Mobile Apps	Not realized	- Eclipse plug-ins	Not realized
	V5.2: Software Service	- Cloud services, e.g., work.com	- iCloud services	- AWS-based SaaS, PaaS, and IaaS	- Eclipse Cloud Development - Eclipse Che	- Alexa Skills
	V5.3: Source Code	- Some Apps on AppExchange	- Some Apps (e.g., on GitHub)	- Open source services	- Most of plug.ins and projects	Not realized
VP6: Extension Development	**V6.1: Platform SDK**	- Force.com SDK - Aura UI framework	- iOS SDK: to develop native iOS Apps	- AWS SDKs, e.g., AWS Lambda - Command line and Powershell	- Eclipse SDK: to develop Java applications	- Alexa Skills Kit - Amazon Lex
	V6.2: IDE	- force.com IDE: eclipse plug-in - Point-and-Click App Building	- Xcode IDE	- AWS Tool Kits: Eclipse and Visual Studio plug-ins - AWS web console	- Eclipse IDE (platform)	- Skill Builder to design voice interaction models
	V6.3: Programming Language	- Apex (proprietary) - Visualforce (proprietary)	- Swift (general-purpose programming language)	- General-purpose languages - Amazon Simple Queue Service	- Java: The main development language	- Any programmin g language - JSON to link skills to users' inputs
	V6.4: Communic-ation Protocol	- SOAP API and REST API	- Rest API - XMPP, SIP, etc.	- REST API - SOAP over HTTP	- Internet protocols, e.g., HTTP, TCP	- HTTP(S)
	V6.5: Testing	- Apex testing framework: Basically to create unit tests	- Xcode testing - TestFlight: testing by external testers	- Unit Testing in Eclipse and Visual Studio - AWS Device Farm	- JUnit testing framework	- Service Simulator - Echosim.io
VP7: Platform Interfaces	**V7.1: User Interface**	- VisualEditor Namespace, Canvas Namespace, etc.	- Cocoa Touch Frameworks - Partly Media Frameworks	Not realized	- org.eclipse.ui.p erspectives - org.eclipse.swt - org.eclipse.ui	- Alexa video features
	V7.2: Platform Features	- Search Namespace - *ChatterAnswe r* Namespace - Datacloud Namespace	- Media Frameworks (e.g., audio) - Core Services Frameworks (e.g., iCloud)	- Amazon Simple Email Service - Amazon Elastic MapReduce - Amazon CloudSearch	- Docker APIs - EMF (e.g., org.eclipse.linu xtools.docker.fe ature.*)	- Alexa Voice Service libraries
	V7.3: System Libraries	- System, Messaging, and Database Namespaces	- Core OS Frameworks, e.g., file-system access and memory allocation	- Amazon Elastic Load Balancing - CloudWatch: monitoring the AWS	- I/O Streams java.io package - Memory management java.nio.Buffer	- Core voice-enabling technologies
VP8: Security Check	**V8.1: Leaks and Bugs**	- AppExchange Certification (Required by V3.2)	- Store review process: checking broken links, performance, etc. (Required by V3.2)	- Store review process: checking services and their metadata	- Intensive review of Eclipse projects (Required by V3.2) - No checking for plug-ins	- Store certification process (Required by V3.2)
	V8.2: Policy Violation	- Store review process: Apps and their listing data	- Store review process: software and hardware compatibility	- Self-service AMI scanning tool: Checking by providers	- Moderation process: checking plug-ins relevance	- Store submission checklist

Table 9.6 Application-Related Design Decisions of Existing Ecosystems

Variation Point	Variant	Provider: salesforce.com Platform: Salesforce and force.com Store: AppExchange	Provider: Apple Inc. Platform: iOS Store: Apple App Store	Provider: Amazon.com Platform: AWS Store: AWS Marketplace	Provider: Eclipse Foundation Platform: Eclipse IDE Store: Eclipse Marketplace	Provider: Amazon.com Platform: Alexa Store: Alexa Skills Store
VP9: Delivery Mode	**V9.1: Local Installation**	- Yes (Required by V5.1)	- Yes (Required by V5.1)	Not realized	- Yes (Required by V5.1, V5.3)	Not realized
	V9.2: Cloud Delivery	- Yes (Required by V5.2)	- Yes (Required by V5.2)	- Yes (Required by V5.2)	- Yes (Required by V5.2)	- Yes (Required by V5.2)
VP10: Service Execution	**V10.1: Operating System**	- The Salesforce Operating System (SOS) (Required by V9.2)	- iOS (platform)	- Amazon Linux: a Linux-based OS for AWS usage (Required by V9.2)	Not realized	- Amazon Alexa (Required by V9.2)
	V10.2.: Compute Center	- Requires V1.2 (external supplier) (Required by V9.2)	- Virtual compute centres (Required by V9.2)	- AWS servers - Requires V1.2 (external supplier of TestDrive) (Required by V9.2)	Not realized	- AWS - Third-party web servers (Required by V9.2)
	V10.3: Data Centre	- Requires V1.2 (external supplier) (Required by V9.2)	- Apple data center (Required by V9.2)	- Amazon DynamoDB: NoSQL database (Required by V9.2)	Not realized	- Amazon data centers (Required by V9.2)
VP11: Service Delivery	**V11.1: Telecommun-ication**	Not realized	- Requires V1.2 (external supplier) (Required by V9.2)	- Requires V1.2 (external supplier) (Required by V9.2)	Not realized	Not realized
	V11.2: Content Delivery Network	- Requires V1.2 (external supplier) (Required by V9.2)	- Requires V1.2 (external supplier) (Required by V9.2)	- Amazon ElastiCache: in-memory caching (Required by V9.2) - Amazon CloudFront	Not realized	Not realized
	V11.3: Exclusively WWW	- Access to the ecosystem, e.g., AppExchange	- iOS Apps providing services over Internet (Required by V9.2)	- Purchase on Marketplace	- Service and product delivery over Internet	- Service delivery over Internet (Required by V9.2)
VP12: Asset	**V12.1: Asset**	Salesforce servers	- iPhone, iPad (Requires V1.2)	AWS servers	Not realized	- AWS and Amazon Echo (speakers)

Table 9.7 Infrastructure-Related Design Decisions of Existing Ecosystems

independent developers to develop mobile Apps on top of it. Such Apps can be made available for mobile users on Apple App Store.

Amazon.com is an e-commerce company and the provider of Amazon Web Services (AWS). AWS refers to a wide range of SaaS, PaaS, and IaaS services. Other service providers can customize instances of AWS and trade them on AWS marketplace[6]. AWS marketplace offers such providers licensing and billing services. In addition, Amazon.com is the provider of a smart voice assistant platform, namely Alexa. Alexa is designed to handle tasks like home automation and controlling connected smart devices. Independent developers develop third-party applications, namely Alexa skills, for the Alexa platform. These skills are published on Alexa Skill Store[7].

Eclipse Foundation consists of a hierarchy of leading and contributing projects, which govern the Eclipse ecosystem. Eclipse IDE is the software platform for open-source software communities. An organization may officially become a member of the ecosystem. Members influence strategic decisions. The degree of influence depends on the level of contribution. Entering the ecosystem as an independent developer is characterized by providing Eclipse-relevant plug-ins. Eclipse marketplace is the online store, where projects and plug-ins are published.

At the business level, the different ecosystem openness is achieved using decisions related to *VP2: Fee*, *VP3: Openness*, and *VP4: Licensing*. At the application level, it is defined how the software can be developed at top of the platforms while the decisions at the infrastructure level concern the technologies that enable service delivery and execution.

In addition to the design decisions, our examination revealed the interdependencies that exist among the design decisions. Tables 9.5–9.7 refers to these interdependencies by using the keyword *requires*. If *Variant 1* requires *Variant 2*, then the existence of *Variant 1* in the architecture depends on *Variant 2*. These dependencies conform to the meta definition of the dependencies in the variability model as discussed in Section 6.1.3. We notice a certain type of dependency frequently repeated in the ecosystems examined in our study, i.e., the situations that the provision of infrastructure resources and services such as data centers and content delivery networks are often performed by the third-party providers (*requires V1.2*).

In summary, the examination of the Salesforce, Apple, Amazon AWS, Eclipse, and Amazon Alexa ecosystems confirms that the platform providers took a wide range and

[6]`aws.amazon.com/marketplace`, Last Access: January 10, 2022
[7]`www.amazon.com/b?node=13727921011`, Last Access: January 10, 2022.

inter-dependent range of architectural design decisions. Another common aspect to notice was the deligation of infrastructural service provision to the third-party partners.

9.3.3 Validation Questions Revisited

In this section, we revisit the validation questions mentioned in Section 9.3.1 by referring to the results of examining existing ecosystems in Section 9.3.2.

VQ3: Does the variability model represent design decisions of the real world ecosystems?

We identified concepts that could not be mapped to any of the variants. This resulted in identifying the variation point *store* and its variants, which we added to a later version of the variability model (M8). Our results did not reveal any situation, where the concepts of the ecosystems could be mapped to more than one variants (M8).

Moreover, the results show that most of the variants can not be mapped to more than one concept. We noticed that *VP12: Asset* is mapped to the concepts that differ from each other. Thus, we identified a variation point that can be mapped to more than one concept (M10). Although asset is a variation point in the variability model rather than a variant, we have not identified any variant for it in our initial work. The identified variants of asset are *servers*, *personal computer*, and *mobile devices* that are used in different ecosystems for different purposes, which affect the decisions regarding service delivery, execution, etc.

The results show that all variation points have been realized by the ecosystems except for *VP: service execution* in the Eclipse ecosystem (M11). This variation point was declared as optional in the variability model. Furthermore, several variants were not realized in the study that conformed to the semantic of the variability model.

VQ4: Does the variability model capture interdependencies between architectural design decisions?

In total, we identified 38 dependency constraints that show 38 decisions (among 155 decisions) are the direct consequences of making the other decisions. This is 24% of the total decisions made (M12). These dependencies fall under variability dependency constraints introduced in Section 6.1.3. In Tables 9.5-9.7, we expressed these dependencies in two different ways. Firstly, the dependencies that are forced by another decision, denoted by *"required by"*. These are mainly service-related dependencies that refer to a situation when the service provisioning becomes a consequence of certain decision-making. Another kind of dependencies is denoted by *"requires"* that mainly express the supplier-related dependencies, specifically when a decision is executed by a third-party provider. Therefore, we can make a conclusion about the tasks that are usually delegated to the suppliers/partners. This is specially visible with respect to the variants of the infrastructure aspect. Except for the Amazon AWS ecosystem, which, itself, is the provider of cloud computing resources, computing tasks are often performed by external parties in the ecosystems.

In addition to the discussed ecosystems, further ecosystems (i.e., Adobe, Symantec, Citrix, and Cloud Foundry) were discussed on the basis of the architecture framework. The challenges that we faced during the conduction of our second validation study are discussed in Section 9.4.

9.4 Threats to Validity

In response to the challenges faced by platform providers to create software ecosystems, our goal in this PhD thesis was to provide systematic architectural knowledge that can facilitate informed architectural decision-making. We made the systematic knowledge available by extracting and conceptualizing the knowledge of existing ecosystems and related literature by performing several exhaustive examinations of these sources of knowledge. Having our contributions in mind (cf. Sections 9.2.3 and 9.3.3), we discuss threats to validity [WRH$^+$12] that our research approach has been exposed to.

Internal Validity

Internal validity refers to casual relations between investigated sources and extracted data, and the factors that might affect the findings but not being included in the study [WRH$^+$12]. A potential risk to the internal validity of our research was the limited access to the knowledge that could reveal platform providers' architectural design decisions while developing the SecoArc knowledge base (RQ3, RQ4). This knowledge could have been collected directly from the platform providers, e.g., by interviewing them. Considering that this thesis is concerned with a notably broad topic, we sought an appropriate abstraction by our research that covers architectural characteristics of as many ecosystems as possible. Yet, to minimize the effects of the risk mentioned above, we took two actions: a) To obtain a better understanding of the platform providers' business decisions, we read their business reports that are annually made public. b) To gain more insight into the technical characteristics of the ecosystems, we created accounts in those ecosystems and observed them from inside as a registered member. This was specifically useful since accessing technical documentation was often subject to registration as a developer in the developer portals. Furthermore, in order to answer our research questions, we used several systematic methods in our design science approach that enabled us to collect data from a variety of sources.

Construct Validity

Construct validity is concerned with whether the findings and conclusions are correct regarding the observed data [WRH$^+$12]. With this respect, a risk was the misunderstanding that usually emerges when people from different disciplines discuss some common topics. Parts of the process of solution development for RQ4 and RQ9 involved interviewing people. To mitigate the risks of misunderstanding, we aimed for semi-structured interviews that included sets of fixed questions accompanied with open-ended discussions. The goal was to ensure the consistency of our understanding with the interviewees regarding the similar terms with different meanings in different disciplines or the different terms used for the same concepts.

In addition, a potential risk is whether our conclusions are the results of an honest interpretation of the data by us (i.e., the thesis's writer). To mitigate this threat, we developed our research results in multiple iterations that included many talks with internal and external supervisors. We collected reviews from colleagues in the research group and from our reviewers at the conferences and workshops, where the results have been published. Moreover, to avoid loose and inaccurate interpretations of the data collected during the interviews, we conducted all the interviews with two interviewers while a second interviewer was responsible to write the transcripts for the interviewees' given answers. These included an interview with one architect within the scope of our validation study and ten interviews to identify service provision scenarios while each interview questioning one or two interviewees.

External Validity

External validity is about the generalizability of the findings [WRH$^+$12]. The validation study performed within the scope of on-the-fly computing is by interviewing one architect. This makes it impossible to draw statistically strong conclusions, which raises a validity concern whether the SecoArc framework would be useful in other projects with different requirements (RQ9). To mitigate this risk, we performed a quality attribute workshop (QAW) with six domain experts and a developer to extract variable requirements of PoC. Thereby, the requirements were mainly introduced by another group of people than the architect. Furthermore, we attempted to increase the intensity of the design tasks by asking the architect to design more than one architecture and compare them by using the framework. In addition, we aimed for enhancing the depth of our validation by implementing the major part of the SecoArc framework in a tool. The source code[8] has been always public. We cannot claim the usefulness of the SecoArc framework in other contexts, however, on account of the wide range of software ecosystems captured by our research, we expect that our findings can be valuable to a wide range of companies, which aim at opening their platforms to external parties.

Another potential risk is whether the findings are applicable by the platform providers, who deal with application domains that are not captured by our work. This potential generalizability threat concerns RQ2. We tried to address this group of platform providers by a) consistently providing real-world instances for the architectural guidance of the SecoArc framework, and b) developing the pattern stories. Using this knowledge, the platform provider can reflect on the applicability of the architectural guidance for their specific needs. The architectural patterns are a core part of the findings that reveal the groups of design decisions. During the architectural analysis, the framework refers to the exemplary real-world ecosystems relevant to the results of the analysis. Additionally, the pattern stories uncover the architectural problems, forces, consequences, etc. that can be addressed by applying the groups of decisions.

A further threat to the external validity of the SecoArc framework is whether the framework provides valid knowledge when architectural characteristics of software ecosystems change in the future. This is specially relevant for the SecoArc knowledge base that may require to be updated by future research. To mitigate this threat, we attempted to gain an estimation of *maintainability* of the framework. For the period of this PhD thesis (from 2015-2021), we observed the architecture of software ecosystems in many different ways. We noticed that the knowledge obtained in the early phase of the PhD thesis is still valid, but new knowledge has been added through the years. For example, we developed the variability model in 2017 and ran an update in 2020. The efforts took two weeks time and resulted in the identification of a new variation point and refining an existing variation point to four other variation points (Except for the variabilities of the social and organizational aspects that were not a part of the original work [JZEK17a]). The refinement was related to the concept of openness policies (VP9-VP12). Likewise, the original work related to the architectural commonalities [JPEK16a] included six primary features, which increased by one primary

[8]`https://git.cs.uni-paderborn.de/bahareh`, Last Access: January 10, 2022.

feature in an update in 2020 (Except for the primary features related to the social and organizational aspects that were not a part of the original work). This update took less than a week. While we did not performed any quantified experiment with this regard, the past experiences give us an impression that the amount of time and effort associated with maintaining the framework will remain reasonable.

Reliability

This validity aspect is specially relevant for qualitative research and refers to whether the data and analyses depend on specific researchers [WRH+12]. To ensure that our findings are consistent with what should have been measured regarding our sources of data and research methods, we firstly chose systematic methods for collecting and interpreting the data (e.g., SLR, grounded theory, KJ method, etc.). Details of the steps taken and the data collected were carefully documented and made available in form of three technical reports and the SecoArc specification that concern Chapters 5-8. In the case of the architectural patterns, rich feedbacks are collected from a supervisor in the pattern community, referred to as *shepherd*. In addition, the patterns are presented and critically discussed with the pattern community in a so-called writer's workshop.

9.5 Discussion and Summary

In this chapter, we presented our approach of validating the SecoArc architecture framework developed in this thesis. In general, we discussed two studies.

In our first study, we discussed a case study that we conducted in a real-world project within the scope of on-the-fly computing. In this study, a platform provider uses the knowledge base and the methodical knowledge of the framework to create and analyze models of software ecosystems.

In the second study, we examined existing ecosystems on the basis of our framework to identify whether the architectural characteristics of a diverse range of ecosystems can be captured by the framework. In particular, we performed a comparative analysis of five ecosystems based on our variability model. We identified the instances of the variants and the interdependences between the variants. In the following, we discuss the validation results concerning the subcharacteristics of functional suitability.

Functional Completeness of the SecoArc Framework: Functional completeness is concerned with the number of requirements (R1-R9) that are developed in the SecoArc framework and validated. The SecoArc framework includes the model of primary features that provides an integrated view of the decisions related to the architectural aspects and their interdependencies (R1, R3). Furthermore, the variability model captures the architectural variation points and variants and their interdependencies. It is obtained by examining ecosystems from a diverse range of application domains (R2, R3). The linkage between the decisions, three mostly addressed business objectives, and ecosystem health are clarified in terms of three architectural patterns (R4, R6). Furthermore, the SecoArc modeling framework comprises a modeling language and architectural analysis technique to assess ecosystem health (R5, R6). It provides a design process for applying the patterns in the architecture and using the architectural analysis (R7). Furthermore, the modeling language is implemented in a tool (R8). During the development of the SecoArc framework, models with mathematical foundation such as feature model and OVM are used (R9).

In our first validation study, the constituents of SecoArc were used and validated except for the knowledge of social aspects that were additionally added to the thesis's problem statement, after performing the validation study. As mentioned in Section 8.7, the social aspect is not a part of the SecoArc implementation. A reason is that simulation techniques are the proper option to model this aspect, which is out of the scope of the thesis. With this respect, we discuss future research directions in Section 10.3. Furthermore, our validation is limited in terms of applying the SecoArc design process while one possible way (out of two) is used in the validation. It concerned the manual design of two architectures and later analyzing them. We discussed the other possibility, i.e., applying the architectural patterns, in an illustrative example in Chapter 8. Business objectives identified by platform providers are a topic for future improvement of the SecoArc framework. We discuss the future work in Chapter 10.

Functional Correctness of the SecoArc Framework: To validate the correctness, it should be considered whether the results of using the framework were satisfactory. We validated the correctness by means of an interview with an architect, who used the framework to design and analyze two alternative architectures. While the validation study showed satisfactory results, we identified a limitation that was related to the reusability of the models during the ecosystem evolution. We discussed this limitation as a threat to validity and our the actions to mitigate it in Section 9.4.

Functional Appropriateness of the SecoArc Framework: To validate the appropriateness of the framework, we aimed for measuring the time associated with using the framework in the case study. We reported that the architect installed and set up the working environment independently and in an acceptable amount of time. The preparation time was in total 40 minutes. Two ecosystem architectures are modeled and analyzed within two hours. Three quarters of the time was spent on modeling while one fourth was dedicated to analyzing the architectures.

Furthermore, the second validation study demonstrated that the variability model provides an appropriate abstraction that can capture the architectural variabilities of existing ecosystems. However, we needed to consider two points: in the process of the validation, we identified a new variation point namely store and its variants, and the variants of a variation point (asset).

Chapter 10

Conclusion and Future Work

In this thesis, we developed an architecture framework that facilitates designing healthy software ecosystems. We referred to the framework as *SecoArc* that is aimed for *architecting software ecosystems.* The SecoArc framework spans across the interdisciplinary architectural aspects of software ecosystems, i.e., social, business, application, and infrastructure aspects. It provides the architectural knowledge to design and analyze the ecosystem architecture. We presented a prototypical implementation of the SecoArc framework and used it in a case study within the scope of on-the-fly computing. Furthermore, we conducted a comparative analysis of existing ecosystems based on the knowledge of the framework. This chapter concludes the thesis by first summarizing our key contributions in Section 10.1. Afterward, Section 10.2 discusses how the SecoArc framework fulfills the requirements presented in Section 3.1, followed by Section 10.3, where future research is introduced.

10.1 Summary of Contributions

Designing software ecosystems is a complex and challenging task. To open software platforms to external parties and create a working ecosystem of human actors and software elements around the platforms, platform providers have to face an overwhelming design space of business and technical architectural decisions. In the lack of systematic architectural knowledge, the platform providers have to rely on ad-hoc decision-making that makes the process of creating healthy software ecosystems error-prone and risky.

The goal of this thesis was to facilitate the systematic design of healthy software ecosystems. A solution approach should help to overcome the lack of systematic knowledge in two different ways. On one hand, the main challenge is the lack of knowledge regarding the structure of software ecosystems that is poorly understood until now. In particular, the solution approach should provide knowledge on key architectural design decisions that are commonly applied in software ecosystems as well as architectural variabilities that appear in ecosystems of different application domains. On the other hand, the lack of methodical knowledge hinders the systematic development of ecosystems that should be addressed by the solution approach by providing design methods, processes, and tools.

In this thesis, we developed the SecoArc architectural framework by using different sources of knowledge including a diverse range of existing ecosystems, literature, and human experts to derive, consolidate, and develop *the knowledge base* and *methodical knowledge* of the SecoArc framework by using the design science methodology. The SecoArc knowledge base comprises *architectural commonalities* and *variabilities* of software ecosystems whereas the SecoArc methodical knowledge includes *three architectural patterns* and *a modeling framework* for software ecosystems. In the following, we present a summary of the constituents of the framework mentioned above as well as our validation approach.

Architectural Commonalities of Software Ecosystems

Initially, we developed a conceptual model of architectural commonalities of software ecosystems. For this purpose, we extracted the key architectural knowledge by performing a systematic literature review. On the basis of the extracted knowledge, we developed the conceptual model by identifying and consolidating the architectural commonalities by using the grounded theory methodology. The conceptual model captures the key architectural design decisions and the interrelations between these decisions. In this context, the interrelations reveal the key impact that each of the design decisions on the other decisions has. Our study of architectural commonalities leveraged the knowledge of various literature in different areas to provide an integrated view to the interdisciplinary key architectural decisions that exist in different kinds of software ecosystems such as mobile web service markets, App stores, and markets of cloud computing services.

Architectural Variabilities of Software Ecosystems

We developed a variability model by means of a taxonomy development method based on the architectural knowledge available on the existing ecosystems and literature. For this purpose, we used various sources of information such as Q&A developer forums, annual business reports, and user manuals. Thereby, we extracted the alternatively applied architectural design decisions at the social, business, application, and infrastructure levels. In addition to that, using the knowledge of architectural variabilities, we showed which parts of the ecosystem architecture are usually subject to the variable design decisions. Moreover, our solution approach provides three alternative scenarios describing alternative ways of service provision in software ecosystems that we obtained by interviewing experts. The scenarios reveal the interactions between the human actors that take place once the users submit their requests in the ecosystem until the time that the request is fulfilled, and services are provided.

Architectural Patterns of Software Ecosystems

The methodical knowledge of developing ecosystem architecture is provided in terms of three architectural patterns for software ecosystems. We identified the architectural patterns by using a pattern-mining technique based on the knowledge of 111 existing ecosystems. Each pattern is characterized by a set of architectural design decisions, whereas the design decisions help to achieve certain business objectives and quality attributes of ecosystem health. Furthermore, on the basis of our pattern mining study, we gave insight into when to apply each pattern, and why and how to combine the patterns in the architecture. The conditions to apply a certain pattern are specified based on the organizational characteristics of the platform provider. We also showed how often the patterns are applied together in real-world and what kinds of motivations can trigger combining the patterns.

A Modeling Framework for Software Ecosystems

Our solution approach supports designing software ecosystem models by providing a modeling framework based on the knowledge of the architectural commonalities, variabilities, and patterns developed in this thesis. The modeling framework consists of a modeling language to design the ecosystem architecture, an architectural analysis technique to analyze the suitability of the architecture with respect to the knowledge of the patterns, and design processes to guide platform providers through the design and analysis of the ecosystem architecture. In addition, we developed a modeling workbench that enables the design and analysis activities using our modeling framework. For the implementation of the modeling language, we used EMF and the Sirius framework.

Validation of the SecoArc Framework

We validated our solution approach by means of two studies to show its feasibility in practice. In our first study, the SecoArc framework is actively used by the platform provider within the scope of on-the-fly computing. Thereby, we validated the SecoArc knowledge base as well as the SecoArc methodical knowledge. The architecture team specified the requirements of the future ecosystem around the platform. Based on this knowledge, the architect designed and analyzed two alternative ecosystem architectures using the SecoArc framework.

Our second study aimed at assessing whether the SecoArc framework provides *a suitable abstraction.* Thereby, our study targets the validity of the framework specifically with respect to the existing ecosystems. We examined the ecosystems around the Salesforce, Apple, Amazon AWS, Eclipse, and Amazon Alexa platforms to identify the instances that the ecosystems provide for the variants of the variability model. Furthermore, we used the architectural patterns to describe the ecosystems around the Adobe, Symantec, Citrix, and Cloud Foundry platforms.

10.2 Requirements Revisited

In Section 3.1, we expressed the requirements that an architectural solution needs to fulfill in order to enable the systematic design of software ecosystems. In the following, we discuss how the SecoArc framework fulfills these requirements.

An Architectural Knowledge Base

The requirements of an architectural knowledge base claim that an architectural solution for designing software ecosystems facilitates an integrated view of interdisciplinary key design decisions (R1), the clarification of architectural variation points and their variants across application domains (R2) as well as the identification of decision interdependencies (R3).

The SecoArc framework introduces a conceptual model for the *architectural commonalities of software ecosystems* that includes *the primary features related to the social and organizational, business and management, and application and infrastructure aspects* that were derived from an interdisciplinary set of sources in literature . Therefore, the requirement R1 is fulfilled. Moreover, the framework provides the knowledge of *architectural variation points and variants* by means of *the variability model* and *the service provision scenarios* that both support the specification of variable design decisions. Thereby, the requirement R2 is fulfilled as well. Furthermore, the third requirement (R3) is addressed by the model of primary features and the variability model since parts of these models capture the interrelations between the design decisions.

Methodical Architectural Guidance

The requirements of methodical architectural knowledge claim that a solution enables the design of the ecosystem architecture by providing the following knowledge: Guidance on the linkage between the design decisions and business objectives (R4), modeling support (R5), guidance on the linkage between ecosystem health and design decisions (R6), a design process (R7). In addition, the methodical knowledge should consists of a tool support (R8) while having a formal and precise foundation (R9).

As a part of the SecoArc framework, three architectural patterns were developed. The patterns provide guidance on the linkage between the sets of architectural design decisions and three business objectives that are mostly pursued by the existing platform providers, i.e., *business scalability*, *strategic growth*, and *innovation*. This knowledge can be used during the decision-making. Thereby, the requirement R4 is fulfilled.

Our solution approach introduces the SecoArc modeling framework, which consists of a domain-specific modeling language. The language comprises a domain model for software ecosystems and visual notation. The domain model is based on the knowledge of the architectural commonalities and variabilities. The visual notation enables the design of ecosystem architecture using the concepts of the domain model in a modeling workbench. Thus, the requirement R5 is fulfilled.

The linkage between ecosystem health and architectural design decisions is specified as a part of the architectural patterns while the patterns relate the set of design decisions to the quality attributes of ecosystem health. There, three sets of decisions are related to nine attributes of ecosystem health, i.e., *sustainability*, *robustness*, *profitability*, *niche creation*, *customer satisfaction*, *interoperability*, *modifiability*, *creativity*, and *cost-effectiveness*. Furthermore, the modeling framework consists of the SecoArc architectural analysis technique that facilitates the assessment of ecosystem health based on the knowledge of the architectural patterns. The analysis takes place in the modeling workbench and allows several architectures to be compared with each other. A report of analysis reveals the design decisions that can be made to improve a specific quality attribute. The requirement R6 is thereby satisfied.

By the development of *the SecoArc design process*, the requirement R7 is addressed while the stepwise activities of designing and analyzing software ecosystems are described by means further processes that aim at assisting platform providers to select among the patterns, apply the patterns in the architecture, tailor the architecture, and analyze the suitability of the architectures.

The SecoArc framework has been implemented in a tool (R8) so that the thesis's concepts can be applied in models. Additionally, the tool has been used during the validation of the SecoArc framework to model and analyze ecosystem architectures. To address *maintainability*, the framework has been implemented as a domain-specific language using Eclipse Modeling Framework (EMF). The SecoArc domain model and the visual notation are implemented in separate modules while the domain model is realized in a metamodel and the visual notation is implemented in a Sirius project. This architecture addresses *modularity* and *modifiability* and, thereby, it helps realize future changes in the tool more efficiently. For example, changes in the visual notation can be made without the need to change the domain model, e.g., other visual notations or textual syntax can be developed for the domain model. Furthermore, for testing purposes, the domain model can be changed without the need to change the visual notation. By creating dynamic instances, facilitated by EMF, ecosystem models can be created and analyzed independent of the visual notation. To this end, *testability* is enhanced. Moreover, the SecoArc tool support is an Eclipse plug-in that can be installed on different operating systems, which are supported by the Eclipse platform. Thus, *portability* is addressed.

The SecoArc framework is grounded on models with formal specifications (R9). Specifically, the architectural commonalities are described by means of feature models while the architectural variabilities are expressed using the orthogonal variability model (OVM). Furthermore, our language engineering approach applied in the modeling framework conforms to the meta-object facility (MOF) standard proposed by the Object Management Group (OMG). Thus, the ecosystem models created by using the SecoArc framework conform to the SecoArc domain model. Additionally, the semantics of the decision interdependencies are realized by means of rule-based constraints.

The interview comprised direct questions to the *functional correctness*, *completeness*, and *appropriateness* of the SecoArc framework. The results showed the knowledge base and methodical knowledge of the framework provide a correct and appropriate basis to design software ecosystems. Specifically, designing the architectural variabilities using the grouping feature in the modeling workbench and the visual notation help to achieve a suitable abstraction. Furthermore, the SecoArc architecture analysis technique enhanced correctness by identifying sub-optimal decisions and recommending decisions that can help improve ecosystem health or to achieve a certain business objective. Moreover, completeness is addressed by capturing decisions related to the business, application, and infrastructure aspects in a unified view. In addition, we validated the correctness of the domain knowledge of variabilities with respect to the existing ecosystems.

We addressed *usability* of the knowledge in two ways: Firstly, we provided the SecoArc specification [Jaz21] to enhance *learnability*. The specification contains information on the conceptual parts of the framework and a user manual. The user manual provides guides on the installation activities and how to model and analyze ecosystem models in the SecoArc modeling workbench. Secondly, the SecoArc tool is implemented using EMF. Our goal was to provide our solution concept on the basis of a platform, which is well-known in practice [Ecl21]. The installation of the SecoArc framework follows the typical process of installing Eclipse Plug-ins. In addition, *user interface aesthetics* of the framework is similar to the Eclipse modeling perspective that relieves the users of the framework to work with a new modeling environment.

As described above, the SecoArc framework satisfies the requirements stated in Section 3.1. Therefore, we conclude that the SecoArc framework provides a solution for the thesis's problem stated in Section 1.2 by providing the systematic architectural knowledge, in terms of the knowledge base and the methodical knowledge, to design ecosystem architectures and analyze the architectures with respect to the quality attributes of ecosystem health.

10.3 Future Work

In this thesis, we presented our model-based approach for designing software ecosystems. We showed that our solution facilitates systematic design and analysis of the ecosystem architecture. There are issues that are not covered by our solution, which we address in this section. In the following, we discuss research directions that can be performed to enhance the development of software ecosystems in the future. First, we provide suggestions on how to extend our solution. Afterward, we refer to future research concerning *ecosystem governance* that can further contribute to the systematic development of software ecosystems and it is closely related to the thesis topic.

Extensions of Architectural Commonalities

In our work, we captured the common architectural design decisions of software ecosystems from a strategic point of view that belongs to platform providers. The architectural commonalities span across social, business, and technical aspects to reveal the most important design decisions of platform providers while creating software ecosystems. This knowledge can be extended to a more technical level by providing guidelines for the integration of third-party developments in ecosystems. Dealing with interfaces, where architectural components of third-party developments and software platforms interact with each other, is an error-prone and complex task in software ecosystems [XZZ+20]. In particular, the architectural commonalities indicate the importance of *API documentation* and *API Management*. The usage of APIs and constraints applied in practice remain for future work. With this respect, systematic knowledge can help third-party providers with heterogeneous skills, requirements, and cultural backgrounds to cope with independent software development in large ecosystems. A research direction to structure interfaces could be to use *multiple software product lines (MSPL)* (also known as *product lines of product lines*). MSPL are a set of interrelated product lines that results in the configuration of single customized products. In the context of software ecosystems, service provision is the result of a certain configuration of each third-party development as the inner product line as well as a certain configuration of the overall software, i.e., all third-party developments and the platform. Applying MSPL to manage architectural knowledge of software ecosystems has been explored by Urli et al. [UBC+14]. An improvement to such work would be to provide concepts that are not limited to a specific group of ecosystems by performing more experimental work in this area.

Extensions of Architectural Variabilities

The knowledge of architectural variabilities is aimed at supporting platform providers in dealing with architectural complexity on the basis of the variation points and variants. Dealing with architectural variabilities from the other stakeholders' perspectives, i.e., third-party providers and users, was out of the scope of our work. However, these stakeholders have to face architectural variabilities as well, which is mainly in terms of using the ecosystem resources. In the future, SLA-driven resource management could help handle, and thereby, regulate resource usage in software ecosystems. In the context of third-party providers, SLAs could be used to support the diversity of business models. In the classical discussion on ecosystem openness, entrance barriers are introduced to protect platform providers' intellectual property by imposing certain business models on third-party providers. However, in some situations, ecosystem openness is rather defined in relation to certain groups of third-party providers. For example, in PSE ecosystems, API management is not solely a technical task. Accessibility of the platform

APIs depends on the tiers of partnerships as well. Therefore, the APIs are subsequently scoped and monetized with respect to those tiers. Furthermore, in the context of users, SLAs for resource management would concern the service provision scenarios discussed in this thesis. For commercial or organizational purposes, users might need to be able to decide on resources offered to them in an ecosystem. SLAs for users should consider the possibility to choose among the resources.

In our work, we captured the knowledge of architectural variabilities at design time. During the ecosystem evolution, running ecosystems are subject to changing requirements that can impact the key design decisions made at the design time or may introduce new decisions. Thereby, methods for the adoption of ecosystems are required. *Dynamic software product lines (DSPL)* is a set of approaches to enable variability management at runtime. DSPL could help make architectural descriptions of software ecosystems capable of incorporating new design decisions and ease challenges, which might occur during ecosystem maintenance such as conflict resolution between the design decisions or stakeholders' requirements.

Extensions of Architectural Patterns

Our goal of developing the architectural patterns was to identify the ecosystem architectural designs that are frequently applied in practice. While our validation in the context of on-the-fly computing showed the applicability of this knowledge, the architectural patterns do not claim completeness. In the future, the knowledge of patterns can be applied in different projects with two main objectives. An important objective can be to extend the existing patterns with experimental data collected from inside of companies. Such a study could specially contribute to our understanding of architectural evolution in software ecosystems and governance strategies applied by the companies. For instance, *rate of change* and *stable architecture* are examples of contextual factors that are related to the project dynamics. Such factors need to be monitored and measured in running ecosystems for a reasonable amount of time to understand their impact on the ecosystem architecture. Furthermore, another objective could be to identify further related patterns. Such a study may ultimately result in developing a pattern language for software ecosystems. A pattern language comprises an extensive and coherent set of interrelated patterns that concern different aspects of a family of software systems.

We considered the relations between the three patterns in terms of two forms for ecosystems of ecosystems, i.e. matryoshka and side-by-side ecosystems. While ecosystems of ecosystems were not a primary focus of this thesis, future work could consider this topic, which requires a special focus on social aspects. While this is related to a macro-view to the architecture of software ecosystems, understanding interaction models of human actors would be the prerequisite research. Our suggestion is to use *simulation* for this purpose to model ecosystem entities and their evolution. In the context of matryoshka ecosystems, it would be crucial to know when and how the ecosystems evolve to larger ecosystems while, in the context of side-by-side ecosystems, a central question is *how ecosystem participants move between the ecosystems, e.g., how a community member of an OSE ecosystem become a partner member of a PSE ecosystem.*

Extensions of Modeling Framework

The goal of our modeling framework was to provide modeling support to design ecosystem-specific architectural design decisions. The modeling framework captured decision interdependencies based on our conceptual model of commonalities and variabilities as well as dependency-checking by introducing the SecoArc architectural analysis technique. An enhanced modeling of advanced decision interdependencies such as containment, decomposition, and refinement, as well as logical and temporal relations between the decisions, can be a topic of future research. *Architectural decision models* are formally defined models, e.g., using graph theories, to document design decisions and their dependencies.

While the focus of our work was to make the systematic architectural knowledge available, neither completeness of the ecosystem models designed using the SecoArc modeling framework nor generation of code from these models was an objective of our work. The ecosystem models capture the critical ecosystem-specific architectural design decisions whereas these models can be extended to a comprehensive description of system architecture by adding further components. With this respect, one may use *aspect-oriented programming* in the future, to weave the concepts defined in the ecosystem models into the system architecture. In addition, traceability and closer integration between the variabilities and architectural decision modeling can help close the gap between requirements and architecture of software ecosystems.

Different human actors have different access permissions to the ecosystem resources and accordingly different processes of utilizing the ecosystems. These processes constantly change during the evolution. To this end, the processes should adapt in order to correctly reflect the actor rights. Existing work [KVF$^+$16] can be a starting point towards developing shared and dynamic process models in software ecosystems.

The implementation of the SecoArc framework in our tool support is one of several possibilities to realize the SecoArc framework. The SecoArc modeling language with the visual notations can be improved by extending the notation for the presentation of ecosystems of ecosystems. Another direction would be to introduce a textual modeling language. An example of technologies to realize such a language is *Xtext*. An advantage of a modeling language with textual concrete syntax would be the possibility to add auto-completion and syntax checks to ease the creation of ecosystem models.

Extensions of Validation

In our work, we showed how we extracted and conceptualized the knowledge of existing ecosystems from a diverse range of application domains. By our validation, we were able to demonstrate the feasibility of applying the SecoArc framework with the objective to enhance the systematic design of software ecosystems. Further evaluation of the methodical knowledge and the knowledge base can be performed by creating ecosystems in different application domains in the future. Instantiation of ecosystems and processes of technology realization in a specific application domain are interesting future topics that could contribute to understanding the ways that the ecosystem architecture can be tailored in a certain domain. For instance, fine-grained instances for the variants of the variability model can be defined by using the SecoArc framework. With this respect, we have presented our preliminary ideas for the domain of Internet-of-Things [JS17a], to achieve a more concrete blueprint of the ecosystem architecture for this domain.

Ecosystem Governance

Our work clarified the relations between the ecosystem architecture and critical quality attributes of ecosystem health. Follow-up research may consider ecosystem governance at run-time, i.e., regulating running ecosystems so that they continue to fulfill requirements and deliver value to the stakeholders. Such work could help improve business modeling coupled with technical architecture and establish an improved consistency between the models and code during the ecosystem evolution. As mentioned earlier, the architectural patterns developed in this thesis are related to each other, e.g., a platform provider may grow an ecosystem applying a pattern in a way that it applies a second pattern. Given the dynamics and the high rate of change in the architecture of software ecosystems, these systems can benefit from *situational method engineering* for governance purposes. In this direction, Gottschalk [Got21] considers the high adaptability of business models in running ecosystems in response to uncertain user requirements that are subject to constant changes. In addition, to assess quality aspects of ecosystem behavior at the design time, architectural support concerning dynamic semantics of the ecosystem architecture models plays a central role, which can be considered in the future.

"The research results presented in this PhD thesis contribute to the state of the art concerning software ecosystem design. The thesis provides empirical evidence for the novel architectural knowledge that can be applied to enhance software ecosystems. We extracted and conceptualized the knowledge, and developed modeling and analysis instruments in an architecture framework. We showed that the framework can provide concrete guidelines for improved architectural decision-making, related to the technical but also to the organizational and business aspects of the ecosystem architecture. We executed the modeling and analysis as well as comparison of ecosystem architectures in an integrated tool environment. Our vision of future software ecosystems pertains to the closer integration of the architectural aspects, specially, the social aspect that is twisted with the architecture of software ecosystems. Thereby, modeling and simulation methods and tools are of special importance for future."

References

[AAS09] Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi. The Role of Software Licenses in Open Architecture Ecosystems. In *IWSECO@ ICSR*, 2009.

[ABD13] Katja Andresen, Carsten Brockmann, and Christina Dräger. A Classification of Ecosystems of Enterprise System Providers–An Empirical Analysis. In *2013 46th Hawaii International Conference on System Sciences*, pages 4034–4044. IEEE, 2013.

[AG19] Memoona J. Anwar and Asif Q. Gill. A review of the seven modelling approaches for digital ecosystem architecture. In *2019 IEEE 21st Conference on Business Informatics (CBI)*, volume 1, pages 94–103. IEEE, 2019.

[AGB19] Memoona J. Anwar, Asif Q. Gill, and Ghassan Beydoun. Using Adaptive Enterprise Architecture Framework for Defining the Adaptable Identity Ecosystem Architecture. 2019.

[AHKZ08] Witold Abramowicz, Konstanty Haniewicz, Monika Kaczmarek, and Dominik Zyskowski. E-marketplace for semantic web services. In *Service-Oriented Computing–ICSOC 2008*, pages 271–285. Springer, 2008.

[AJ10] Mohsen Anvaari and Slinger Jansen. Evaluating architectural openness in mobile software platforms. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 85–92. ACM, 2010.

[AOJ17] Carina Alves, Joyce Oliveira, and Slinger Jansen. Software Ecosystems Governance-A Systematic Literature Review and Research Agenda. In *19th Int. Conf. on Enterprise Info. Sys.*, volume 3, pages 26–29, 2017.

[APA14] Jakob Axelsson, Efi Papatheocharous, and Jesper Andersson. Characteristics of software ecosystems for Federated Embedded Systems: A case study. *Information and Software Technology*, 56(11):1457–1475, 2014.

[APB$^+$14] Svetlana Arifulina, Marie Christin Platenius, Steffen Becker, Christian Gerth, Gregor Engels, and Wilhelm Schäfer. Market-optimized service specification and matching. In *Service-Oriented Computing*, pages 543–550. Springer, 2014.

[AS16] Jakob Axelsson and Mats Skoglund. Quality assurance in software ecosystems: A systematic literature mapping and research agenda. *Journal of Systems and Software*, 114:69–81, 2016.

[AT16] Abilio Avila and Orestis Terzidis. Management of Partner Ecosystems in the Enterprise Software Industry. In *IWSECO@ ICIS*, pages 39–55. Citeseer, 2016.

[Aza15] Shams Abubakar Azad. *Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations.* PhD thesis, Concordia University, 2015.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[BB10a] Jan Bosch and Petra Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83(1):67–76, 2010.

[BB10b] Jan Bosch and Petra M. Bosch-Sijtsema. Softwares product lines, global development and ecosystems: Collaboration in software engineering. In *Collaborative Software Engineering*, pages 77–92. Springer, 2010.

[BB14] Jan Bosch and Petra Bosch-Sijtsema. ESAO: A holistic ecosystem-driven analysis model. In *ICSOB14*, pages 179–193. Springer, 2014.

[BCH15] Jan Bosch, Rafael Capilla, and Rich Hilliard. Trends in Systems and Software Variability [Guest editors' introduction]. *IEEE Software*, 32(3):44–51, 2015.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, 2003.

[BCPR09] David F. Bacon, Yiling Chen, David Parkes, and Malvika Rao. A market-based approach to software evolution. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 973–980. ACM, 2009.

[BdA$^+$13] Olavo Barbosa, Rodrigo Pereira dos Santos, Carina Alves, Claudia Werner, and Slinger Jansen. A systematic mapping study on software ecosystems from a three-dimensional perspective. In *Software Ecosystems.* Edward Elgar Publishing, 2013.

[BDS19] Fabian Burmeister, Paul Drews, and Ingrid Schirmer. An ecosystem architecture meta-model for supporting ultra-large scale digital transformations. 2019.

[BEL$^+$03] Mario R. Barbacci, Robert J. Ellison, Anthony Lattanze, Judith Stafford, Charles B. Weinstock, and William Wood. Quality attribute workshops. Technical Report CMU/SEI-2003-TR-016, 2003.

[BHH15] Alexander Benlian, Daniel Hilkert, and Thomas Hess. How open is this platform? the meaning and measurement of platform openness from the complementors' perspective. *Journal of Information Technology*, 30(3):209–228, 2015.

[BHS07] Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John wiley & sons, 2007.

[BHSM13] Amel Ben Hadj Salem Mhamdia. Performance measurement practices in software ecosystem. *Int. J. of Productivity and Performance Management*, 62(5):514–533, 2013.

[BJ12] Alfred Baars and Slinger Jansen. A framework for software ecosystem governance. In *International Conference of Software Business*, pages 168–180. Springer, 2012.

[BLP05] Stan Buhne, Kim Lauenroth, and Klaus Pohl. Modelling requirements variability across product lines. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 41–50. IEEE, 2005.

[Bos04] Jan Bosch. Software architecture: The next step. In *European Workshop on Software Architecture*, pages 194–199. Springer, 2004.

[Bos09] Jan Bosch. From software product lines to software ecosystems. In *Int. Conf. on Soft. Product Line*, pages 111–119. CMU, 2009.

[Bos10] Jan Bosch. Architecture challenges for software ecosystems. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 93–95. ACM, 2010.

[Bou10] Kevin Boudreau. Open platform strategies and innovation: Granting access vs. devolving control. *Management science*, 56(10):1849–1872, 2010.

[BPT$^+$14] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wasowski, and Steven She. Variability mechanisms in software ecosystems. *Info. and Soft. Tech.*, 56(11):1520–1535, 2014.

[BRC10] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Algorithms and Architectures for Parallel Processing*, pages 13–31. Springer, 2010.

[BRN$^+$13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*, page 7. ACM, 2013.

[BUI16] Build an in-house enterprise app store without breaking the budget. http://searchcloudapplications.techtarget.com/answer/Build-an-in-house-enterprise-app-store-without-breaking-the-budget, 2016.

[Car12] Octavian Carare. The impact of bestseller rank on demand: Evidence from the app market*. *International Economic Review*, 53(3):717–742, 2012.

[CF07] Davide Cerri and Alfonso Fuggetta. Open standards, open formats, and open source. *Journal of systems and software*, 80(11):1930–1937, 2007.

[CFHW12] Marco Ceccagnoli, Chris Forman, Peng Huang, and D. J. Wu. Cocreation of value in a platform ecosystem! The case of enterprise software. *MIS quarterly*, pages 263–290, 2012.

[CG12] Rishi Chandy and Haijie Gu. Identifying spam in the iOS app store. In *Proceedings of the 2nd Joint WICOW/AIRWeb Workshop on Web Quality*, pages 56–59. ACM, 2012.

[CH10] Marcelo Cataldo and James D. Herbsleb. Architecting in software ecosystems: Interface translucence as an enabler for scalable collaboration. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 65–72. ACM, 2010.

[CHKM14] Henrik Christensen, Klaus Marius Hansen, Morten Kyng, and Konstantinos Manikas. Analysis and design of software ecosystem architectures–Towards the 4S telemedicine ecosystem. *Info. and Soft. Tech.*, 56(11):1476–1492, 2014.

[CJP^{+}11] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database-as-a-service for the cloud. 2011.

[CJT^{+}16] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software*, 116:191–205, 2016.

[CLH^{+}14] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. AR-Miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, pages 767–778. ACM, 2014.

[CM98] Miriam Catterall and Pauline Maclaran. *The Death of Competition: Leadership and Strategy in the Age of Business Ecosystems*, volume 40. 1998.

[CR94] Victor R. Basili1 Gianluigi Caldiera and H. Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[CSS+13] Gabriella Costa, Felyppe Silva, Rodrigo Santos, Cláudia Werner, and Toacy Oliveira. From applications to a software ecosystem platform: An exploratory study. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, pages 9–16. ACM, 2013.

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[DGSB10] Deepak Dhungana, Iris Groher, Elisabeth Schludermann, and Stefan Biffl. Software ecosystems vs. natural ecosystems: Learning from the ingenious mind of nature. In *European Conference on Software Architecture Workshops*, pages 96–102. ACM, 2010.

[DHS19] Julian Dörndorfer, Florian Hopfensperger, and Christian Seel. The SenSoMod-Modeler–A Model-Driven Architecture Approach for Mobile Context-Aware Business Applications. In *International Conference on Advanced Information Systems Engineering*, pages 75–86. Springer, 2019.

[DLCA17] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. The evolution of technical debt in the apache ecosystem. In *ECSA17*, pages 51–66. Springer, 2017.

[DMA+10] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl. The home needs an operating system (and an app store). In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 18. ACM, 2010.

[DMHP14] Anh Nguyen Duc, Audris Mockus, Randy Hackbarth, and John Palframan. Forking and coordination in multi-platform development: A case study. In *ACM/IEEE Int. Symp. on Empirical Soft. Eng. and Measurement*, page 59. ACM, 2014.

[DS14] Paul Drews and Ingrid Schirmer. From enterprise architecture to business ecosystem architecture: Stages and challenges for extending architectures beyond organizational boundaries. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, pages 13–22. IEEE, 2014.

[DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[EB12] Ulrik Eklund and Jan Bosch. Using architecture for multiple levels of access to an ecosystem platform. In *Proceedings of the 8th International*

ACM SIGSOFT Conference on Quality of Software Architectures, pages 143–148. ACM, 2012.

[EB14] Ulrik Eklund and Jan Bosch. Architecture for embedded open software ecosystems. *Journal of Systems and Software*, 92:128–142, 2014.

[Ecl21] Eclipse: Still the Best IDE! - DZone Integration. https://dzone.com/articles/eclipse-still-the-best-ide, 2021.

[EJM$^+$14] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104. ACM, 2014.

[Fau19] Edward Faulkner. Growing a Healthy Software Ecosystem. https://medium.com/cardstack/growing-a-healthy-software-ecosystem-746c3f7eefb1, October 2019.

[FHT$^+$12] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, RaüL Sirvent, Jordi Guitart, Rosa M. Badia, Karim Djemame, et al. OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012.

[FLL$^+$13] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1276–1284. ACM, 2013.

[FMM15] Mirco Franzago, Ivano Malavolta, and Henry Muccini. Stakeholders, viewpoints and languages of a modelling framework for the design and development of data-intensive mobile apps. *arXiv preprint arXiv:1502.04014*, 2015.

[Fou18] Cloud Foundry. How to Create a Pivotal Cloud Foundry Support Ticket Successfully. https://community.pivotal.io/s/article/How-to-Create-a-Cloud-Foundry-Support-Ticket-Successfully?language=en_US, 2018.

[Fow97] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1997.

[Fow12] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[G$^+$13] OBJECT MANAGEMENT GROUP et al. Object Management Group (OMG). https://www.omg.org/, 2013.

[Gab02] Richard P. Gabriel. *Writers' Workshops & the Work of Making Things: Patterns, Poetry...* Addison-Wesley Boston, 2002.

[GAM$^+$17] María Gómez, Bram Adams, Walid Maalej, Martin Monperrus, and Romain Rouvoy. App Store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems. *IEEE Software*, 34(2):81–89, 2017.

[Gar] Definition of Enterprise Architecture (EA) - Gartner Information Technology Glossary. https://www.gartner.com/en/information-technology/glossary/enterprise-architecture-ea.

[GAT13] Matthias Galster, Paris Avgeriou, and Dan Tofan. Constraints for the design of variability-intensive service-oriented reference architectures–An industrial case study. *Information and Software Technology*, 55(2):428–441, 2013.

[GCCJ11] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, pages 21–26. ACM, 2011.

[GHJ$^+$95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Elements of Reusable Object-Oriented Software*, volume 99. Addison-Wesley Reading, Massachusetts, 1995.

[Git21] The Legal Side of Open Source. https://opensource.guide/legal/, 2021.

[GMBV12] Michael Goul, Olivera Marjanovic, Susan Baxley, and Karen Vizecky. Managing the Enterprise Business Intelligence app store: Sentiment analysis supported requirements engineering. In *System Science (HICSS), 2012 45th Hawaii International Conference On*, pages 4168–4177. IEEE, 2012.

[Gol10] David Goldman. Microsoft has a chicken and egg problem with Windows Phone 7 - Nov. 8, 2010. https://money.cnn.com/2010/11/08/technology/windows_phone_7/index.htm, 2010.

[Got21] Sebastian Gottschalk. Situation-specific Development of Business Models for Services in Software Ecosystems. In *Advanced Software Engineering*, 2021.

[GPVS05] Giancarlo Guizzardi, Luís Ferreira Pires, and Marten Van Sinderen. An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 691–705. Springer, 2005.

[Gro21] The Open Group. Architectural Patterns. http://www.opengroup.org/public/arch/p4/patterns/patterns.htm, 2021.

[GS67] Barney G. Glaser and Anselm L. Strauss. The discovery of grouded theory. *Chicago (US): Aldine*, 1967.

[GSE21] Sanket Kumar Gupta, Bahar Schwichtenberg, and Gregor Engels. A Reference Architecture for Enhanced Design of Software Ecosystems. In *International Symposium on Business Modeling and Software Design*, pages 59–77. Springer, 2021.

[GWB10] Vania Goncalves, Nils Walravens, and Pieter Ballon. "How about an App Store?" Enablers and Constraints in Platform Strategies for Mobile Network Operators. In *Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR), 2010 Ninth International Conference On*, pages 66–73. IEEE, 2010.

[HA05] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *METRICS05*, pages 10–pp. IEEE, 2005.

[HA12] Hatem Hamad and Mahmoud Al-Hoby. Managing intrusion detection as a service in cloud networks. *International Journal of Computer Applications*, 41(1), 2012.

[Han12] Geir K. Hanssen. A longitudinal case study of an emerging software ecosystem: Implications for practice and theory. *Journal of Systems and Software*, 85(7):1455–1466, 2012.

[Has18] Todd Haselton. Departing Windows chief Terry Myerson explains why Microsoft failed in smartphones. https://www.cnbc.com/2018/03/29/why-microsoft-failed-in-phones.html, March 2018.

[HAZ07] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *IEEE software*, 24(4):38–45, 2007.

[HGS13] Ralf Herbrich, Thore Graepel, and David Shaw. Reputation system, February 2013.

[HHYH09] Seung-Min Han, Mohammad Mehedi Hassan, Chang-Woo Yoon, and Eui-Nam Huh. Efficient service recommendation system for cloud computing market. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 839–845. ACM, 2009.

[HJZ12] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: MSR for app stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 108–111. IEEE Press, 2012.

[HO11] Adrian Holzer and Jan Ondrus. Mobile application market: A developer's perspective. *Telematics and informatics*, 28(1):22–31, 2011.

[HOZZ16] Gregor Hohpe, Ipek Ozkaya, Uwe Zdun, and Olaf Zimmermann. The software architect's role in the digital age. *IEEE Software*, 33(6):30–39, 2016.

[HS05] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[HSN$^+$15] Sami Hyrynsalmi, Marko Seppänen, Tiina Nokkala, Arho Suominen, and Antero Järvi. Wealthy, Healthy and/or Happy—What does 'ecosystem health'stand for? In *International Conference of Software Business*, pages 272–287. Springer, 2015.

[IKC09] Wassim Itani, Ayman Kayssi, and Ali Chehab. Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures. In *Dependable, Autonomic and Secure Computing, 2009. DASC'09. Eighth IEEE International Conference On*, pages 711–716. IEEE, 2009.

[IL02] Marco Iansiti and Roy Levien. Keystones and dominators: Framing the operational dynamics of business ecosystems. *The operational dynamics of business ecosystems*, 2002.

[IL04a] Marco Iansiti and Roy Levien. *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability.* Harvard Business Press, 2004.

[IL04b] Marco Iansiti and Roy Levien. *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability.* Harvard Business Press, 2004.

[ISO11] ISO/IEC/IEEE Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46, December 2011.

[ISO12] ISO/IEC 25010:2011. https://www.iso.org/standard/35733.html, July 2012.

[IvJ11] Andrei Idu, Tommy van de Zande, and Slinger Jansen. Multi-homing in the apple ecosystem: Why and how developers target multiple apple app stores. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, pages 122–128. ACM, 2011.

[IYM17] Takashi Iba, Ayaka Yoshikawa, and Konomi Munakata. Philosophy and methodology of clustering in pattern mining: Japanese anthropologist Jiro Kawakita's KJ method. In *Proceedings of the 24th Conference on Pattern Languages of Programs*, pages 1–11, 2017.

[Jan14] Slinger Jansen. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*, 56(11):1508–1519, 2014.

[Jan20] Slinger Jansen. A focus area maturity model for software ecosystem governance. *Information and Software Technology*, 118:106219, 2020.

[Jär13] Kati Järvi. Ecosystem architecture design: Endogenous and exogenous structural properties. 2013.

[Jaz16] Bahar Jazayeri. Architectural management of on-the-fly computing markets. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, page 42. ACM, 2016.

[Jaz21] SecoArc. sfb901.uni-paderborn.de/secoarc, April 2021.

[JB05] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120. IEEE, 2005.

[JB13] Slinger Jansen and Ewoud Bloemendal. Defining app stores: The role of curated marketplaces in software ecosystems. In *Software Business. From Physical Products to Software Services and Solutions*, pages 195–206. Springer, 2013.

[JBL12] Euy-Young Jung, Chulwoo Baek, and Jeong-Dong Lee. Product survival analysis for the App Store. *Marketing Letters*, 23(4):929–941, 2012.

[JBSL12] Slinger Jansen, Sjaak Brinkkemper, Jurriaan Souer, and Lutzen Luinenburg. Shades of gray: Opening up a software producing organization with the open software enterprise model. *Journal of Systems and Software*, 85(7):1495–1510, 2012.

[JC13] Slinger Jansen and Michael A. Cusumano. Defining software ecosystems: A survey of software platforms and business network governance. In *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, volume 13, 2013.

[JFB09] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. A sense of community: A research agenda for software ecosystems. In *International Conference on Software Engineering (ICSE)-Companion Volume*, pages 187–190. IEEE, 2009.

[JPEK16a] Bahar Jazayeri, Marie C. Platenius, Gregor Engels, and Dennis Kundisch. Features of IT Service Markets: A Systematic Literature Review. In *ICSOC16*, pages 301–316. Springer, 2016.

[JPEK16b] Bahar Jazayeri, Marie Christin Platenius, Gregor Engels, and Dennis Kundisch. IT Service Markets: A Systematic Literature Review – Technical Report. Technical Report tr-ri-16-350, Heinz Nixdorf Institute, University of Paderborn, May 2016.

[JS17a] Bahar Jazayeri and Simon Schwichtenberg. On-The-Fly Computing Meets IoT Markets – Towards a Reference Architecture. In *Int. Conf. on Soft. Arch. Companion Volume*, pages 120–127. IEEE, 2017.

[JS17b] Bahar Jazayeri and Simon Schwichtenberg. On the Necessity of an Architecture Framework for On-The-Fly Computing. *Softwaretechnik-Trends: Vol. 37, No. 2*, 2017.

[JSK$^+$20] Bahar Jazayeri, Simon Schwichtenberg, Jochen Küster, Olaf Zimmermann, and Gregor Engels. Modeling and Analyzing Architectural Diversity of Open Platforms. In *CAiSE20*, pages 36–53. Springer, 2020.

[JSW11] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. The onion patch: Migration in open source ecosystems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 70–80, 2011.

[JZE$^+$18] Bahar Jazayeri, Olaf Zimmermann, Gregor Engels, Jochen Küster, Dennis Kundisch, and Daniel Szopinski. Design Options of Store-Oriented Software Ecosystems: An Investigation of Business Decisions. In *Int. Symp. on Business Modeling and Soft. Design*, pages 573–588. Springer, 2018.

[JZEK17a] Bahar Jazayeri, Olaf Zimmermann, Gregor Engels, and Dennis Kundisch. A Variability Model for Store-Oriented Software Ecosystems: An Enterprise Perspective. In *ICSOC17*, pages 573–588. Springer, 2017.

[JZEK17b] Bahar Jazayeri, Olaf Zimmermann, Gregor Engels, and Dennis Kundisch. A Variability Model for Store-oriented Software Ecosystems: An Enterprise Perspective (Supplementary Material). Technical report, Technical report, 2017.

[JZK$^+$18] Bahar Jazayeri, Olaf Zimmermann, Jochen Küster, Gregor Engels, Daniel Szopinski, and Dennis Kundisch. Patterns of Store-oriented Software Ecosystems: Detection, Classification, and Analysis of Design Options. In *Lathin American Conference on Pattern Languages of Programs*. ACM, 2018.

[JZKE18] Bahar Jazayeri, Olaf Zimmermann, Jochen Küster, and Gregor Engels. Patterns of Store-oriented Software Ecosystems: Detection, Classification, and Analysis of Design Options. Dataset . https://www.overleaf.com/read/njqzqhsmvctk. Technical report, 2018.

[Kaw91] Jiro Kawakita. The original KJ method. *Tokyo: Kawakita Research Institute*, 5, 1991.

[KBB$^+$09] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, 51(1):7–15, 2009.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical report, Keele University and Durham University, 2007.

[KC10] Rick Kazman and Hong-Mei Chen. The metropolis model and its implications for the engineering of software ecosystems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 187–190, 2010.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[KDKB14] Eric Knauss, Daniela Damian, Alessia Knauss, and Arber Borici. Openness and requirements: Opportunities and tradeoffs in software ecosystems. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 213–222. IEEE, 2014.

[KHMA12] Terhi Kilamo, Imed Hammouda, Tommi Mikkonen, and Timo Aaltonen. From proprietary to open source—Growing an open source ecosystem. *Journal of Systems and Software*, 85(7):1467–1478, 2012.

[KJ11] Jaap Kabbedijk and Slinger Jansen. Steering insight: An exploration of the ruby software ecosystem. In *International Conference of Software Business*, pages 44–55. Springer, 2011.

[KKLK13] Jognwoo Kim, Sanggil Kang, Yujin Lim, and Hak-Man Kim. Recommendation algorithm of the app store by using semantic relations between apps. *The Journal of Supercomputing*, 65(1):16–26, 2013.

[KLVV06] Philippe Kruchten, Patricia Lago, and Hans Van Vliet. Building up and reasoning about architectural knowledge. In *International Conference on the Quality of Software Architectures*, pages 43–58. Springer, 2006.

[KPKL14] Jieun Kim, Yongtae Park, Chulhyun Kim, and Hakyeon Lee. Mobile application service networks: Apple's App Store. *Service Business*, 8(1):1–27, 2014.

[Kru04] Philippe Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen Workshop on Software Variability*, pages 54–61. Citeseer, 2004.

[Kru13] Philippe Kruchten. Contextualizing agile software development. *Journal of Software: Evolution and Process*, 25(4):351–361, 2013.

[KVF+16] Jochen Küster, Hagen Völzer, Cédric Favre, Moisés Castelo Branco, and Krzysztof Czarnecki. Supporting different process views through a shared process model. *Software & Systems Modeling*, 15(4):1207–1233, 2016.

[KW08] Dominik Kuropka and Mathias Weske. Implementing a semantic service provision platform. *Wirtschaftsinformatik*, 50(1):16–24, 2008.

[Kwa17] Gabe Kwakyi. How to Keep Spammers Out of the App Store. https://incipiagabe.medium.com/how-to-keep-spammers-out-of-the-app-store-9bbed7b4e609, February 2017.

[KZ10] Patrick Könemann and Olaf Zimmermann. Linking design decisions to design models in model-based software development. In *European Conference on Software Architecture*, pages 246–262. Springer, 2010.

[LB13] Soo Ling Lim and Peter J. Bentley. Investigating app store ranking algorithms using a simulation of mobile app ecosystems. In *Evolutionary Computation (CEC), 2013 IEEE Congress On*, pages 2672–2679. IEEE, 2013.

[LBWK18] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163, 2018.

[LFZD09] Geng Lin, David Fu, Jinzy Zhu, and Glenn Dasmalchi. Cloud computing: IT as a service. *IT Professional Magazine*, 11(2):10, 2009.

[LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, pages 62–77. Springer, 2002.

[LMD$^+$13] Samad Lotia, Jason Montojo, Yue Dong, Gary D. Bader, and Alexander R. Pico. Cytoscape app store. *Bioinformatics*, page btt138, 2013.

[LR11] Gunwoong Lee and T. S. Raghu. Product Portfolio and Mobile Apps Success: Evidence from App Store Market. In *AMCIS*, 2011.

[LS20] Maxime Lamothe and Weiyi Shang. When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds. In *Proceedings. 42nd International Conference on Software Engineering, ICSE*, volume 2020, 2020.

[LYW$^+$19] Zhifang Liao, Mengjie Yi, Yan Wang, Shengzong Liu, Hui Liu, Yan Zhang, and Yun Zhou. Healthy or not: A way to predict ecosystem health in Github. *Symmetry*, 11(2):144, 2019.

[Man15] Konstantinos Manikas. *Analyzing, Modelling, and Designing Software Ecosystems*. PhD thesis, University of Copenhagen Denmark, 2015.

[Man16] Konstantinos Manikas. Revisiting software ecosystems research: A longitudinal literature study. *J. Sys. and Soft.*, 117:84–103, 2016.

[MG11] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011.

[MH13] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems–a systematic literature review. *J. Sys. and Soft.*, 86(5):1294–1306, 2013.

[MHJ$^+$15] William Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The app sampling problem for app store mining. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference On*, pages 123–133. IEEE, 2015.

[MHT16] Konstantinos Manikas, Mervi Hämäläinen, and Pasi Tyrväinen. Designing, Developing, and Implementing Software Ecosystems: Towards a Step-wise Guide. In *CEUR Workshop Proceedings; 1808*. Editors; Sun SITE Central Europe, 2016.

[MI11] Kazunori Mizushima and Yasuo Ikawa. A structure of co-creation in an open source software ecosystem: A case study of the eclipse community. In *Technology Management in the Energy Smart World (PICMET)*, pages 1–8. IEEE, 2011.

[MKM11] Roland M. Müller, Bjorn Kijl, and Josef KJ Martens. A comparison of inter-organizational business models of mobile app stores: There is more than open vs. closed. *Journal of theoretical and applied electronic commerce research*, 6(2):63–76, 2011.

[MPH$^+$07] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 243–253. IEEE, 2007.

[MRK13] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE, 2013.

[MS94] Daniel L. Moody and Graeme G. Shanks. What makes a good data model? evaluating the quality of entity relationship models. In *International Conference on Conceptual Modeling*, pages 94–111. Springer, 1994.

[MS20] Poonacha K. Medappa and Shirish C. Srivastava. Ideological shifts in open source orchestration: Examining the influence of licence choice and organisational participation on open source project outcomes. *European Journal of Information Systems*, 29(5):500–520, 2020.

[MSJ$^+$16] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2016.

[MSJ$^+$17] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2017.

[MSS$^+$13] Tobias Mueller, Denis Schuldt, Birgit Sewald, Marcel Morisse, and Jurate Petrikina. Towards inter-organizational enterprise architecture management-applicability of TOGAF 9.1 for network organizations. 2013.

[MVG$^+$14] Andreas Menychtas, Jürgen Vogel, Andrea Giessmann, Anna Gatzioura, Sergio Garcia Gomez, Vrettos Moulos, Frederic Junker, Mathias Müller, Dimosthenis Kyriazis, Katarina Stanoevska-Slabeva, et al. 4caast marketplace: An advanced business environment for trading cloud services. *Future Generation Computer Systems*, 41:104–120, 2014.

[NSV19] Jarkko Nurmi, Ville Seppänen, and Meri Katariina Valtonen. Ecosystem Architecture Management in the Public Sector–From Problems to Solutions. *Complex Systems Informatics and Modeling Quarterly*, (19):1–18, 2019.

[NVM13] Robert C. Nickerson, Upkar Varshney, and Jan Muntermann. A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, 22(3):336–359, 2013.

[OMG16] OMG. Meta Object Facility (MOF) Core. http://www.omg.org/spec/MOF/2.5.1, 2016.

[OMG17] OMG. Unified Modeling Language™ (UML®) Version 2.5. https://www.omg.org/spec/UML, 2017.

[ONE12] 5 Mistakes to avoid when deploying an enterprise App Store. http://www.cio.com/article/2394413, 2012.

[PAS12] People Also Search For (PASF): Search Volume For Google PASF. https://keywordseverywhere.com/people-also-search-for.html, July 2012.

[PBv05] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

[PO09] Radu Prodan and Simon Ostermann. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In *Grid Computing, 2009 10th IEEE/ACM International Conference On*, pages 17–25. IEEE, 2009.

[PRS09] Pankesh Patel, Ajith H. Ranabahu, and Amit P. Sheth. Service level agreement in cloud computing. 2009.

[PSS$^+$16] Konstantinos Plakidas, Srdjan Stevanetic, Daniel Schall, Tudor B. Ionescu, and Uwe Zdun. How do software ecosystems evolve? a quantitative assessment of the r ecosystem. In *International Systems and Software Product Line Conference*, pages 89–98. ACM, 2016.

[Pub21] Public API. https://microservice-api-patterns.org/patterns/foundation/PublicAPI, 2021.

[RA12] Juthasit Rohitratana and Jörn Altmann. Impact of pricing schemes on a market for software-as-a-service and perpetual software. *Future Generation Computer Systems*, 28(8):1328–1339, 2012.

[Ray97] Raymond Scupin. The KJ method: A technique for analyzing data derived from Japanese ethnology. *Human organization*, pages 233–237, 1997.

[REV19] REVaMP^2 Project. http://www.revamp2-project.eu/, 2019.

[RKZF00] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[RL11] Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 904–907. ACM, 2011.

[RLR12] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation? the case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[Roh19] Benjamin Rohe. Why corporate innovation is extremely hard, and how to navigate it? https://www.linkedin.com/pulse/why-corporate-innovation-extremely-hard-how-navigate-benjamin-roh%C3%A9, 2019.

[ROP16] Moisés Rodríguez, Jesús Ramón Oviedo, and Mario Piattini. Evaluation of Software Product Functional Suitability: A Case Study. *Software Quality Professional*, 18(3), 2016.

[RS12] Philip Holst Riis and Petra Schubert. Upgrading to a new version of an erp system: A multilevel analysis of influencing factors in a software ecosystem. In *2012 45th Hawaii International Conference on System Sciences*, pages 4709–4718. IEEE, 2012.

[RW12] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley, 2012.

[SA12] Walt Scacchi and Thomas A. Alspaugh. Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software*, 85(7):1479–1494, 2012.

[Sau11] Stefan Sauer. *Systematic Development of Model-Based Software Engineering Methods.* PhD thesis, University of Paderborn, 2011.

[SBH14] Anisa Stefi, Matthias Berger, and Thomas Hess. What Influences Platform Provider's Degree of Openness?–Measuring and Analyzing the Degree of Platform Openness. In *International Conference of Software Business*, pages 258–272. Springer, 2014.

[SBL$^+$19] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect API compatibility issues in Android: An empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, 2019.

[SC94] Anselm Strauss and Juliet Corbin. Grounded theory methodology. *Handbook of qualitative research*, pages 273–285, 1994.

[SC06] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE software*, 23(2):31–39, 2006.

[Sca07] Walt Scacchi. Free/open source software development: Recent research results and methods. *Advances in Computers*, 69:243–295, 2007.

[Sch04] Jaap Schekkerman. *How to Survive in the Jungle of Enterprise Architecture Frameworks: Creating Or Choosing an Enterprise Architecture Framework*. Trafford Publishing, 2004.

[SDY15] Mahsa H. Sadi, Jiaying Dai, and Eric Yu. Designing software ecosystems: How to develop sustainable collaborations? In *International Conference on Advanced Information Systems Engineering Workshops*, pages 161–173. Springer, 2015.

[SE20] Bahar Schwichtenberg and Gregor Engels. SecoArc: A Framework for Architecting Healthy Software Ecosystems. In *European Conference on Software Architecture Companion Volume*, pages 95–106. Springer, 2020.

[Sea99] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999.

[SEL14] Klaus-Benedikt Schultis, Christoph Elsner, and Daniel Lohmann. Architecture challenges for internal software ecosystems: A large-scale industry case study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 542–552, 2014.

[Ses07] Roger Sessions. Comparison of the Top Four Enterprise Architecture Methodologies. Technical report, May 2007.

[SFB20] Collaborative research Center 901 - On_The-Fly Computing. https://sfb901.uni-paderborn.de/, 2020.

[SH92] S. H. Spewak and S. C. Hill. Enterprise Architecture. *Planning„ Princeton, NJ, USA,: John Wiley, and Sons, inc*, 1992.

[SHMS17] Marko Seppänen, Sami Hyrynsalmi, Konstantinos Manikas, and Arho Suominen. Yet another ecosystem literature review: 10+ 1 research communities. In *2017 IEEE European Technology and Engineering Management Summit (E-TEMS)*, pages 1–8. IEEE, 2017.

[SK11] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[SKKR00] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 158–167. ACM, 2000.

[SKTI16] Alice Sasabe, Tomoki Kaneko, Kaho Takahashi, and Takashi Iba. Pattern mining patterns: A search for the seeds of patterns. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, page 12. The Hillside Group, 2016.

[SMB18] What is SMB? - Gartner Defines Small and Midsize Businesses. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, Last access: January 2018.

[SO11] Sebastian Schlauderer and Sven Overhage. How perfect are markets for software services? an economic perspective on market deficiencies and desirable market features. In *ECIS*, 2011.

[Sou18] Lamia Soussi. Health vulnerabilities in software ecosystems: Five cases of dying platforms. Master's thesis, 2018.

[Spa21] Sparx Systems. https://sparxsystems.com/, 2021.

[SY15] Mahsa H. Sadi and Eric Yu. Designing software ecosystems: How can modeling techniques help? In *International Conference on Enterprise, Business-Process and Information Systems Modeling*, pages 360–375. Springer, 2015.

[SY17a] Mahsa H. Sadi and Eric Yu. Accommodating Openness Requirements in Software Platforms: A Goal-Oriented Approach. In *CAiSE17*, pages 44–59. Springer, 2017.

[SY17b] Mahsa H. Sadi and Eric Yu. Modeling and analyzing openness trade-offs in software platforms: A goal-oriented approach. In *REFSQ17*, pages 33–49. Springer, 2017.

[SZZ$^+$18] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–16, 2018.

[Tee10] David J. Teece. Business models, business strategy and innovation. *Long range planning*, 43(2):172–194, 2010.

[The11] The TOGAF® Standard, from The Open Group | The Open Group. http://www.opengroup.org/subjectareas/enterprise/togaf, 2011.

[TRG15] Jose Teixeira, Gregorio Robles, and Jesús M. González-Barahona. Lessons learned from applying social network analysis on an industrial Free/Libre/Open Source Software ecosystem. *Journal of Internet Services and Applications*, 6(1):1–27, 2015.

[TSB10] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-oriented cloud computing architecture. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference On*, pages 684–689. IEEE, 2010.

[TTP11] Virpi Kristiina Tuunainen, Tuure Tuunanen, and Jouni Piispanen. Mobile service platforms: Comparing nokia ovi and apple app store with the iisin model. In *Mobile Business (ICMB), 2011 Tenth International Conference On*, pages 74–83. IEEE, 2011.

[UBC$^+$14] Simon Urli, Mireille Blay-Fornarino, Philippe Collet, Sébastien Mosser, and Michel Riveill. Managing a software ecosystem using a multiple software product line: A case study on digital signage systems. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 344–351. IEEE, 2014.

[VAKJP11] Joey Van Angeren, Jaap Kabbedijk, Slinger Jansen, and Karl Michael Popp. A Survey of Associate Models used within Large Software Ecosystems. In *Int. Work. on Soft. Ecosystems*, pages 27–39. CEUR-WS, 2011.

[Val13] George Valen. Requirements negotiation model: A social oriented approach for software ecosystems evolution. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 393–396. IEEE, 2013.

[VBD$^+$13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* dslbook. org, 2013.

[VDBJL10] Ivo Van Den Berk, Slinger Jansen, and Lútzen Luinenburg. Software ecosystems: A software ecosystem strategy assessment model. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 127–134. ACM, 2010.

[VDGS18] Aparna Vegendla, Anh Nguyen Duc, Shang Gao, and Guttorm Sindre. A systematic mapping study on requirements engineering in software ecosystems. *Journal of Information Technology Research (JITR)*, 11(1):49–69, 2018.

[VK11] Martti Viljainen and Marjo Kauppinen. Software ecosystems: A set of management practices for platform integrators in the telecom industry. In *International Conference of Software Business*, pages 32–43. Springer, 2011.

[VSB$^+$13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2013.

[VWWH$^+$10] Jack Van't Wout, Maarten Waage, Herman Hartman, Max Stahlecker, and Aaldert Hofman. *The Integrated Architecture Framework Explained: Why, What, How.* Springer Science & Business Media, 2010.

[WB10] Yi Wei and M. Brian Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing*, 14(6):72, 2010.

[WB15] Eoin Woods and Rabih Bashroush. Modelling large-scale information systems using adls–an industrial experience report. *J. Sys. and Soft.*, 99:97–108, 2015.

[Wei18] Michael Weiss. The Business of Open Source. In *EuroPLoP2018*, 2018.

[WF06] Robert Winter and Ronny Fischer. Essential layers, artifacts, and dependencies of enterprise architecture. In *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06. 10th IEEE International*, pages 30–30. IEEE, 2006.

[WF12] Tim Wellhausen and Andreas Fießer. How to write a pattern?: A rough guide for first-time pattern authors. In *EuroPLoP2012*, page 5. ACM, 2012.

[WFW13] Joost F. Wolfswinkel, Elfi Furtmueller, and Celeste PM Wilderom. Using grounded theory as a method for rigorously reviewing literature. *European Journal of Information Systems*, 22(1):45–55, 2013.

[WFYZ20] Fenfen Wei, Nanping Feng, Shanlin Yang, and Qinna Zhao. A conceptual framework of two-stage partner selection in platform-based innovation ecosystems for servitization. *Journal of Cleaner Production*, 262:121431, 2020.

[WGB11] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium On*, pages 195–204. IEEE, 2011.

[Wie09] Roel Wieringa. Design science as nested problem solving. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, page 8. ACM, 2009.

[Wie14] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering.* Springer, 2014.

[WN13] Michael Weiss and Nadia Noori. Architecture as Enabler of Open Source Project Contributions. In *EuroPLoP2013*, 2013.

[Woh14] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 38. ACM, 2014.

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering.* Springer Science & Business Media, 2012.

[XZZ+20] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, and Min Yang. How Android developers handle evolution-induced API compatibility issues: A large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 886–898. IEEE, 2020.

[YD11] Eric Yu and Stephanie Deng. Understanding software ecosystems: A strategic modeling approach. In *The 3rd Inter. Work. on Soft. Ecosystems*, pages 65–76, 2011.

[YRB08] Liguo Yu, Srini Ramaswamy, and John Bush. Symbiosis and software evolvability. *IT Professional*, 10(4):56–62, 2008.

[Zac96] John A. Zachman. The framework for enterprise architecture: Background, description and utility. *Zachman International*, pages 1–5, 1996.

[ZGS+14] Alfred Zimmermann, Bilal Gonen, Rainer Schmidt, Eman El-Sheikh, Sikha Bagui, and Norman Wilde. Adaptable enterprise architectures for software evolution of SmartLife ecosystems. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, pages 316–323. IEEE, 2014.

[Zim11] Olaf Zimmermann. Architectural decisions as reusable design assets. *IEEE software*, 28(1):64–69, 2011.

[ZJD14] Anneke Zuiderwijk, Marijn Janssen, and Chris Davis. Innovation with open data: Essential elements of open data ecosystems. *Information Polity*, 19(1, 2):17–33, 2014.

[ZKL+09] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *Journal of Systems and Software*, 82(8):1249–1267, 2009.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.

Appendix A

Interview Questionnaire

The following questionnaire is used during the interview with the researchers of CRC 901 On-The-Fly Computing.

Table A.1 Interview Questionnaire

Q1	Where are you in the OTF Basic Scenario?
Q2	What feature do you provide?
Q3	Which roles are involved in your scenario?
Q4	Which roles are involved in your scenario?Who (which role) in general/in OTF Markets is interested in buying your research results as a feature? Why?
Q5	Which quality attributes does your feature add to the market?
Q6	Can you tell one or more scenarios that explain how your research work in OTF Markets?
Q7	Do you imagine more roles in OTF Markets?
Q8	Do you imagine more functions for existing roles?
Q9	In which architectural layer are you?

Appendix B

Quality Attribute Workshop Scenarios

In the following, we present the list of quality attribute scenarios collected during the Mini Quality Attribute Workshop (QAW).

Table B.1 Quality Attribute Scenarios

No.	Architectural Aspect	Scenario	Stakeholder
1	**Business**	**The participant of the market wants fairness. The market should provide accountability measures to incentivise good behaviour**	Subproject C5
2	**Business**	**The requester wants to offer the composition, he bought as a service to others. The market should make it easy to control access to the composition from the view of the requester, compute centre and user.**	Subproject C5
3	**Business**	**An OTF Provider wants to get paid for its services. In order to enforce payment, it restricts access to paying customers. The ecosystem should provide a reliable way to distinguish between paying/non paying customers by providing authentication system.**	Subproject C5
4	**Infrastructure**	**A service provider wants to execute services efficiently in the compute centre. To control the efficiency of the execution, the ecosystem should provide non functional properties such as execution time, power consumption to check. (Productivity/ efficieny)**	Subproject C2
5	**Business**	**A market provider wants to publish services to the online market. To control quality of new services, the ecosystem should provide access to the name and solvency of market participants before they are published at the store**	Subproject C5

No.	Architectural Aspect	Scenario	Stakeholder
8	Application	A service provider wants to update or remove a service from the online market. Should be able to push to or remove from git. (Modifiability)	PoC
9	Infrastructure	An adversary is able to shut down multiple OTF Providers. The system should still be able to guarantee its services to the users via using self-stabilising algorithms. (Robustness)	Subproject A1
10	Application	An OTF provider wants to subscribe to a domain in the market. The market should provide multiple domains that cover the entire interests of the OTF Providers. On the other hand, domains should not be too specific such that the domain only has one or no subscribers. (Creativity)	Subproject A1
11	Infrastructure	Source: Compute Centers Stimulus: Add new type of hardware Artefact : Processing unit Response Measure: Abstraction layer for hardware/hardware virtualisation (Interoperability)	PoC
12	Infrastructure	A requester wants to request a service. He wants to do so in secret i.e., without his competitors learning about his query. The market should hide the query from untrusted entities or preserve the requester's anonymity.	Subproject C1
13	Infrastructure	Source: Service Provider Stimulus: Integrate existing software component into market artefact: legacy software component response measure: tools to generate wrappers/adapters to integrate service (Productivity/Interoperability)	PoC
14	Application	Any framework should offer the same services in one standard way (Consistency)	External service provider 1
15	Business	Source: OTF Provider Stimulus: Combine only services in a service composition that have requested non functional properties artefact: Certificate Response: Validated Service (Robustness)	Subproject C5
16	Business	Source: OTF Provider Stimulus: Create new templates for service compositions artefact: template Response measure: Easy language to describe compositions (Creativity, Modifiability)	Subproject C5
17	Application	A service provider offers a platform for other entities to publish their ML frameworks. One thing to be aware of is readiness of the framework. Any framework should provide methods for training, validation, prediction going online.	External service provider 2
18	Application	A requester wants to ensure that a candidate composition offer has high quality. The market should offer a reputation system to incorporate other users experience into that check. The rating should be of high quality and should be authentic (non fake)	Subproject C1

No.	Architectural Aspect	Scenario	Stakeholder
19	**Application**	**OTF Provider wants to ensure a robust , fast and well performing provision of services tailored to the user's needs and asks the user to state his requirements in a formal way such that it can be created automatically .**	Subproject B2
20	**Application**	**A requester wants to rate service compositions. To allow the requester to give an honest rating, without fearing revenge, the market should ensure anonymity of the requester with respect to the rating .**	Subproject C1
21	**Application**	**Service providers may have interest in getting their code to the market and may want to optimise their code for OTF markets. To this end, they need feedback on the performance of their single service.**	Subproject B2
22	**Infrastructure**	**Executors may want to incorporate specialised hardware as it is the case of using crypto currencies to improve their capabilities and efficiency.**	Subproject B2
23	**Application**	**Service Providers want to make their code available in the OTF market. They want to use foreign code as efficiently as possible while maintaining its main functionality**	Subproject B2
24	**Infrastructure**	**Executors want high throughput at low costs and want to leverage different hardware for that.**	Subproject B2
25	**Application**	**An OTF Provider may want to select only certain sources from the market for services to ensure a good quality from the very beginning and saving resources for exploring the quality on its own. OTF Provider wants not all services all the time, he wants to learn about the space of possible candidates. So he needs matching non functional requirements and a configurator with learning components.**	Subproject B2

Appendix C

List of Existing Ecosystems

This appendix presents the list of the software ecosystems (SECOs) and the architectural patterns associated with them that we identified in the process of pattern mining.

Table C.1 List of Software Ecosystems Captured by Our Pattern Mining

No.	SECO	Architectural Pattern	Architectural Pattern	URL of Store
1	ArcGIS by Esri	Resale Software	—	https://marketplace.arcgis.com/index.html
2	Adobe	Resale Software	—	https://exchange.adobe.com/addons http://cc-extensions.com/index.html
3	Autodesk	Resale Software	—	https://apps.autodesk.com/en
4	Visual Studio	Resale Software	—	https://marketplace.visualstudio.com/search?target=VS&category=All%20categories&vsVersion=&sortBy=Downloads
5	Odoo	OSS-Based	Resale Software	https://www.odoo.com/apps/modules https://github.com/odoo/odoo https://www.odoo.com/page/all-apps
6	AppScale	OSS-Based	Partner-Based	https://www.appscale.com/products/appscale-customer-success/#Marketplace https://github.com/AppScale
7	Cloud Foundry	OSS-Based	—	https://github.com/cloudfoundry/
8	Pivotal	OSS-Based	Partner-Based	https://pivotal.io/platform/pcf-marketplace
9	Docker	OSS-Based	Partner-Based	https://store.docker.com/ https://github.com/docker
10	Datakit	Cross-Platform	—	http://www.datakit.com/en/product_search.php

No.	SECO	Architectural Pattern	Architectural Pattern	URL of Store
11	ExtendSim	Partner-Based	—	https://www.extendsim.com/resources/esp
12	OpenFOAM	OSS-Based	—	https://github.com/OpenFOAM
13	NS-3	OSS-Based	—	https://apps.nsnam.org/
14	Citrix	Partner-Based	—	https://citrixready.citrix.com/ https://citrixready.citrix.com/category-results.html?f1=applications&lang=en_us https://marketplace.visualstudio.com/items?itemName=CitrixDeveloper.CitrixDeveloperVisualStudioExtension
15	BMC	Partner-Based	—	https://marketplace.bmc.com/
16	Sage	Partner-Based	—	https://www.sage.com/marketplace
17	ADP	Partner-Based	—	https://partners.adp.com/
18	Hexagon Geospatial	Resale Software	—	https://store.hexagongeospatial.com/home
19	Ansys	Resale Software	—	https://appstore.ansys.com/shop/ACTApps_act%20apps
20	Informatica	Resale Software	—	https://marketplace.informatica.com/index.jspa
21	Software AG	Partner-Based	Resale Software	https://marketplace.softwareag.com/home
22	The Attachmate Group	Resale Software	—	https://marketplace.microfocus.com https://marketplace.microfocus.com/vertica http://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps
23	Facebook	Resale Software	—	https://developers.facebook.com/docs/plugins/
24	Predix	Partner-Based	—	https://www.predix.io/catalog/apps
25	IFTTT	Partner-Based	—	https://ifttt.com/services/ https://ifttt.com/collections
26	ThingSpeak	Resale Software	OSS-Based	https://thingspeak.com/apps https://thingspeak.com/channels/public https://github.com/iobridge/thingspeak
27	Dropbox	Cross-Platform	—	https://www.dropbox.com/business/app-integrations
28	SimScale	Partner-Based	—	https://www.simscale.com/docs/content/platform/quickstart.html#cae-community
29	SolidWork	Partner-Based	—	http://www.solidworks.com/sw/products/engineering-software-partners.htm
30	Apple	Resale Software	—	https://www.apple.com/app-store/

No.	SECO	Architectural Pattern	Architectural Pattern	URL of Store
31	Google Android	Resale Software	OSS-Based	https://play.google.com/store
32	Microsoft Windows	Resale Software	—	https://www.microsoft.com/en-us/store/apps
33	BlackBerry	Resale Software	—	https://appworld.blackberry.com/webstore/?lang=en&countrycode=DE
34	Amazon App Store	Resale Software	OSS-Based	https://www.amazon.com/s?k=App+Store
35	Mozilla Firefox	Resale Software	OSS-Based	https://addons.mozilla.org/
36	Google Chrome Store	Resale Software	OSS-Based	https://chrome.google.com/webstore/category/extensions
37	SAP	Partner-Based	—	https://store.sap.com/
38	Opera browser	Resale Software	—	https://addons.opera.com/en/
39	Safari	Resale Software	—	https://safari-extensions.apple.com/
40	Salesforce	Resale Software	—	https://appexchange.salesforce.com/
41	Amazon Web Services	Partner-Based	Resale Software	https://aws.amazon.com/marketplace
42	Oracle Cloud	Partner-Based	Resale Software	https://cloudmarketplace.oracle.com/marketplace
43	Eclipse IDE	Resale Software	OSS-Based	https://marketplace.eclipse.org/
44	Envato	Resale Software	—	https://codecanyon.net/category/php-scripts/add-ons
45	Mashape	Resale Software	OSS-Based	https://market.mashape.com/ https://github.com/Mashape
46	Binpress	Resale Software	—	http://www.binpress.com/browse
47	CytoScape	Resale Software	OSS-Based	http://apps.cytoscape.org/apps/all https://github.com/cytoscape/cytoscape
48	CTAN: Packages	OSS-Based	—	https://ctan.org/pkg
49	Atlassian Marketplace	Resale Software	—	https://marketplace.atlassian.com/
50	Github	Resale Software	—	https://github.com/marketplace
51	GitLab	Partner-Based	Resale Software	https://about.gitlab.com/applications/
52	Apache Subversion	OSS-Based	—	https://subversion.apache.org/
53	Ubuntu	Resale Software	OSS-Based	https://apps.ubuntu.com/cat/
54	Sparx EA	Partner-Based	—	http://sparxsystems.com/products/3rdparty.html
55	Matlab and Simulink	Partner-Based	—	https://www.mathworks.com/products/connections.html
56	Siemens PLM	Partner-Based	—	https://www.plm.automation.siemens.com/en/products/
57	CorelDraw	Partner-Based	—	https://www.coreldraw.com/en/pages/third-party-tools/

No.	SECO	Architectural Pattern	Architectural Pattern	URL of Stores
58	Inkscape	OSS-Based	—	https://inkscape.org/en/gallery/%3Dextension/ gitlab.com/inkscape
59	QGIS	OSS-Based	—	https://plugins.qgis.org/plugins/
60	Microsoft Azure	Partner-Based	—	https://azuremarketplace.microsoft.com/
61	OwnCloud	Resale Software	OSS-Based	https://marketplace.owncloud.com/ https://github.com/owncloud
62	Ingram Micro.	Partner-Based	—	https://us.cloud.im/
63	WordPress	Resale Software	OSS-Based	https://wordpress.org/plugins/ https://wordpress.org/download/source/
64	Intuit QuickBooks	Partner-Based	Resale Software	https://apps.intuit.com/
65	G Suite	Resale Software	—	https://gsuite.google.com/marketplace/
66	FreshDesk	Resale Software	—	https://apps.freshdesk.com/ https://apps.freshservice.com/
67	Slack	Partner-Based	—	https://slack.com/apps
68	IBM	Partner-Based	—	https://www.ibm.com/marketplace
69	Dolibarr	OSS-Based	—	https://www.dolistore.com/2-modules https://www.dolistore.com
70	CenturyLink	Partner-Based	—	https://www.ctl.io/marketplace/
71	GNS3	Resale Software	—	https://www.gns3.com/marketplace
72	HP	Partner-Based	—	https://marketplace.hpe.com
73	Alfresco	OSS-Based	Partner-Based	https://addons.alfresco.com
74	Oracle Enterprise Manager	Partner-Based	—	https://apex.oracle.com/
75	the-iot-marketplace	Partner-Based	—	https://www.the-iot-marketplace.com/
76	The Neura	Cross-Platform	—	https://www.theneura.com/solutions/
77	Google Cloud Platfrom	Partner-Based	—	https://cloud.google.com/products/
78	OpenStack	OSS-Based	Partner-Based	https://www.openstack.org/marketplace/ https://github.com/openstack
79	OpenShift	OSS-Based	Partner-Based	https://marketplace.openshift.com/ https://github.com/openshift
80	Adobe Marketing Cloud Exchange	Resale Software	—	https://exchange.adobe.com/experiencecloud.html

No.	SECO	Architectural Pattern	Architectural Pattern	URLs of Stores
81	VMware Vsphere	Partner-Based	—	https://marketplace.vmware.com/vsx/
82	WHMCS	Resale Software	—	https://marketplace.whmcs.com/ github.com/piwik/piwik
83	Matomo	OSS-Based	—	https://plugins.matomo.org/
84	OpenNebula	OSS-Based	—	https://marketplace.opennebula.systems/appliance github.com/OpenNebula/one.git
85	Oracle Cloud marketplace	Resale Software	—	https://cloudmarketplace.oracle.com/
86	Chef	OSS-Based	—	https://github.com/chef
87	Symantec	Partner-Based	—	https://www.symantec.com/products/products-az
88	OpenText	Partner-Based	—	https://www.opentext.com/what-we-do/partners-and-alliances/app-marketplace-solutions
89	OrCad	Partner-Based	—	http://www.orcad.com/products/marketplace-apps
90	Netapp	Partner-Based	—	http://www.netapp.com/us/cloud-marketplace/index.aspx
91	TIBCO	Partner-Based	—	https://www.mashery.com/
92	Zen-Cart	OSS-Based	—	https://www.zen-cart.com/downloads.php
93	Codeclerks	Resale Software	—	https://codeclerks.com/
94	ClickWorker	Partner-Based	—	https://www.clickworker.de/loesungen/
95	Xamarin Studio	OSS-Based	—	https://github.com/xamarin/XamarinComponents
96	Amazon Alexa	Resale Software	—	https://www.amazon.com/b?node=13727921011
97	Zotero	OSS-Based	—	https://www.zotero.org/support/plugins
98	Zapier	Partner-Based	—	https://zapier.com/apps
99	Zendesk	Resale Software	—	https://www.zendesk.com/apps
100	Segment	Partner-Based	—	https://segment.com/catalog
101	Shippo	Cross-Platform	—	https://goshippo.com/
102	Heptio	OSS-based	—	https://heptio.com/opensource/
103	LibreOffice	Resale Software	OSS-Based	https://extensions.libreoffice.org/
104	VLC	Resale Software	OSS-Based	https://addons.videolan.org
105	Cisco	Partner-Based	—	https://marketplace.cisco.com
106	Apache OpenOffice	Resale Software	OSS-Based	https://extensions.openoffice.org
107	OnShape	Partner-Based	Resale Software	https://appstore.onshape.com
108	WP ERP	Partner-Based	—	https://wperp.com/downloads/
109	IntelligentPlant	Partner-Based	—	https://appstore.intelligentplant.com/Welcome
110	Gradle	OSS-Based	—	https://plugins.gradle.org/ https://github.com/gradle/gradle
111	Apache Cordova	Resale Software	OSS-Based	https://cordova.apache.org/plugins/

Appendix D

General Guide To SecoArc Visual Notation

Table D.1 List of SecoArc Notation Used in the Thesis

SecoArc Notation	Description*
🏢👤 / 👤✓	A *platform provider* is the keystone player and the main decision-maker, responsible for the design and governance of the ecosystem.
👥	A *user* uses a software product or service in the ecosystem.
👥	A *strategic partner* is a long-term third-party provider with a certain access to the platform.
👨‍💻	An *independent developer* implements third-party applications on top of the platform without having a direct partnership with the platform
👤⚙	A *supplier* is responsible for providing a software or hardware resource.
▭	A *basic software element* is a software platform, store, or third-party development.
🟩 Name of Application Feature [Type of Application Feature]	An *application feature* is a software application that is used for the purpose of service provision.
🟦 Name of Application Feature [Type of Application Feature]	An *infrastructure feature* is a software/hardware resource that is used for the purpose of service delivery and execution.
➞	A *business strategy* relation represents a set of business decisions that affect a human actor's interactions with the architecture.
→	A *feature relation* shows the interaction between a human actor with the architecture.
€	A *Fee* refers to a payment imposed on the users or third-party providers.
🔒	An *openness policy* is a business decision that determines permissions to accesse or change the source code.
🏅	An *entrance certificate* shows that a third-party provider has fulfilled certain requirements upon entering the ecosystem.
💿	A *license* refers to a set of legal rules governing usage and redistribution of software in or outside of the ecosystem.
🔍	*Code analysis* checks for leaks and bugs in third-party developments.

* taken from the thesis's main chapters