Universität Stuttgart

A Method for Pattern Application based on Concrete Solutions

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Michael W. Falkenthal
aus Böblingen

Hauptberichter:	Prof. Dr. Dr. h. c. Frank Leymann
Mitberichter:	Univ. Prof. Dr. Schahram Dustdar
Tag der mündlichen Prüfung:	01. Februar 2024

Institut für Architektur von Anwendungssystemen

2024

Contents

1 Introduction and Motivation 13
- 1.1 Research Challenges and Contributions 21
- 1.2 Summary of the Contributions 27
- 1.3 Scientific Publications 28
- 1.4 Structure of the Thesis 31

2 Fundamentals and Related Work 33
- 2.1 Patterns, Pattern Languages, and Pattern Application . . 34
- 2.2 Formalization of Pattern Languages and Pattern Aggregation 38
- 2.3 Pattern Application and Reuse of Implementations . . . 40
- 2.4 Patterns and Frameworks 45
- 2.5 Pattern Libraries and Pattern Repositories 48
- 2.6 Limits in Pattern Application 51

3 The EINSTEIN-Method 55
- 3.1 Roles participating in the EINSTEIN-Method 56
- 3.2 General Prerequisites and Requirements 57
- 3.3 Steps of the EINSTEIN-Method 61
- 3.4 Implementation of the EINSTEIN-Method 75

3.5 Chapter Conclusion 77

4 Pattern Languages and Pattern-based Design 79
4.1 Pattern Languages as Graphs 80
4.2 Pattern Refinement 88
4.3 Pattern-based Designs 98
4.4 Pattern-based Design Models 101
4.5 Implementation of the Design Model Concept 102
4.6 Chapter Conclusion 103

5 Reuse of Pattern Implementations as Concrete Solutions 105
5.1 Concrete Solutions 106
5.2 Formalization of Concrete Solutions 110
5.3 Accessing Concrete Solutions via Selection Criteria . . 115
5.4 Requirements and Capabilities 117
5.5 Pattern Graph with Connected Concrete Solutions . . . 119
5.6 Solution Models 121
5.7 Chapter Conclusion 124

6 Aggregation of Concrete Solutions 127
6.1 Solution Algebras 128
6.2 Solution Languages and Concrete Solution Aggregation Descriptors 135
6.3 Concrete Solution Aggregation Programs 139
6.4 Chapter Conclusion 147

7 Semi-Automated Aggregation of Concrete Solutions 149
7.1 Concept of a Toolchain to support the EINSTEIN-Method 150
7.2 Algorithmic approach to implement the semi-automated Aggregation of Concrete Solutions 154
7.3 Discussion 158
7.4 Chapter Conclusion 160

8 Toolchain and Validation 163
8.1 Prototypes to support the EINSTEIN-Method 164

8.2 Validation 167
8.3 Chapter Conclusion 178

9 Conclusion and Outlook 181

Bibliography 185

List of Figures 209

List of Symbols 211

List of Algorithms 217

List of Definitions 219

Zusammenfassung

Muster und Mustersprachen haben sich in vielen Domänen zu wertvollen Werkzeugen zur Repräsentation von bewährten Lösungen für oft wiederkehrende Probleme etabliert. Die Anwendung von Mustern ist in der Praxis jedoch mit einigen Herausforderungen verbunden. So ist es beispielsweise schwierig, die richtigen Muster für ein Problem zu finden. Zudem ist die Anwendung von Mustern oft mit einem hohen manuellen Aufwand verbunden, da die Abstraktion von Implementierungsdetails beim Verfassen der Muster dazu führt, dass Musterimplementierungen nicht systematisch wiederverwendet werden können. Dies führt dazu, dass Muster zwar bewährtes Wissen für konzeptionelle Lösungen liefern, diese jedoch stets manuell implementiert werden müssen, wenn sie in einem bestimmten Anwendungsfall verwendet werden. Insbesondere das Zusammenspiel von Mustern in Mustersprachen führt so zu hohem manuellen Aufwand bei der Umsetzung komplexer Anwendungsfälle.

In dieser Arbeit wird deshalb ein Ansatz vorgestellt, welcher die Anwendung von Mustern in der Praxis erleichtern soll. Der Ansatz basiert auf der Idee, dass Implementierungen von Mustern als *konkrete Lösungen* vorliegen, die bei der Umsetzung von Anwendungsfällen direkt wiederverwendet werden können. Dazu wird mit der *EINSTEIN-Methode* ein Rahmen geschaffen, der die systematische Speicherung konkreter Lösungen für deren Wiederverwendung ermöglicht. Die Methode nutzt *Muster-basierte Entwurfsmodelle* zur Modellierung konzeptioneller Lösungen, welche anschließend semi-automatisiert in konkrete Lösungen überführt werden können. Dabei wird die Verfeinerung von abstrakten Mustern über technologisch spezifischere Muster in Richtung konkreter Lösungen unterstützt. Basierend auf einer Formalisierung von Mustersprachen als Graphen werden *Mustergraphen mit verbundenen konkreten Lösungen* eingeführt, welche die systematische Wiederverwendung konkreter Lösungen ermöglichen. Da Muster oft in Kombination verwendet werden, wird ein Ansatz zur automatischen Aggregation konkreter Lösungen mittels *Aggregationsoperatoren* vorgestellt. Dabei wird mit *Lösungssprachen* zunächst ein Ansatz präsentiert, der die manuelle Aggregation konkreter Lösungen zu einer Gesamtlösung unterstützt. Für die Wiederverwendung konkreter Lösungen wird ein iterativer IT-gestützter Ansatz vorgestellt, der es ermöglicht, Muster in Entwurfsmodellen mit konkreten Lösungen zu ersetzen. Daraus entstehende *Lösungsmodelle* können dann mittels Aggregationsoperatoren zu einer Gesamtlösung aggregiert werden. Für die Automatisierung der Aggregation konkreter Lösungen werden *Lösungsalgebren* eingeführt, die es erlauben anhand von Aggregationsoperatoren mathematische Strukturen über der Menge an konkreten Lösungen zu definieren, wobei Aggregationsoperatoren für die automatisierte Aggregation konkreter Lösungen als *Lösungsaggregationsprogramme* implementiert werden können. Diese ermöglichen es, Lösungsmodelle semi-automatisiert zu Gesamtlösungen zu aggregieren, wofür ein Algorithmus vorgestellt wird. Die praktische Umsetzbarkeit der vorgestellten Ansätze sowie die Übertragbarkeit der *EINSTEIN-Methode* auf verschiedene Domänen wird durch den Entwurf einer Werkzeugumgebung und anhand entwickelter Werkzeugprototypen dargelegt. Abschließend wird die Anwendbarkeit der vorgestellten Konzepte anhand von Validierungsszenarien in verschiedenen Domänen gezeigt.

Abstract

Patterns and pattern languages have become valuable tools in many domains for representing proven solutions to frequently recurring problems. However, the use of patterns presents some challenges in practice. For example, it is often difficult to find the right patterns for a problem at hand. In addition, the application of patterns often involves a lot of manual effort, since the abstraction of implementation details when writing the patterns means that pattern implementations cannot be systematically reused. As a result, although patterns provide proven knowledge for conceptual solutions, they always have to be manually transformed into concrete implementations when a pattern is used in a specific use case. In particular, the interaction of patterns in pattern languages, thus, leads to high manual effort when implementing complex use cases.

Therefore, in this thesis an approach is presented which aims at facilitating the use of patterns in practice. The approach is based on the idea that implementations of patterns are kept available as *Concrete Solutions* that can be directly reused in the implementation of use cases. To this end, the *EINSTEIN-Method* provides a framework for systematically storing concrete solutions for their reuse. The method uses *Pattern-based Design Models* to model conceptual solutions, which can subsequently be trans-

formed into concrete solutions in a semi-automated way. This involves supporting the refinement of abstract patterns via more technologically specific patterns towards concrete solutions. Based on a formalization of pattern languages as graphs, *Pattern Graphs with connected Concrete Solutions* are introduced, which enable the systematic reuse of concrete solutions. Since patterns are often used in combination to solve complex problems, an approach for automating the aggregation of concrete solutions using *Aggregation Operators* is presented. In addition, the principle of pattern languages is also projected to the space of concrete solutions and, thus, with *Solution Languages* an approach is presented that also supports the manual aggregation of concrete solutions to an overall solution. For the reuse of concrete solutions, an iterative IT-supported approach is presented that allows to replace patterns in design models with concrete solutions. Resulting *Solution Models* can then be aggregated to an overall solution using aggregation operators. For automating the aggregation of concrete solutions, *Solution Algebras* are introduced that allow mathematical structures to be defined over the set of concrete solutions. For automating the aggregation of concrete solutions, it is also shown how the concept of aggregation operators can be implemented as *Solution Aggregation Programs*. These allow solution models to be aggregated into overall solutions in a semi-automated manner controlled by the user. For the identification of potential aggregation steps in a solution model, an algorithm is presented that supports the user in the selection of concrete solutions to be aggregated in the solution model.

For the transferability of the *EINSTEIN-Method* into different domains, a tool environment is conceptually described. The practical feasibility of the presented approaches as well as the tool environment is demonstrated by an overall architecture and various tool prototypes. Finally, the feasibility of the presented concepts is shown by means of validation scenarios in different domains.

Acknowledgements

First of all, I would like to thank Prof. Hon.-Prof. Dr. Dr. h. c. Frank Leymann for his support and guidance while working on this dissertation. He brought me to the University of Stuttgart and thus enabled me to conduct the research presented in this thesis. It has been my privilege to be part of the Institute of Architecture of Application Systems and I am grateful for the many opportunities and experiences it has provided me. Furthermore, I would also like to thank Prof. Dr. Schahram Dustdar for taking on the second reviewer position. Many thanks go also to my colleagues at the Institute of Architecture of Application Systems who supported me during my time at the University of Stuttgart and beyond. In particular, I would like to thank Prof. Dr. Uwe Breitenbücher and Dr. Oliver Kopp, with whom I have developed a close friendship over the years. With them I was able to have many interesting discussions late into the night, and they always challenged but also always supported me in my research work. I would like to dedicate this thesis to the people in my life who made it possible for me to follow the path that led me to this thesis – my parents Monika and Klaus Falkenthal. Finally, I would also like to dedicate this work to my family, who grew up during the dissertation, my children Henri and Lotta Falkenthal, and especially my wife Julia Falkenthal. They have always been the rock of my life while working on this dissertation.

Chapter 1

Introduction and Motivation

Pattern languages, originally introduced by Alexander et al. [AIS77] in the domain of building architecture, evolved during the last decades in many domains to powerful means for documenting and communicating knowledge about proven solutions for frequently recurring problems. In his work, Alexander defines patterns by the following characteristics: "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*" [AIS77, p.x]. Thereby, "*the pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing*" [Ale79, p.247]. Thus, patterns encapsulate knowledge about designs that have proven to solve problems we are often faced within specific contexts. Moreover, patterns are interwoven to pattern languages, i.e., to networks of patterns that help to solve more complex and holistic problems [Koh10]. Such connections of patterns offer generativity that goes far beyond isolated patterns. In this sense, they are capable of creating entire structures [AIS77; App97].

This is why throughout the years a manifold of pattern languages has been authored in many different domains. For example, pattern languages describe, among others, how costumes can be used in films to communicate certain character traits, moods or character stereotypes in the humanities [Bar18]. Further, pattern languages from the domain of educational sciences describe how creative learning [IM10] can be supported or how good teaching [Ant96] can be carried out. In addition, there also exist pattern languages for surviving earthquakes [FSS+13] or to purposefully manage challenges due to ongoing societal change [FHM+15]. Domains, where pattern languages got also prevalent, are computer science and information technology. For example, looking at designing application systems one can identify a variety of available pattern languages dealing with this topic. Moreover, there are pattern languages dealing with message-based integration of application components [HW04], the interplay of Internet of Things devices [RBF+16; RBF+17a; RBF+17b; RBF+17c; RBF+19; RFBL17], the development of cloud-native application components enabling them to scale for alternating workloads [FLR+14]. Also new emerging technologies, such as quantum computing, are covered by pattern languages [BBB+23; BBL+22; WBLS20; WBLS21]. Thus, it is a common approach for practitioners to design applications based on available languages, which is not least evident from the book series *Pattern-oriented Software Architecture* [BMR+96].

Consequently, patterns are a powerful means to solve frequently recurring problems because they capture the essence of many proven solutions in an abstract way, i.e., technology and implementation agnostic [AIS77; Ale79]. Thus, the solutions described in pattern documents are not limited to specific use cases or setups but rather enable to transfer the captured solutions principles to individual situations. However, although this principle promotes broad reusability, it also opens up challenges in using patterns and pattern languages. E.g., this can be attributed to the fact that pattern implementations are lost during the authoring process of patterns, although they might be helpful when applying patterns later. Thus, while concrete implementations are sometimes described textually in pattern sections such

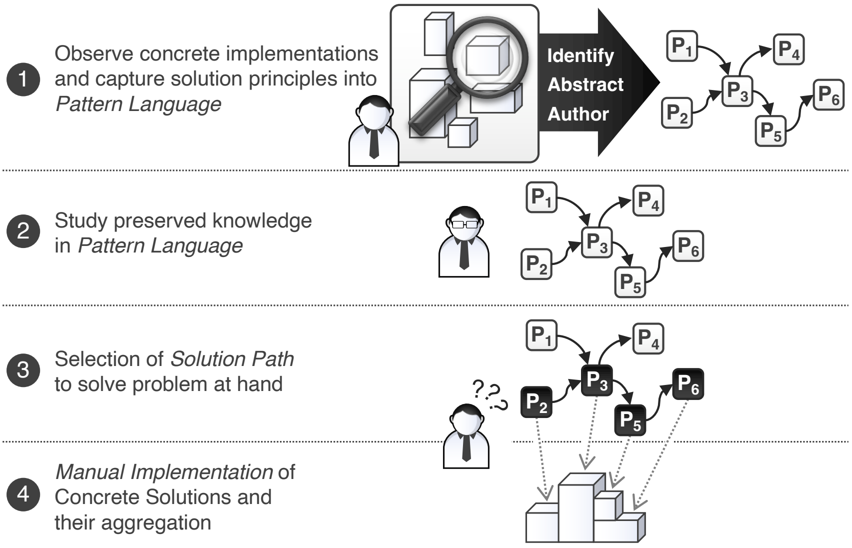

1
Observe concrete implementations and capture solution principles into *Pattern Language*
Identify Abstract Author
P1
P2
P3
P4
P5
P6
2
Study preserved knowledge in *Pattern Language*
3
Selection of *Solution Path* to solve problem at hand
???
4
Manual Implementation of Concrete Solutions and their aggregation

Figure 1.1: Current state of the art of pattern application

as *Known Uses* or *Examples* [MD97] or in the form of (pseudo) programming code in the domain of IT, the actual implementations are not captured in the patterns themselves. This shortcoming is conceptually depicted in Figure 1.1, where proven solution principles are identified on the basis of concrete implementations and then abstracted and authored into patterns and pattern languages in Step 1. Those pattern languages can then be used to solve recurring problems that are addressed by one or several patterns or their interplay, respectively.

Due to the abstract nature of the captured solution knowledge in the patterns they can be applied for many different use cases because they are not bound to specific technologies, tools, or environmental constraints [AIS77; Ale79]. However, for purposefully applying a pattern language it has to be closely studied beforehand, such that especially the interplay of the patterns is grasped and understood as depicted in Step 2. When applying a pattern language an user has to find and select matching patterns that typically solve his or her problem at hand in combination. Such a selection results in pattern sequences or so-called *solution paths* [Zdu07] or *solution graphs* [FBB+15], respectively, as shown in Step 3 in Figure 1.1. It has to be noted that these selected patterns provide solution principles on the conceptual level and have to be implemented for a concrete use case at hand. However, since concrete implementations typically get lost during the authoring process of the patterns, the initially investigated solution artifacts cannot be reused for pattern application. The patterns must, therefore, be implemented for each application manually as indicated in Step 4 of Figure 1.1. This also applies to use cases in which previous implementations could be reused. All this leads to an immense amount of time spent applying patterns [FBB+14a; FBB+14b], often accompanied by implementation mistakes that could be prevented if pattern implementations could be reused and improved over time – the same way patterns themselves strive for reusability of proven knowledge. According to this, the concept of pattern languages currently only supports conceptually solving problems and designing solutions but is decoupled from the level of the actual implementation.

This is where the vision of this thesis is heading to overcome the disconnection of *conceptual pattern-based solution designs* from their actual implementation. The core idea is to establish guidance for pattern applications as depicted in Figure 1.2. Instead of leaving users alone after selecting patterns, the implementation of the patterns should be supported.

The vision of this thesis, as shown in Figure 1.2, therefore, strives for supporting the user after Step 3. Thereby, rather than studying patterns and applying them manually users can utilize the selected patterns to conceptually design the desired solution to solve the problems at hand, which is shown in the differently shaped Step 4 in Figure 1.2 in contrast to Figure 1.1. Thereby it is important that the selected patterns represent real entities from the domain addressed by the pattern language. They describe an entity, which is ultimately part of the system to be built and at the same time, abstractly, how the entity itself is built [Ale79]. Such patterns are referred to as *generative patterns* in contrast to non-generative patterns,

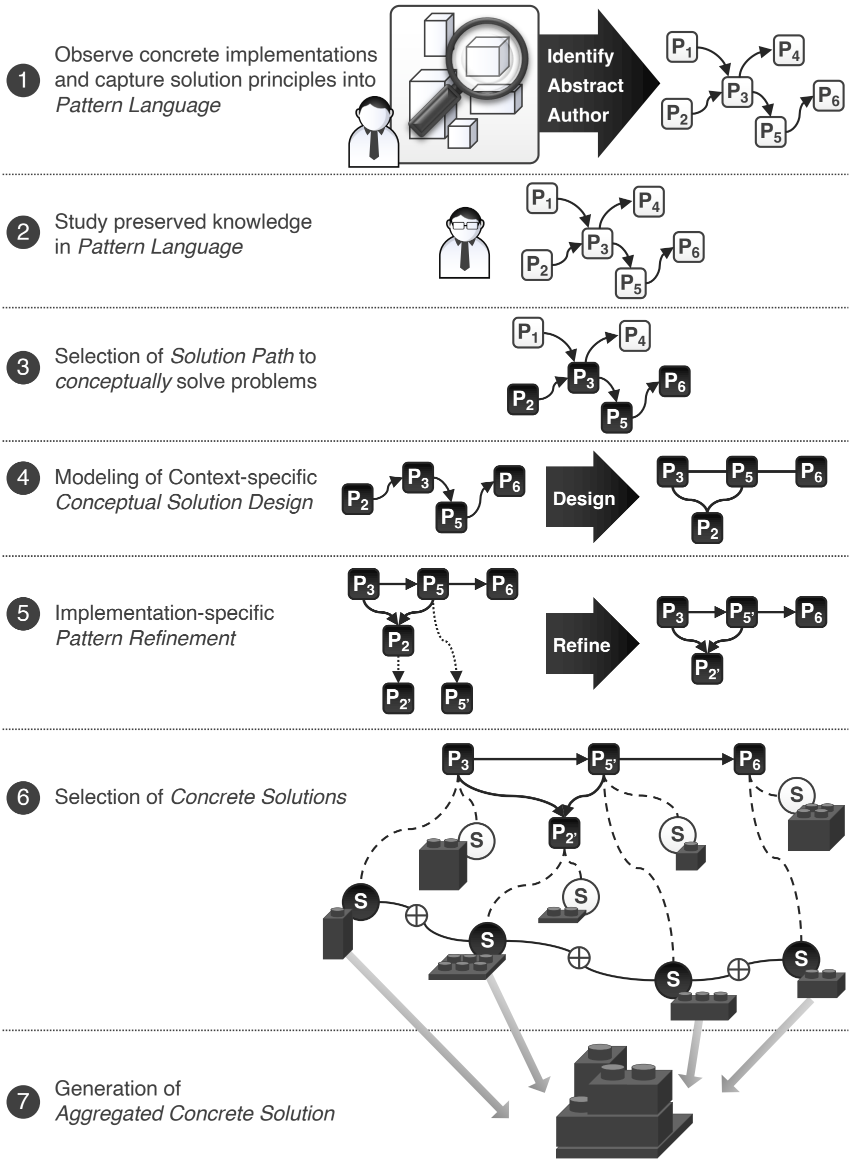

1
Observe concrete implementations and capture solution principles into *Pattern Language*
Identify Abstract Author
P1 P2 P3 P4 P5 P6
2
Study preserved knowledge in *Pattern Language*
3
Selection of *Solution Path* to *conceptually* solve problems
4
Modeling of Context-specific *Conceptual Solution Design*
Design
5
Implementation-specific *Pattern Refinement*
Refine
P2' P5'
6
Selection of *Concrete Solutions*
S
7
Generation of *Aggregated Concrete Solution*

Figure 1.2: Vision of pattern application based on the selection of concrete solutions and their aggregation

which just describe phenomena without providing hints about how to solve or reproduce them [App97]. The conceptual design of solutions on the basis of patterns transforms the solution path into a model that represents the interplay of the selected patterns according to the real problem to be solved. This means that the connections between the patterns are enriched and supplemented by semantics that is specific to the domain of the use case. In this way, the patterns are arranged according to the actual context and requirements a user is confronted with. This is necessary, e.g., if a pattern serves as the basis for the succeeding patterns in a selected solution path. Depicted in Step 4 of Figure 1.2, the entities represented by the patterns P_3 and P_5 are interrelated with P_2. This connection explicates domain-specific semantics in the designed conceptual solution. Furthermore, if multiple instances of a pattern are required in the intended target solution design this has to be specified in such a model. Thus, the solution path has to be translated into an interplay of patterns representing the desired system configuration.

In some cases, pattern languages exist that allow to refine such conceptual solution designs via patterns that capture solution principles that describe more specific implementation contexts. Thus, a user can replace more abstract patterns with those that explain how to elaborate implementations of the pattern closer to actual implementations. The refinement of the two patterns P_2 and P_5 by the patterns $P_{2'}$ and $P_{5'}$ is illustrated in Step 5 of Figure 1.2. This allows to reduce the gap towards the concrete implementations of solution concepts described by patterns. An example for this principle can be found in the interplay of the technology-agnostic patterns by Fehling et al. [FLR+14] and the more technology-specific ones by Amazon Webservice [Ama13] and Homer et al. [HSB+14]. Patterns from the latter pattern languages can be used to refine patterns by Fehling et al. [FLR+14] towards the respective cloud environments from AWS and Microsoft. In that respect, a user can receive implementation advice regarding the technical context of these cloud offerings and update the conceptual design by the more specific patterns.

Finally, to overcome the disconnection between conceptual solutions captured in patterns and their actual implementations for specific use cases, *Concrete Solutions* are introduced which are captured along with the patterns they implement. This is depicted in Step 6 of Figure 1.2. Once a pattern is implemented the developed artifacts implementing the pattern can be stored to be reused later. Concrete solutions enable to reduce efforts for the application of patterns because they either can be reused directly or at least they can provide a basis for the implementation of a pattern for a specific use case, which also supports reusability of proven artifacts. Furthermore, by keeping concrete solutions connected with the patterns themselves they can evolve over time in quality by reuse. This is because they can be reused and improved each time a pattern needs to be applied. Of course, concrete solutions of a single pattern can vary significantly depending on technical requirements and implementation contexts. An example are implementations in different programming languages. This creates more and more concrete solutions over time, which have to be managed to reuse them appropriately according to different problems, technical circumstances, and requirements of an user.

Therefore, as part of the vision if this thesis, it is also important to propagate the principle of combining patterns to the level of concrete solutions. This is illustrated in Figure 1.2 in the succeeding and final Step 7. To this end, a mechanism to ease the aggregation of concrete solutions to overall solutions is envisioned. For this purpose, *Aggregation Operators* that provide domain-specific logic to assemble concrete solutions and generate an implementation of the formerly specified interplay of patterns in the conceptual solution design are used. Thus, by selecting concrete solutions and aggregating them, the conceptual solution design is converted into an implementation, which can then be refined and adapted manually.

To summarize, the vision of this thesis targets systematically reusing existing implementations of patterns rather than redeveloping them every time a pattern is applied. This supports a change in the way patterns are applied for use cases at hand and aims for easing their application. Moreover, it shifts the focus from spending efforts for the recurring design and construction of pattern implementations towards finding and reusing appropriate concrete solutions connected to patterns and aggregating them. This enables that new implementations of patterns in the form of concrete solutions can be subjected to the same quality assurance and improvement over time as the solutions that originally led to patterns. Since the findings and results of this thesis primarily stem from the analysis of patterns from the IT-domains of application architecture and cloud computing, the contributions and concepts will be explained and discussed mainly in this context. However, to demonstrate the transferability of the presented concepts to other domains, they will also be applied and discussed in the non-IT area of Costumes in Films.

1.1 Research Challenges and Contributions

Challenge 1: The Need for a methodical Approach for Applying Patterns based on Concrete Solutions and their Aggregation

As motivated above, there is currently a lack of reuse of concrete solutions at the time of pattern application. Consequently, there is also a lack of a fundamental methodical approach that allows to leverage the connection between patterns and their implementations to establish, ease, and guide reuse of the latter. Thereby, it has to be considered that patterns are typically applied in combination rather than as single solutions. More precisely, patterns are often used as interwoven artifacts in conceptual solution designs. Therefore, it is inevitable to reflect this fact in a methodical framework that enables to capture besides concrete solutions also documentation and, if applicable, proper logic and programs allowing to automatically aggregate concrete solutions. This must enable the translation of pattern-based conceptual solution designs, which specify the combination of patterns, to the level of concrete solutions along with mechanisms enabling to aggregate them as intended by the pattern-based solution design. Furthermore, to leverage the reusability of concrete solutions, the aggregation of them has to be automated.

Contribution 1 (The EINSTEIN-Method) In this work, the *EINSTEIN-Method* (patt**e**rn appl**i**catio**n** ba**s**ed on concre**te** solut**i**o**n**s) for applying patterns supported by means of concrete solutions, which are concrete implementations of patterns, and their aggregation is presented. The EINSTEIN-Method is based on a formalization of pattern languages as *Pattern Graphs with connected Concrete Solutions*. The concrete solutions are stored along with *Aggregation Descriptors* and *Aggregation Operators* forming *Solution Languages*. Thereby, aggregation operators represent domain-specific logic that enables to combine concrete solutions. The individual building blocks are further detailed in the next contributions of this work.

Challenge 2: Design Modeling based on Patterns from different Pattern Languages

Pattern languages typically cover a limited scope of problems and solutions in a domain and, therefore, provide a strict network of patterns. However, since the design of solutions is mostly an endeavor that requires expertise from many different perspectives, different pattern languages from different domains typically have to be used in combination to develop comprehensive solutions [FBL18]. But since authored in different pattern languages, they are disconnected from each other in the first place. Therefore, it is essential to establish an approach that enables to combine pattern languages in the sense that relations between patterns from different pattern languages can be captured. This is especially of interest if patterns from different pattern languages refine each other in terms of details that ease and guide their realization with different technologies [FBB+15], or if a user requires to combine patterns from different yet disconnected pattern languages in a way that was not reflected by the initial languages [WBB+20]. Thereby, it is also useful to support to reduce the initially vast number of patterns from different pattern languages to only those that are of interest for solving the problems under investigation. To systematically reuse concrete solutions it is necessary to lay a foundation via pattern-based solution designs.

Contribution 2 (Pattern Graphs and Pattern-based Design Models) As a prerequisite to translate problem solving via pattern languages to the reuse and aggregation of concrete solutions, a formalization of pattern languages as *Pattern Graphs* is introduced. Pattern graphs enable to link patterns, even from different pattern languages, via expressing the relations between patterns. Those relations can be enriched by arbitrary additional structured explanations, which guide users in applying patterns in combination. These structures are formalized as typed edges of a pattern graph. As a foundation for expressing solution designs with patterns, *Pattern-based Design Models* are introduced as the conceptual specifications of overall solutions.

Challenge 3: Linking Patterns and Concrete Solutions

To enable the systematic reuse of concrete solutions, all artifacts that form the concrete solutions have to be gathered and stored. It is important to note that some kinds of concrete solutions can be directly stored, such as programming code or configuration files, while for others it is necessary to create digital representations beforehand. The latter applies typically to concrete solutions that are tangible in the sense that they are physical objects, such as costumes in films [FBB+17], or complex concepts, such as specific intonations and tone sequences in music [BBE+17]. However, since a pattern can be implemented in many different manners, the amount of concrete solutions gathered increases over time and can easily get unmanageable. Therefore, it is important to systematically connect concrete solutions with the patterns they implement along with additional information that allows users to purposefully search the set of available solutions [FBB+14a; FBB+14b]. It is not only to support distinguishing the different concrete solutions but to enable resolving specific requirements of users by selecting proper concrete solutions of a pattern.

Contribution 3 (Concrete Solution Descriptors and Pattern Graphs with connected Concrete Solutions) Concrete solutions are the core means to enable the systematic reuse of pattern implementations as described in this work. Therefore, pattern implementations are enriched by metadata and linked with the patterns they implement to form *Pattern Graphs with connected Concrete Solutions*. Metadata annotated to concrete solutions is documented in *Concrete Solution Descriptors* capturing requirements, capabilities, and arbitrary selection criteria. Requirements define circumstances and constraints that have to hold to apply a concrete solution while capabilities describe features, which are added by a concrete solution. This means capabilities of a concrete solution can fulfill requirements of others to generate a concrete solution aggregate. In addition, selection criteria add metadata to support the selection of appropriate concrete solutions.

Challenge 4: Domain-specific Analysis, Formalization, and Documentation of Concrete Solution Aggregation

While pattern graphs with connected concrete solutions help in the selection of concrete solutions of individual patterns, the conversion of pattern-based design models into comprehensive implementations often requires the aggregation of concrete solutions of different patterns. This is because the implementation of pattern-based design models requires the interplay of concrete solutions associated with the patterns contained in the models. This is referred to as *Concrete Solution Aggregation*, i.e., the artifacts of the concrete solutions are put together to form an aggregate [FBB+14a; FBB+14b]. However, since artifacts of concrete solutions from different domains typically differ in their technical realization, a means is required to systematically analyze and grasp the domain-specific properties and operations required to aggregate concrete solutions. This is especially important in all domains where concrete solution aggregation can be automated. In addition, in areas that are concerned with concrete solutions that cannot be aggregated automatically, it is necessary to document how they can be combined for reuse [FBBL17; FL17].

Contribution 4 (Solution Algebras and Solution Languages for Concrete Solution Aggregation) *Aggregation Operators* are introduced to capture the logic required for aggregating concrete solutions according to the conceptual interplay of the patterns they implement. Aggregation operators are used to define structures on sets of concrete solutions, which are denoted *Solution Algebras*. Solution algebras encapsulate the algebraic properties of concrete solution aggregation and are specific for different types and domains of concrete solutions. They are the foundation for systematically describing and implementing aggregation logic. Further, the concepts of *Solution Languages* along with *Concrete Solution Aggregation Descriptors* as documentation artifacts allowing to systematically keep knowledge about how to aggregate different concrete solutions are introduced.

Challenge 5: Automating the Selection and Aggregation of Concrete Solutions to create Solution Aggregates

Concrete solution aggregation descriptors facilitate and guide the aggregation of concrete solutions in areas where it cannot be fully automated. They provide user readable explanations of how actual aggregations can be performed. However, in many areas, especially in IT, the concrete solution artifacts that are created when applying a pattern are typically digital. In the same course the fast growing number of new base technologies and products in IT leads to many different pattern applications. Thereby, an even bigger number of concrete solutions is created, which requires to be managed and searched in order to enable their systematic and purposeful reuse. Therefore, it is reasonable to enable and support the automated selection and aggregation of concrete solutions related to patterns. In principle this means that the selection of proper concrete solutions, aggregation operators, and their execution have to be supported by algorithms and tools. Thereby, it is important to provide users with the opportunity to specify requirements and constraints that have to hold for an overall solution, which complement previously created pattern-based design models.

Contribution 5 (Semi-automating Approach for the Selection and Aggregation of Concrete Solutions) A semi-automated approach is introduced that enables to guide and automate the process of selecting and aggregating of concrete solutions. Thereby, pattern-based design models are translated into *Solution Models* that represent the interplay of concrete solutions and *Solution Aggregation Programs* are introduced as implementations of aggregation operators that support the automated aggregation of concrete solutions. An algorithm is introduced, which consumes solution models and solution aggregation programs to enable the iterative and semi-automated aggregation of the concrete solutions in the solution model.

Challenge 6: Validation and Tool Support

The overall approach as presented in this work is all about capturing and reusing existing implementations of patterns. To provide evidence about the practical feasibility of the presented approaches and concepts, a prototypical implementation covering all aspects of the EINSTEIN-Method is required. Thus, fundamental artifacts such as connected pattern and solution repositories are needed. Alongside, components based on those repositories have to be developed that allow to specify user inputs, such as pattern-based design models, the documentation of solution languages, and the execution of solution aggregation programs. Thereby, it is important to enable users to influence the selection of concrete solutions and their aggregation where needed by specifying their requirements. Whereas, cumbersome, time-consuming, and error-prone steps, such as the actual selection of suitable concrete solutions from a solution repository or the aggregation of them to overall solutions, have to be automated as much as possible. All these aspects target to leverage reuse of concrete solutions to gain time-savings and preventing errors when (re)implementing patterns over and over again.

Contribution 6 (Toolchain for the EINSTEIN-Method) To demonstrate the practical feasibility of the EINSTEIN-Method, concepts and approaches introduced, an architecture and prototype of a toolchain for connecting patterns, concrete solutions, and aggregation operators is presented. On the one hand, the toolchain consists of extendable pattern and solution repositories enabling to capture and store pattern languages, solution languages, concrete solution aggregation descriptors, and solution aggregation programs. On the other hand, it enables to connect additional services, such as engines for automating concrete solution selection and the aggregation of concrete solutions, aiming for extendability of the functionalities of the underlying repositories.

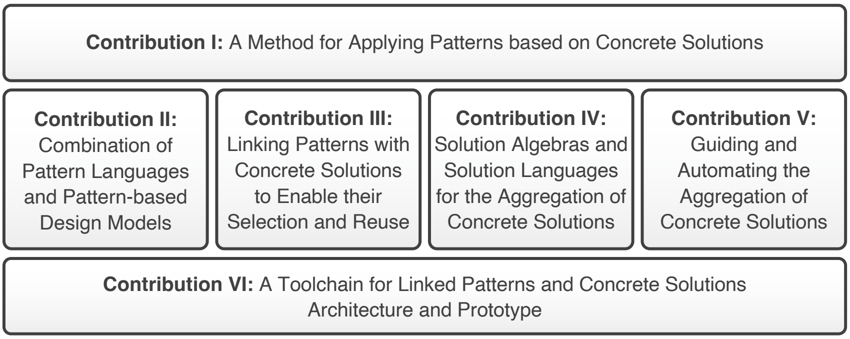

Contribution I: A Method for Applying Patterns based on Concrete Solutions
Contribution II: Combination of Pattern Languages and Pattern-based Design Models
Contribution III: Linking Patterns with Concrete Solutions to Enable their Selection and Reuse
Contribution IV: Solution Algebras and Solution Languages for the Aggregation of Concrete Solutions
Contribution V: Guiding and Automating the Aggregation of Concrete Solutions
Contribution VI: A Toolchain for Linked Patterns and Concrete Solutions Architecture and Prototype

Figure 1.3: Overview of the research contributions

1.2 Summary of the Contributions

The vision outlined in the previous section can be broken down into six major research challenges and contributions that were elaborated in the course of this thesis. The contributions complement and build on each other to realize the outlined vision, as depicted in Figure 1.3. Thereby, Contribution 1 is the EINSTEIN-Method for applying patterns based on concrete solutions, which lays a systematic framework for the reuse of concrete solutions. Further, Contribution 2 introduces a concept to ease the usage of patterns from different pattern languages in pattern-based design models. To bridge the gap between conceptual design via patterns and their implementation, Contribution 3 introduces concrete solutions as reusable pattern implementations that are organized in solution repositories. Contribution 4 introduces solution algebras as the formal foundation of automated concrete solution aggregation. This, in turn, is introduced as Contribution 5 by means of a semi-automated approach that enables to aggregate concrete solutions based on constraints specified by the user. Finally, Contribution 6 is the technical validation of the presented concepts by means of an architecture and prototype supporting the EINSTEIN-Method along with validation scenarios from different domains.

1.3 Scientific Publications

The following peer-reviewed papers were published in journals and conferences and resulted from research conducted in the course of this thesis.

1. M. Falkenthal et al. “From Pattern Languages to Solution Implementations”. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications*. Xpert Publishing Services, May 2014, pp. 12–21
2. M. Falkenthal et al. “Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains”. In: *International Journal On Advances in Software* 7.3&4 (Dec. 2014). IARIA, pp. 710–726
3. M. Falkenthal et al. “Leveraging Pattern Application via Pattern Refinement”. In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change*. epubli, June 2015, pp. 38–61
4. M. Falkenthal and F. Leymann. “Easing Pattern Application by Means of Solution Languages”. In: *Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications*. Xpert Publishing Services, 2017, pp. 58–64
5. M. Falkenthal et al. “Solution Languages : Easing Pattern Composition in Different Domains”. In: *International Journal On Advances in Software* 10.3&4 (2017). IARIA, pp. 263–274
6. M. Falkenthal et al. “Pattern research in the digital humanities: how data mining techniques support the identification of costume patterns”. In: *SICS Software-Intensive Cyber-Physical Systems* 32.3-4 (2017). Springer, pp. 311–321
7. M. Falkenthal et al. “On the Algebraic Properties of Concrete Solution Aggregation”. In: *SICS Software-Intensive Cyber-Physical Systems* (2019). Springer

8. M. Falkenthal et al. “The Nature of Pattern Languages”. In: *Pursuit of Pattern Languages for Societal Change*. Edition Donau-Universität Krems, 2018, pp. 130–151

The next list shows co-authored publications that additionally complement the approaches presented in this work, but were also shaped on the basis of these.

1. R. Reiners et al. “Requirements for a Collaborative Formulation Process of Evolutionary Patterns”. In: *Proceedings of the 18th European Conference on Pattern Languages of Programs EuroPlop*. ACM, 2013, Article No. 16
2. C. Fehling et al. “PatternPedia – Collaborative Pattern Identification and Authoring”. In: *Proceedings of Pursuit of Pattern Languages for Societal Change. The Workshop 2014.* epubli, Aug. 2015, pp. 252–284
3. H. Finidori et al. “The PLAST Project – Pattern Languages for Systemic Transformations”. In: *International Journal of the Spanda Foundation* VI.1 (2015). Spanda Foundation, pp. 205–218
4. L. Reinfurt et al. “Internet of Things Patterns”. In: *Proceedings of the 21th European Conference on Pattern Languages of Programs*. ACM, 2016
5. C. Endres et al. “Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications”. In: *Proceedings of the Ninth International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services, Feb. 2017, pp. 22–27
6. L. Reinfurt et al. “Internet of Things Patterns for Devices”. In: *Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications*. Xpert Publishing Services, 2017, pp. 117–126

7. L. Reinfurt et al. “Internet of Things Patterns for Devices: Powering, Operating, and Sensing”. In: *International Journal on Advances in Internet Technology* (2017). IARIA, pp. 106–123
8. L. Reinfurt et al. “Applying IoT Patterns to Smart Factory Systems”. In: *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2017, pp. 1–10
9. L. Reinfurt et al. “Internet of Things Security Patterns”. In: *Proceedings of the 24th Conference on Pattern Languages of Programs*. ACM, 2017
10. L. Reinfurt et al. “Internet of Things Patterns for Device Bootstrapping and Registration”. In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. ACM, 2017
11. J. Barzen et al. “The vision for MUSE4Music – Applying the MUSE method in musicology”. In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 323–328
12. L. Harzenetter et al. “Pattern-based Deployment Models and Their Automatic Execution”. In: *11th IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE, 2018, pp. 41–52
13. L. Reinfurt et al. “Internet of Things Patterns for Communication and Management”. In: *Transactions on Pattern Languages of Programming* IV (2019)
14. K. Saatkamp et al. “An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models using First-order Logic”. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 495–506
15. L. Reinfurt et al. “Where to Begin - On Pattern Language Entry Points”. In: *SICS Software-Intensive Cyber-Physical Systems* 35 (2020). Springer

16. M. Weigold et al. “Pattern Views: Concept and Tooling for Interconnected Pattern Languages”. In: *Communications in Computer and Information Science* 1310 (2020), pp. 86–103

Including these listed publications, in the course of this thesis, 53 peer reviewed papers were published.

1.4 Structure of the Thesis

The remainder of this thesis is structured as following: the fundamentals of pattern languages and pattern application are set out in Chapter 2. Thereby, existing formalizations of patterns and pattern languages are discussed besides embedding the concepts of this work in the model-driven architecture and computer-aided software engineering approaches. Finally, related work on pattern repositories is discussed regarding the applicability and suitability for the approaches and concepts introduced in this thesis.

The *EINSTEIN-Method* is introduced in Chapter 3 and forms a methodical bracket around the concepts of this work. There, challenges and requirements for the systematic reuse of concrete solutions are discussed as well as the interplay of different roles that participate in the method.

Chapter 4 lays the conceptual foundation for the creation of *Pattern-based Design Models* by formalizing pattern languages on the basis of graph theory. This is the fundamental structure underlying the concepts of this thesis. However, in order to allow the combination of patterns from different pattern languages the concept of *Pattern Language Aggregation* is introduced based on the previous formalization of pattern graphs.

It is further explained how to foster the reuse of concrete solutions by *Concrete Solution Descriptors* in Chapter 5. In combination with *Selection Criteria* those are means to systematically store concrete solutions connected with patterns. Based on these concepts the formalization of pattern graphs is extended to pattern graphs with connected concrete solutions.

The aggregation of concrete solutions is introduced in Chapter 6. Besides the documentation of aggregation possibilities of concrete solutions via *Solution Languages*, *Aggregation Operators* are introduced as the means to capture (i) the algebraic properties of concrete solution aggregation along with (ii) the actual logic describing how concrete solutions are aggregated. The former aspect provides a means to systematically analyze concrete solutions in order to organize them into sets of related concrete solutions that can be aggregated by specified aggregation operators, which results in *Solution Algebras*. The latter aspect sets the basis for automating the aggregation of concrete solutions by applying *Solution Aggregation Programs* from a solution algebra on concrete solutions.

Chapter 7 introduces a domain-independent approach for automating the aggregation of concrete solutions. The approach considers user inputs to reflect requirements that have to hold for individual concrete solutions and the overall solution aggregate. It shows how *Design Models* are translated into *Solution Models* which, in turn, can be used to iteratively perform aggregations on the contained concrete solutions until a final concrete solution aggregate is generated.

The feasibility of the presented approach is shown by the architecture of a prototypical *Toolchain* in Chapter 8. Validation scenarios spanning different application domains show how the concepts presented in this work can be applied to different pattern languages and application areas.

Finally, the thesis is concluded in Chapter 9 where the results are reflected and an outlook to future work is given, which can further strengthen and extend the presented work valuably.

CHAPTER 2

Fundamentals and Related Work

This chapter discusses the fundamentals and related work on which the presented work is based. It starts with a discussion of the concept of patterns and pattern languages in general and provides informal definitions of essential terms, such as patterns and pattern languages, as used in this work. Furthermore, the state of the art regarding the formalization of patterns, their aggregation and pattern languages is discussed. Moreover, the usage and application of patterns in programming languages, modeling languages, and frameworks is discussed with special attention to the reuse of pattern implementations. The chapter concludes with a discussion of the state of the art of pattern repositories and pattern libraries and identifies deficiencies, which lay the foundation for the research challenges that are tackled in this thesis and led to the contributions of the presented work.

2.1 Patterns, Pattern Languages, and Pattern Application

The concept of patterns and pattern languages was originally developed by Christoper Alexander in the area of building design. The basic ideas about patterns and pattern languages originated in his work *Notes on the Synthesis of Form* [Ale64], a study about the process of designing building architectures that are rated good, vital, and liveable by people. Thereby, he identifies that design problems can be solved by identifying misfits and forces between them. They span a network of trade-offs, called *diagram* by Alexander [Ale64], that have to be balanced once misfits get solved. The fundamental idea Alexander adds to this is that such a network of trade-offs can be analyzed to find significant subsets of misfits that can be solved in combination quite well because the misfits are correlated. This means, creating a design that solves one misfit tends to also solve others in the subset or, in the case of negative correlations, a design decision preferring one misfit over the other has to be taken. The general design principle Alexander suggests is to decompose the overall diagram, which is to complex and overwhelming to be tackled, into smaller subsets until the complexity reaches a manageable level [Ale64, p.93ff]. Such *small* subsets can then be solved and the worked out designs can be aggregated until the overall network of trade-off or misfits is solved.

This fundamental idea can be grasped as the nucleus of the pattern and pattern language idea by Christopher Alexander. Since significant subsets are those that influence the realization of real world use cases the most, it is worth to analyze how those can be resolved properly and how the resulting solutions can be composed to solve the initial overall network of trade-offs [Ale64, p.64ff]. Thus, abstracting the essence of solutions of such subsets leads to solution principles, which solve *recurring problems*. Especially those solutions are of interest, which resolve subsets of misfits in a way that satisfies the requirements of a given use case by considering also the implications imposed by the correlation of the misfits. In the domain of building architecture this means the underlying solution principles materialize in buildings that are rated good, vital, and liveable by people as mentioned above. Such solution principles can be grasped as good designs or *proven solutions* to resolve the tackled trade-offs because they are evaluated by every days live.

Adding those fundamental ideas together, Alexander et al. [AIS77] end up with the definition of what a pattern is:

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* [AIS77, p.x]

The essence of Alexander's definition is that a pattern captures core solution principles enabling to create manifolds of new designs and things on those principles. However, since a pattern itself is the captured essence of solutions, which have proven to be good, Alexander's definition lacks clarity on whether already present solutions could be reused or whether solutions have to be crafted every time a pattern has to be applied (cf. *"[...] without ever doing it the same way twice."*). This can be traced back to the fact that Alexander discovered patterns in the domain of towns and buildings, which merely deals with purely physical entities. However, especially in computer science, the things under investigation are in many cases logical parts, models, code snippets, and in general data, which can be copy and pasted without any effort, which means being highly reproducible. This enables *reusability* of already existing solutions, which goes beyond the definition of a pattern by Alexander. Thus, Alexander's definition focusses on tangible things, such as structures in towns and buildings, but falls short of what is possible with intangible solutions, such as computer programs or digital representations of things. As a result, the main difference between tangible and intangible digital solutions is, that the latter ones can easily be copied and reused.

This requires an adjustment of the above definition of a pattern. The following informal definition of a pattern is used in this work.

Definition 2.1.1 (Pattern – informal) A pattern is a proven conceptual solution to a recurring problem in a certain context. Thereby, the context specifies the situation and setting in which the solution solves the problem, while the solution captures the essence of multiple good solutions implemented in practice, whereby some of them are referenced as known uses. Thus, applying a pattern means applying the solution principles of the pattern to a certain context to craft a suitable solution. ■

In many domains, patterns emerged to a Lingua Franca of the domain, i.e., they form a terminology to communicate about problems and their solutions [DF06]. However, patterns typically do not stand alone, they rather solve overall problems in combination. This has already been illustrated above by the decomposition of misfits into significant subsets, which are solved and the resulting solutions are composed to form an overall solution as described by Alexander [Ale64, p.93ff]. This fundamental principle leads to the definition of an interplay of patterns that is required to design *a new whole*, which corresponds to a solution of the starting point of the above mentioned decomposition of misfits. To form *a whole*, Alexander identifies two basic principles: (i) a pattern is itself a whole, i.e, it defines a space which is filled by other patterns [Ale79, p.185] and (ii) the interplay of patterns required to form a whole. Often this interplay of patterns is called a *pattern language* [Ale79, p.191]. Alexander projects this principles to the area of designing towns, buildings, and further constructions in an interplay. He shows, that *a whole*, such as a town, is essentially made by the interplay of smaller parts, for example, buildings, garages, parks, and all the other sites of a town [Ale79, p.191]. The rational behind that is pointed out in *A Pattern Language: Towns, Buildings, Construction* [AIS77] highlighting the importance of the interplay of patterns:

> *"[...] when you build a thing you can not merely build that thing in isolation, but must also repair the world around it, and within it, so that the larger world at that one place becomes more coherent, and more whole; and the thing which you make takes its place in the web of nature as you make it."* [AIS77, p.xiii]

Alexander underlines this by imposing a structure on a pattern language. He points out that patterns follow a certain dualism by stating that patterns are elements and rules at the same time [Ale79, p.185]. Elements in the sense that a pattern is a thing in the real world and rules in the sense that a pattern describes the interplay of smaller patterns fulfilling the larger space that is made by the pattern itself [Ale79, p.247]. This structure implies a specific semantics represented in a pattern language, namely that there are larger and smaller patterns. This fundamental understanding of a pattern language is condensed in the following informal definition Definition 2.1.2 that lays the informal basis for the formalisms of this work.

Definition 2.1.2 (Alexandrian Pattern Language – informal) A pattern language according to Alexander is a directed acyclic graph comprising of patterns as nodes and relations between the patterns as edges. An edge directing from one pattern to another pattern specifies the semantics that the pattern at its start is a larger pattern than the pattern at its end. ■

This principle was transferred to the discipline of software design and architecture in the mid of the 1990s. The seminal work *Design Patterns: Elements of Reusable Object-oriented Software* [GHJV94] by Gamma et al. [GHJV94], *Software Patterns* by Coplien [Cop96] and *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* by Buschmann et al. [BMR+96] are among the first works that introduced patterns to the discipline of software engineering. They show that also IT systems and other artifacts of software engineering can be designed by applying patterns. Many more pattern collections and pattern languages have been published since then, which, over the years, cover a wide range of domains, such as Cloud Computing [FLR+14; YBB+22], Internet of Things [RBF+16], Microservices [Ric18], Service Meshes [DF23], Consensus Algorithms in Blockchains [SU23], or Patterns for Quantum Software Development [BBB+23; BBL+22; GBB+23; WBLS20; WBLS21; WBLV21]. However, the basic principles of patterns and pattern languages are still the same as introduced by Alexander. But in contrast to the domain of building architecture, the implementations of patterns in software engineering are not tangible things, but rather intangible things, such as computer programs that can be reproduced digitally.

Interestingly, although a multitude of patterns and pattern languages have been published in the domain of IT, another guiding principle of Alexander's work — that pattern languages are living systems, which should evolve over time and being adapted to networks of patterns (cf. [Ale79, p.341]) — is not very present in this domain. Therefore, in contrast to the integrative work by Alexander et al. [AIS77], which covers (i) the planning of cities and shaping the landscape, (ii) designing of districts of a city, (iii) designing buildings and their surrounding, and (iv) designing the interior of buildings to meet requirements for good and comfortable living, the domain of IT is still lacking a comprehensive pattern language that covers the design of IT systems in general, interweaving all the different viewpoints and languages. Thus, it is typically up to the reader to combine patterns from different pattern languages to cover the whole design of a specific IT system.

2.2 Formalization of Pattern Languages and Pattern Aggregation

Since the vision of this thesis is to reuse concrete solutions connected to patterns, it is necessary to investigate how patterns can be formalized and how the interplay of conceptual solutions can be expressed. Thereby, many approaches were proposed to formalize pattern languages and to support the aggregation of patterns. For example, Bayley and Zhu [BZ11; ZB15; Zhu14] propose a pattern composition algebra, which allows to compose patterns to form new patterns. This approach can be used in pattern-oriented software design to ensure the validity of pattern compositions at the model level via a pattern composition algebra for object-oriented design patterns. A pattern is characterized as points within a design space that satisfies particular criteria, while pattern compositions can be constructed by utilizing operators.

Mirnig and Tscheligi [MT14] characterize patterns as collections of interconnected components rather than standalone entities, aligning with the scientific understanding of complex issues involving numerous pertinent factors. Using this framework, established pattern languages can be formalized. Since the framework bases on set theory, it is generally applicable and can be used as a foundation for creating new pattern languages, irrespective of the specific domain for which they were originally conceived.

Bottoni et al. [BGd10] present a structured concept of how models can fulfill patterns and introduce mechanisms for suggesting model adaptions to ensure their alignment with the patterns. Their approach allows to formalize patterns based on graphs and category theory to enable the adaptation of domain-specific models by applying patterns. They also establish procedures for combining patterns and conducting conflict analysis in models. The presented approach's ability to function independently of specific languages renders it well-suited for integration into meta-modeling tools and its applicability within the field of Model-Driven Engineering. When modeling pattern-based designs, a tool can complete models via a palette of patterns and assures that the model complies with the used patterns.

Finally, Waseeb et al. [WKWV20] identify that patterns gain strength in combinations, thus, connecting them just informally in the text of patterns hinders their combined reuse. Thus, they investigate how text mining and natural language processing can be used to identify relationships between patterns from the pattern texts. They extend their work by organizing patterns and pattern languages in a semantic graph, which allows to identify relationships between patterns and pattern languages [WV23]. The resulting ontology is designed to facilitate the expression and dissemination of semantic knowledge among patterns. However, while the discussed approaches focus on pattern aggregation, none of them considers implementations of the patterns.

2.3 Pattern Application and Reuse of Implementations

Although pattern research lead to a vast amount of publications and works on new patterns in different domains, on methods about to capturing and authoring patterns [FBBL14; MD97], and on approaches targeting how to eventually form pattern languages, only few research is conducted focussing on the implementation of patterns. This is because many authors understand patterns only as design rules and guidelines rather than entities that can be mapped to actual implementations [Bos96]. This position must be questioned because already Christopher Alexander pointed out the dualism of patterns by stating that they are *elements* and *rules* [Ale79, p.185]. More concretely, he stresses that a pattern is "*at the same time a thing, which happens in the world, and the rule which tells us how to create that thing*" [Ale79, p.247], what was taken up by Coplien [Cop96] and confirmed for software patterns. Much more it can be stated that patterns immanently abstract the conceptual solution of a large number of implementations that have proven themselves. This is why it is common sense in the pattern community that patterns are *mined* from a reasonable amount of real world solutions [Cop96; Koh12; KU09; Rei13]. Thus, when authoring patterns, it is inevitable to provide evidence that a pattern represents a proven solution, which led to the *rule of three* [Cop96; KU09]. This rule means that a solution has to be detected in at least 3 real world scenarios in order to be accepted as *proven*, which means that patterns directly rely on real implementations. However, during the authoring of patterns, those implementations are typically not captured and at most are mentioned textually in pattern sections, such as *known uses* [GHJV94;

MD97]. Some pattern formats provide *examples* fitting to the intended audience [BHS07b], some provide code snippets in selected programming languages [Hen01; MD97], or via UML diagrams [BHS07a; BMR+96; GHJV94]. Still others follow the notion of diagrams as introduced by Christopher Alexander [Ale64; Ale79] and illustrate pattern implementations using sketches (e.g., [FLR+14; HW04; RBF+16]). While all those means support the understanding and rationale of a pattern they provide merely no sophisticated guidance and support for applying a pattern in real use cases at hand [FBB+14a; FBB+14b]. Even provided code snippets often cannot be reused directly as artifacts because they are printed in books.

In the area of software architecture, the above identified lack of coupling patterns and their concrete implementations is also persistent and referred to as the *traceability problem* [Bos96]. At the same time, it is identified that implementing patterns manually from scratch for individual use cases results in immense implementation overheads because a significant amount of boilerplate code has to be implemented each time [Bos96; Sou95].

Those findings led to approaches targeting to reflect patterns as entities in programming languages. Thereby, especially the *Design Patterns* by Gamma et al. [GHJV94], which capture proven design knowledge for object-oriented software have been objective to research aiming for the automated generation of pattern implementations. For example, Budinsky et al. [BFYV96] focus on the generation of C++ programming code for individual patterns. This is supported by a software tool that allows representing pattern sections on individual views such that the user can navigate sequentially through the sections of a pattern. One special section, the *code generation page*, provides a means for users to specify goals. This way the user can specify, for example, if only *declarations* in terms of C++ headers should be generated or if also implementations of whole classes shall be generated. Further, the code generation allows selecting between different implementation trade-offs. Those can be selected by the user, which results in different code. Thus, the tool allows selecting between different implementations of a pattern in C++ code. To generate code, they use the code generation language *COGENT* that allows specifying scripts and code templates with placeholders, which in turn enables to wire code fragments dynamically. They argue that the support of further object-oriented languages, such as SmallTalk, can be supported by implementing new code fragments for those languages. The work by Budinsky et al. [BFYV96] also motivates the need for reusing pattern implementations very well because they show how implementation efforts can be reduced once pattern implementations can be generated. They also show that it is important to address the need to reflect different trade-offs in pattern implementations, i.e., they identify the demand to select pattern implementations that fit best to the situation a programmer is faced with. However, since this work focusses only on the *Design Patterns* by Gamma et al. [GHJV94], which is just a catalog of 23 patterns rather than a pattern language, they do not cover the combined application of patterns. Furthermore, the capabilities of the software tool are tailored to the application of the *Design Patterns*, e.g., the tool only supports the pattern format on which they are based (cf. [GHJV94]). Thus, the presented approach of automatic code generation is tightly coupled to the *Design Patterns* through the provided tooling.

Bosch [Bos96] argues in a similar direction. He points out that patterns provide concepts, which are widely used concepts by experienced software engineers in their daily work. Thus, he identifies the requirement to reflect them as constructs in programming languages in order to support the reuse of proven solutions in software engineering. To overcome the lack of patterns as constructs in programming languages, Bosch [Bos96] introduces the layered object model *LayOM*, which enables using layers to represent design patterns. By this means he shows that patterns can be put on the implementation level, which allows automating their application. This is realized by translating LayOM into C++ code. Thereby, the specified patterns in LayOM are applied or respectively implemented automatically, which couples them with corresponding implementations in C++ code. More specifically, implementations of the patterns can be directly reused through the automated generation, which eases their repeated usage in different use cases.

Another approach, which addresses the application of the *Design Patterns* [GHJV94] in the programming language Java, is presented by Santos and Coelho [SC16]. They introduce Java annotations to be used for the automated implementation of *Design Patterns*. For effecting this, they extended Lombok[1], which is a Java library that allows defining arbitrary Java annotations. Those annotations are then used to adapt the abstract syntax tree of Java programs at compile time. This way, they introduce *Design Patterns* as Java annotations that can be used by programmers to enrich their code. For instance, one can annotate classes or methods by *Singleton*, *Visitor*, *Decorator*, or *Observer* to specify that code templates are injected during compile time, which directly corresponds to generating implementations of the respective patterns automatically. Thereby, it is even possible to specify the interplay of different interfaces, classes, and methods to realize implementations of more complex patterns, such as the *Visitor* pattern [GHJV94, p.331]. The introduced Java extensions focus on the mentioned *Design Patterns*, which makes the concept of patterns directly usable on the level of a programming language. However, while this approach helps to automatically generate implementations of the mentioned *Design Patterns*, it is limited to the programming language Java and the concept of annotations because of Lombok. Thus, the approach can only be transferred to programming languages that support annotations.

The idea of using patterns to capture proven designs was also brought to the field of Model-driven Architecture (MDA). Krleža and Fertalj [KF14] show how patterns can be used to guide modelers to reuse proven solutions in system designs. Thereby, patterns help to limit the arbitrary design space in Computational Independent Models (CIM), Platform Independent Models (PIM), and Platform Specific Models (PSM) to proven designs. Furthermore, by specifying proper transformation rules, patterns can be translated from abstract representations in CIM via PIM towards representations close to the level of execution in PSM. However, while the approach by Krleža and Fertalj [KF14] focusses on supporting users to create con-

[1] https://projectlombok.org

sistent MDA models it lacks the possibility to purposefully reuse pattern implementations in specific technologies. This could be maintained by introducing further transformation rules enabling to generate executable artifacts from PSMs but this would introduce another layer of complexity. Instead of reusing pattern implementations directly they had to be *implemented* again in the form of transformation rules.

The PALMA-framework by Breitenbücher [Bre16] describes domain-specific concepts similar to the idea of concrete solutions and their aggregation to overall solutions as introduced in this work. This framework focusses on the domain of application management and introduces a means to manage applications based on management patterns [FLR+13]. An instance model describes the current state of an application with respect to its configuration and runtime parameters [Bre16, p.103ff]. An automated management pattern can then be applied to transform an instance model into a declarative management model [BBKL13; BBKL14]. This resulting model contains so-called management annotations that describe the solution of the applied management pattern declaratively. To execute such a model, the framework enables to transform it into an imperative process model by composing so-called Management Planlets [BBKL13; BBKL14]. Thus, the resulting imperative process model is a concrete solution of the applied management pattern for a certain type of application, which can be reused to manage other deployment instances of this application without generating the process model again.

Bibartiu et al. [BDR21] and Bibartiu [Bib23] present an approach that allows to use patterns as modeling elements in cloud architectures. Thereby, cloud computing patterns are placeholders that can be systematically refined to concrete service offerings of cloud vendors. The patterns are connected to other elements of the architecture model via message sequence charts that specify their relationship. Harzenetter et al. [HBF+18a] show how patterns can be used in deployment models to represent the essential semantics of components independently from a specific technology. They also introduce a method for the automatic identification of design patterns within declarative deployment models [HBF+21]. This helps to prevent delving into technical specifics concerning components, relationships, and configurations and allows to focus on pattern-based deployment modeling by introducing patterns as modeling elements for deployment models [HBF+18a; HBF+20]. The approach enables to refine the placeholders later with specific components of the deployment model to execute the deployment, which corresponds to replacing patterns with concrete solutions as introduced in this work [HBF+18a]. Thus, it shows a domain-specific adaptation and realization of concrete solutions.

Scheibler [Sch10] presents a method for the automated generation of integration solutions based on patterns. The method is based on the *Enterprise Integration Patterns* by Hohpe and Woolf [HW04]. The method enables to model integration solutions based on parameterizable patterns and to generate executable integration solutions from the model. Thus, it shows a domain-specific realization of design models as introduced domain-independently in this work. Further, a meta model for parameterizable integration patterns allows to specify pattern semantics in a technology-agnostic way, such that algorithms can generate implementations of the patterns for different integration technologies. In this process, pattern implementations are not directly reused, but are generated automatically and composed to overall integration solutions. Thus, Scheibler [Sch10] provides a complementary approach to the one presented in this work, which enables the generation of holistic solutions from patterns. But by focussing on the generation of integration solutions, it does not provide a domain-independent framework and is, therefore, limited to the domain of integration solutions and the *Enterprise Integration Patterns* by Hohpe and Woolf [HW04].

2.4 Patterns and Frameworks

Since new technologies and frameworks origin and evolve over time, the list of frameworks mentioned here is of course not exhaustive. But it is intended to give an impression of how patterns relate to frameworks, how they are used there, and how they can be applied using frameworks. Frameworks are often based on the conceptual knowledge provided by patterns of the specific domain the framework addresses. They provide good practices, conceptual designs, and solutions from a specific area for a specific area. Thereby, these frameworks often rely on conceptual solutions of patterns and pattern languages and provide a blueprint of their interplay for implementations in a specific technology. Those skeletons can be adapted by users to implement required functionalities, whereby the good design as captured by the underlying patterns is preserved. As denoted by the name *framework*, they build on proven design principles of patterns and guide the development of applications.

For example, the Spring Framework[2] heavily uses the *Inversion of Control* pattern to wire different components of a Spring application [Pra09; RJ07]. Besides, it uses many other patterns from the *Design Patterns*, such as *Singleton* for Bean instantiation or *Proxy* to generate generic implementations for data access components, to name just a few [Alb20]. Thus, Spring incorporates several patterns to a framework that can be used to implement applications. Thereby, the patterns incorporated in the framework automatically become part of the applications themselves, i.e., by using the framework they get automatically applied. However, since the framework predefines the wiring of the patterns there exists typically no freedom for developers to alter their interplay.

Other frameworks, such as Apache Camel [3] or Spring Integration [4], focus on the integration of applications and components and support the *Enterprise Integration Patterns* by Hohpe and Woolf [HW04]. In the case of Apache Camel, even evolving patterns, such as the *Microservice Patterns* by Richardson [Ric18], are supported by the latest releases. In contrast to Spring, Apache Camel and Spring Integration support to create integration

[2]https://spring.io

[3]https://camel.apache.org

[4]https://spring.io/projects/spring-integration

solutions based on those patterns. They do not just allow the implicit usage of patterns through the blueprint offered by the framework. They rather allow composing the mentioned patterns as required to design and implement applications. The patterns are directly usable concepts of Apache Camel and Spring Integration, i.e., a developer can use them as programming constructs. For instance, Apache Camel provides means to specify the usage of a pattern directly in programming code via a domain-specific language based on Java [5]. Thereby, multiple of the above-mentioned patterns can be composed to implement, e.g., a sequence of computation steps required to integrate different applications. For example, if an application may only receive certain messages and the data must be transformed into the format of the target system, then a *Message Filter* [HW04, p.237] can ensure that only intended messages are forwarded, followed by a *Message Translator* [HW04, p.85], which is capable of transforming the transmitted data to the target format.

Bulka [Bul02] discusses the automation of design patterns from a generative perspective. He identifies and describes three automation levels for pattern application, starting from *Single Templates* via *Parameterized Pattern Templates* to *Intelligent Patterns*. While the first category represent static code snippets, the second already provides user guidance to select proper implementations. Finally, the last category identifies approaches that enable to automatically integrate components and structures into UML diagrams while maintaining the validity of the UML at the same time. Those enable to implement *Wizards* into IDEs that support the pattern implementation process. However, the discussed *Wizard* approach is limited to UML and does not cover the implementation of patterns in other programming languages and beyond.

Other works by Bosch [Bos98a; Bos98b] focus on interweaving patterns and implementations of them into programming languages and frameworks to support the reuse of proven solutions. Thereby, patterns are represented

[5] https://camel.apache.org/manual/latest/enterprise-integration-patterns.html

as language constructs to enable their reuse directly. As a result, the object-oriented design patterns [GHJV94] are represented as language constructs in object-oriented programming languages.

2.5 Pattern Libraries and Pattern Repositories

Since patterns and pattern languages are basically about providing proven solution knowledge in a reusable way, in the following it will be considered how patterns can typically be retrieved, adapted, and enriched by readers. Thereby, especially the limitations regarding pattern reuse are investigated. In general, there exist three approaches when it comes to authoring and publishing pattern languages. The first and most straight forward is the so-called *pattern guru approach* [Rei13] combined with publishing patterns and pattern languages in research papers and text books. Back in the late 1980s and 1990s, when pattern research in IT was in its infancy, few authors documented their knowledge about proven design principles for software designs via patterns. Although discussion-centric conferences, such as the *Pattern Languages of Programs* (PLoP) conferences of the hillside group [6], have emerged quickly back then, patterns were typically authored by only few people if not even by individual authors alone. Patterns were then discussed at research conferences gaining only few feedback loops from a community to improve them. Once the patterns were published in papers and text books they got fixed due to the used media, i.e., adapting them to new contexts and domains was not supported by any system. However, since most of the relevant pattern conferences, such as the PLoP conferences of the hillside group, the IARIA Patterns [7], or the PURPLSOC [8] typically publish the research papers about patterns as open access, their online proceedings can be seen as digital pattern

[6]https://www.hillside.net/conferences
[7]https://www.iaria.org/conferences/PATTERNS.html
[8]https://www.donau-uni.ac.at/en/research/project/U7_PROJEKT_4294969506

repositories. Due to the characteristics of conference proceedings, the patterns and pattern languages are just contained in the published papers and are, therefore, not first level entities, which makes it difficult to keep the overview of all. Nevertheless, digital conference proceedings must still be seen as the hugest library of patterns among different disciplines.

The second approach is publishing whole pattern languages via static web pages. This is an approach that can be seen for pattern languages, which are authored by one to few authors who are especially interested in spreading the pattern knowledge widely. For instance, the *Cloud Computing Patterns* [9] by Fehling et al. [FLR+14], the *Enterprise Integration Patterns*[10] by Hohpe and Woolf [HW04], the *Pattern Language for Microservices*[11] by Richardson [Ric18], the *Internet of Things Patterns*[12] by Reinfurt et al. [RBF+16], the *Cloud Design Patterns*[13] by Amazon Web Services [Ama13], or the *Pattern Library for Interaction Design*[14] by Welie and Veer [WV03] are examples for this category. Here, the individual patterns are treated as first level entities and even the pattern languages themselves are essential parts typically represented as visualized graph structures or hyperlinks between patterns. Thereby, patterns are often assigned with an icon, which, in combination with the links between the patterns, eases the study and application of the pattern languages. However, the patterns are published typically once and the underlying systems are only capable of serving them as web pages. Community-driven features, such as collaboration on authoring and evolvement of the patterns or discussions about actual applications of the patterns and their transfer to new use cases, are not supported. Therefore, more far-reaching aspects such as connecting concrete solutions with patterns are not supported as well.

[9] https://www.cloudcomputingpatterns.org
[10] https://www.enterpriseintegrationpatterns.com
[11] https://microservices.io/patterns/index.html
[12] http://internetofthingspatterns.com
[13] http://en.clouddesignpattern.org/index.php/Main_Page
[14] http://www.welie.com

The third approach is based on the idea of capturing and managing pattern languages via IT-supported repositories. A repository is, thereby, an application system that allows to capture patterns, connect them to pattern languages, structure the pattern format and enable the collaborative work on the captured content. The technically most straight forward form is the approach by Cunningham and Mehaffy [CM13]. They show how a wiki can be used to author a pattern language as linked wiki pages. Thereby, the wiki functionality provides collaboration capabilities to formulate and advance patterns together. However, such a system does not automatically support the authoring of patterns in a common format. For instance, the canonical format of a pattern language cannot be defined and it is up to the users to apply a common format. Since the wiki system is based on wiki pages, users have to ensure that patterns become first level entities by putting them on separate wiki pages. The pattern language then emerges by linking to other wiki pages and by introducing cross-cutting wiki pages that describe the pattern language. Thus, it is up to the users to transfer the principle of wiki pages to patterns and pattern languages in a way that they are adequately represented.

More sophisticated approaches are pattern repositories that are specifically designed to capture patterns and pattern languages, such as presented by Fehling et al. [FBFL15], van Heesch [Hee14], Pavlič et al. [PHPR09], and Reiners [Rei13]. For example the PatternPedia approach by Fehling et al. [FBFL15] enhances the wiki approach by semantic extensions, which enable to create meta structures that allow attaching meta information to wiki pages to form pattern languages as constructs. Others, such as the approach by Reiners [Rei13] or Pavlič et al. [PHPR09], focus on representing patterns and pattern languages as first level entities in a content management system or ontology, respectively. This allows them to provide canonical formats for patterns and pattern languages, which can be used to author patterns and pattern languages in a common format. The presented work contributed and influenced the conceptual extension of the PatternPedia approach by Leymann and Barzen [LB21]. They introduce the metaphor of a PatternAtlas, which is enables to link patterns from different pattern languages and even allows to create pattern views among them, which are selections of patterns from different pattern languages that are relevant for a specific use case [WBB+20]. Still, this practice shows that pattern repositories are often focussed to capture just one pattern language. In some cases, this is because they are not feasible to distinguish different pattern languages because they do not provide a suitable domain model that contains the entity *pattern language*.

Moreover, many pattern languages are published on the web as structured and linked web pages [cf. among others Ama13; Feh17; Fow03; Hoh17; Rei17; Ric20]. Thus, they can be seen as *read-only* pattern repositories. While they enable to retrieve the patterns and navigating through the pattern language by means of hyperlinks, they are limited to viewing the contents of the patterns because collaboration and adaptation of contents is not supported. Even the first wiki on pattern languages by Ward Cunningham [Cun], the founder of *the Wiki*, falls under this category for some time now. In all cases, the repos cover just patterns and pattern languages but lack for keeping implementations of the patterns accessible.

2.6 Limits in Pattern Application

Throughout this chapter, the limitations and gaps in the state of the art of pattern language application were discussed. In summary, the following deficiencies can be summarized, which drive the research challenges of this work as presented in Chapter 1 as well as the requirements for the methodical framework presented in Chapter 3.

Deficit 1 (Lack of Solution Reusability) Current pattern approaches typically do not consider the systematic reuse of concrete solutions. This poses a significant barrier for users who seek to reuse and integrate previous implementations of patterns. This leads to recreating pattern implementations over and over again when a pattern has to be applied.

Deficit 2 (Implicit Reuse and Aggregation) While there is some capability for implicit and technology-coupled reuse especially in programming frameworks and domain-specific languages, there is no general theory and domain-independent approach that aids the reuse of pattern implementations. Although designing solutions with patterns is a common approach in multiple domains, projecting these designs to the level of already present pattern implementations is lacking. Despite some programming frameworks and programming languages incorporate the aggregation of pattern implementations, they are limited to those pattern implementations that are an inherent part of the frameworks themselves. Thus they lack the capability to integrate pattern implementations from different pattern languages, which are not yet part of the frameworks' structures.

Deficit 3 (Absence of Automation) The guidance and automation of aggregating pattern implementations is limited to few domain-specific approaches. There is no general approach that allows to organize the aggregation logic for aggregating pattern implementations in a systematic and domain-independent way.

Deficit 4 (Coupling to specific Domains) The state of the art approaches for pattern application are typically tailored to specific domains. Thus, they are often limited to the patterns of the respective domain and do not consider to combine patterns from different pattern languages. This is also often reflected by limited capabilities of pattern repositories to capture and interweave different pattern languages, which narrows their usability to specific domains.

Deficit 5 (Organizational Limitations) While pattern repositories are used and discussed for years in many domains, repositories for systematically collecting, reusing, and linking pattern implementations with patterns, although conceptually described, are missing in practice. Further, there are no systematic domain-independent approaches that facilitate the aggregation of pattern implementations according to pattern-based solution designs. Thus, the aggregation of pattern implementations is typically done manually, which leads to a lack of systematic approaches for organizing the aggregation of pattern implementations. Moreover, repositories, as they currently exist or being discussed in research, mostly focus on a specific pattern language. Thereby, they represent only the structure of a specific pattern language, but they fall short in facilitating combinations across different pattern languages. Again, although conceptually described by the PatternAtlas approach by Leymann and Barzen [LB21], in practice, this still leaves different pattern languages disconnected from each other because the interlinking of their patterns is not elaborated and captured in the repository.

The research challenges and contributions outlined in Section 1.1 of this work tackle those deficiencies, while requirements to a general methodical framework are deduced in Section 3.2.

Chapter 3

The EINSTEIN-Method

In this chapter, the methodical basis for the application of patterns by means of concrete solutions is introduced. The underlying method is called EINSTEIN-Method, which is an acronym for patt**e**rn appl**i**catio**n** ba**s**ed on concre**te** solut**i**o**n**s. The EINSTEIN-Method represents Contribution 1 of this thesis and describes the interplay of *Designers*, who use patterns as modeling elements to design conceptual solutions and respective *Implementers*, realizing the specified models optimally by reusing as many existing concrete solutions as possible. Thus, the method extends the idea to reuse proven solutions via patterns in designs to actual use case and environment-specific implementations. The method is not bound to already existing procedure models or methods but can be applied as a supporting instrument whenever conceptual designs are worked out, which then need to be implemented. Thereby, the EINSTEIN-Method enables to resort on proven solutions in the design utilizing patterns as well as in the implementation by concrete solutions, respectively.

Therefore, in Section 3.1 the involved roles of the *Designer* and the *Implementer* as used in the course of this thesis are introduced. Based on those roles, requirements and challenges the EINSTEIN-Method has to tackle are pointed out in Section 3.2. Then, the EINSTEIN-Method is explained in detail in Section 3.3, whereas their embedding in different domains and procedure models is discussed in Section 3.4. Finally, the chapter is concluded in Section 3.5.

3.1 Roles participating in the EINSTEIN-Method

All approaches introduced in this work rely on the usage of patterns to create designs and the systematic reuse of concrete solutions. Therefore, the roles of a Designer and an Implementer are distinguished, which both participate in the EINSTEIN-Method. The two roles focus on different aspects with regard to the usage of patterns and are introduced in the following in a domain-agnostic manner. A discussion how to implement the EINSTEIN-Method and also how to the presented roles can be fulfilled in different domains is given and exemplified by the roles of architects and software engineers in Section 3.4. The Designer addresses the analysis of conceptual requirements, i.e., overall qualities an intended system has to fulfill. For instance, in the discipline of software architecture, these qualities are typically referred to as quality attributes [cf. BCK03; CKK11] of a system or application architecture. Thus, the Designer is responsible to analyze and understand functional and non-functional requirements and translate them into proper system architectures. To elaborate a proper solution and to make appropriate decisions, the Designer typically harks back to patterns and pattern languages that provide knowledge about proven solution concepts to solve his or her problems at hand. The requirements, which are considered in the conceptual solution design by the Designer, are typically technology- and implementation-agnostic. Hence, the Designer does not deal with technology- or implementation-specific questions and details, but relies on abstract solution concepts.

Such realization details are addressed during the implementation of the conceptual solution by means of specific technologies and artifacts. Here the Implementer steps in. The Implementer takes over the conceptual solution modeled via patterns and realizes a working solution guided by the best practices as described by the patterns and their interplay in the conceptual solution model. Concretely, this means that the Implementer has to come up with implementations of the patterns contained in the conceptual solution and has to integrate them into a real solution. Thus, while the Designer uses patterns to describe the conceptual solution, the Implementer realizes these patterns and implements their solutions. Especially, the Implementer can benefit a lot from concrete solutions that are linked with patterns. By searching and reusing them directly, they can save efforts once patterns have to be applied, because the implementations do not have to be reworked every time a pattern is contained in a conceptual solution developed by a Designer. In fact, the Implementer can hark back to the existing concrete solutions to reuse and adapt them to meet the requirements of the use case at hand and to refine them to provide the needed business functionality.

3.2 General Prerequisites and Requirements

In the following, prerequisites are discussed that lay out general conditions to apply the EINSTEIN-Method in practice. Subsequently, requirements for the method are discussed. These should ensure that the EINSTEIN-Method provides a framework that enables the vision presented in this work to be applied in practice. The specific requirements are identified by numbers. These numbers are referred to in the introduction and discussion of the different steps of the EINSTEIN-Method in Section 3.3 to indicate where and how they are realized.

The general prerequisite that lays the foundation of the EINSTEIN-Method is the concept of patterns and pattern languages itself. Specifically, the EINSTEIN-Method is based on the idea to use *patterns as modeling ele-*

ments that help to design good solutions. Thus, the EINSTEIN-Method relies on the assumption that patterns and pattern languages provide proven solution knowledge for frequently recurring problems as intended and defined originally by Christopher Alexander et al. [AIS77]. Today, this is ensured by pattern authoring processes [FEL+12; MD97] and the fact that patterns are often discussed and quality-assured in writers workshops at pattern conferences etc., such as the PLoP-conferences[1][2]. However, the EINSTEIN-Method is not immune to bad designs based on patterns because the design process lays still in the hands of a Designer. It rather aims to project pattern-based designs to the level of their implementations.

Further, since patterns are the core design elements, the EINSTEIN-Method is only applicable to domains and approaches where the concept of patterns is already established as means to capture and reuse good design knowledge, such as in many fields in IT as already mentioned in Chapter 1. Otherwise, a body of knowledge in the form of patterns and pattern languages has to be elaborated beforehand. Thereby it is important that also the typical interplay of patterns is reflected in the form of pattern languages. In contrast to unstructured pattern catalogs, pattern languages already reflect typical combinations of patterns and, therefore, can be used as blue prints for the design of conceptual solutions by eliminating the arbitrariness of pattern combinations due to the captured pattern interplays. In the wide field of the various disciplines of computer science, there exist many different pattern languages, which is why this essential prerequisite is fulfilled for the areas investigated in this thesis. Besides this general prerequisite the following requirements have to be fulfilled by the method in order to overcome the deficits identified in Section 2.6.

Requirement 1 (Patterns as Modeling Elements) The general prerequisite that patterns can be used as design elements as formulated above is itself a requirement the EINSTEIN-Method has to fulfill. The assumption

[1] https://europlop.net

[2] http://www.hillside.net/plop

that patterns capture proven conceptual solution knowledge and represent *things and rules how to create that thing* (cf.[Ale79, p.247]) is thereby the core idea that allows to use them as modeling elements for conceptual solution designs. Especially in the field of application architectures, patterns play a vital role in specifying architecture models [AZ05; BMR+96; FLR+14; HW04]. Thus, the EINSTEIN-Method has to ensure that patterns and their interplay in specific use cases at hand can be captured as the conceptual baseline for the implementation of solutions. This requirement addresses Deficit 1 because using patterns as modeling elements for a conceptual solution design is the foundation, such that implementations of them can be systematically reused for implementing the conceptual solution design.

Requirement 2 (Reuse of Existing Pattern Implementations) In order to avoid reworking pattern implementations each time a pattern has to be applied in a specific use case, a user has to be able to reuse already existing implementations. The EINSTEIN-Method has to assure that the applicability of existing pattern implementations for use cases at hand can be identified. This requirement addresses Deficit 1 and Deficit 2 because it ensures that existing pattern implementations can be reused in the implementation of conceptual solution designs. Furthermore, it incorporates Deficit 5, as it also implies proper tools to support the EINSTEIN-Method.

Requirement 3 (Tailoring of Existing Pattern Implementations to specific Use Cases) On the one hand, the EINSTEIN-Method has to aim to reuse pattern implementations systematically. However, on the other hand, the EINSTEIN-Method also has to enable use case-specific adaptations of the implementations for new use cases. Therefore, it is inevitable in many cases that the existing pattern implementations to be reused have to be enriched and adapted, for example by manipulating the business logic. Thus, the EINSTEIN-Method as a framework has to allow adaptations of existing pattern implementations and their aggregations. Further, the selection of existing pattern implementations has to consider requirements, such as the need for specific technologies or configurations. This requirement tackles the aspect of Deficit 2 that pattern implementations are often interwoven into frameworks and cannot easily be adapted.

Requirement 4 (Aggregation of Existing Pattern Implementations) Requirement 1 focusses to provide conceptual solution designs by means of interplays of patterns. However, in order to support the systematic reuse of pattern implementations also this interplay has to be propagated to the implementation level. Therefore, the EINSTEIN-Method has to support the aggregation of existing pattern implementations. Moreover, the EINSTEIN-Method has to ensure that the aggregation of implementations can be executed automatically if technically possible. Otherwise, proper guidance has to be provided for Implementers, which implies that aggregation knowledge has to be captured. This requirement extends Requirement 2 by addressing the aggregation of pattern implementations. It further tackles both, Deficit 3 and Deficit 5, by requiring guidance for the automated aggregation of pattern implementations.

Requirement 5 (Openness to new Technologies) Patterns aim to provide proven solutions to recurring problems. Thus, they provide a viewpoint on solution principles that have proven to be efficient in the past. Through the abstraction of such principles from specific technologies, patterns are sometimes seen to capture *timeless* ideas and concepts [AIS77; Ale79; AZ05; Rei13]. However, especially in the field of IT, emerging technologies are a driver of innovation. Thus, the EINSTEIN-Method has to be open for emerging technologies, which require new specific implementations of patterns and their aggregation. This requirement addresses Deficit 4 because it implies that new emerging pattern languages must be interwoven with already existing ones, which must also be reflected in tools supporting the EINSTEIN-Method.

Requirement 6 (Domain-independent Applicability) Although the general prerequisite to apply the EINSTEIN-Method is that patterns and pattern languages have to exist for the application domain, the method has to be applicable independently from specific domains. The approach in its general form must be tailored to the general concept of patterns and pattern languages rather than specific structures of pattern languages that reflect interplays in a specific domain. Thus, the general approach framed by the EINSTEIN-Method has to be technology- and environment-agnostic so that it can be adapted and implemented for different specific domains. Building upon Requirement 5 this requirement addresses Deficit 5 as well by imposing the support of managing patterns, pattern languages, and pattern implementations from different domains.

Requirement 7 (Compatibility to Procedure Models and Frameworks) In practice, there are many different procedure models in place that are optimized for different purposes of use (e.g., waterfall, Scrum and spiral model etc.), software development approaches (e.g., model-driven architecture), or even ways of thinking (e.g., agile methods). Thus, the EINSTEIN-Method must not replace a present procedure model but be in place to support activities in there. Therein, it has to focus on translating pattern-based design to aggregated implementations. This requirement is general and assures that the EINSTEIN-Method does not incorporate constraints and limitations imposed by a specific procedure model.

3.3 Steps of the EINSTEIN-Method

In this section, the vision of this thesis (cf. Figure 1.2) is translated into a methodical framework. Thereby, core concepts are introduced and informally defined. First, the EINSTEIN-Method is presented in a comprehensive overview, and then the individual steps of the method are detailed in separate sections. The method is shown in Figure 3.1.

The method is designed to transform abstract pattern-based solutions into directly usable solutions. For instance, in the context of software development, this means that the EINSTEIN-Method can be used to transform a conceptual solution design into a concrete software implementation. To achieve this, 7 steps are defined as shown in Figure 3.1. The first two steps – *Design Conceptual Solution* and *Refine Conceptual Solution* – are conducted by the Designer. During these steps the Designer uses patterns as the main design elements to create an abstract model of the intended solution, which is called a *Pattern-based Design Model* or *Design Model* in short. Thereby, the first step focusses on modelling an initial design model mostly independent from any technological or implementation specific considerations, which is illustrated by the modelled graph of connected patterns *A*, *B*, *C*, and *X* as rounded squares attached to the arrow directing from Step 1 to Step 2. The second step takes such technology- or implementation-specific constraints into account by replacing initially used technology-agnostic patterns in the design model with those that already describe technology- or implementation-specific solution concepts. This is exemplified by the refinement of pattern *C* into pattern *C'* as the outcome of Step 2. The basis for both steps are patterns authored and available in a pattern repository representing a reusable body of knowledge.

Starting from Step 3, the Implementer takes over and turns the design model via Step 4 to 6 into an *Aggregated Concrete Solution* that can finally be used. First of all, in Step 3 the Implementer selects concrete solutions, i.e., concrete implementations of the patterns stored in a *Solution Repository* to avoid reimplementing the patterns over and over again. This is indicated in the outcome of Step 3 where the interplay of patterns is replaced by an interplay of concrete solutions illustrated as circles. In particular, the selection of appropriate concrete solutions that meet technical- and implementation-specific requirements at hand can be supported by automation (cf. Section 3.3.3 and Chapter 5), as illustrated by the interlocking gears in Figure 3.1. The EINSTEIN-Method takes into account that concrete solutions are subject to a much faster life cycle decay and become irrelevant much faster than patterns. This is due to the fact that patterns prepare solution essences, i.e., contain timeless knowledge, while concrete solutions represent implementations of these concepts by means of technologies – and these technologies change more often in many domains. For example, concrete solutions implemented in a specific programming lan-

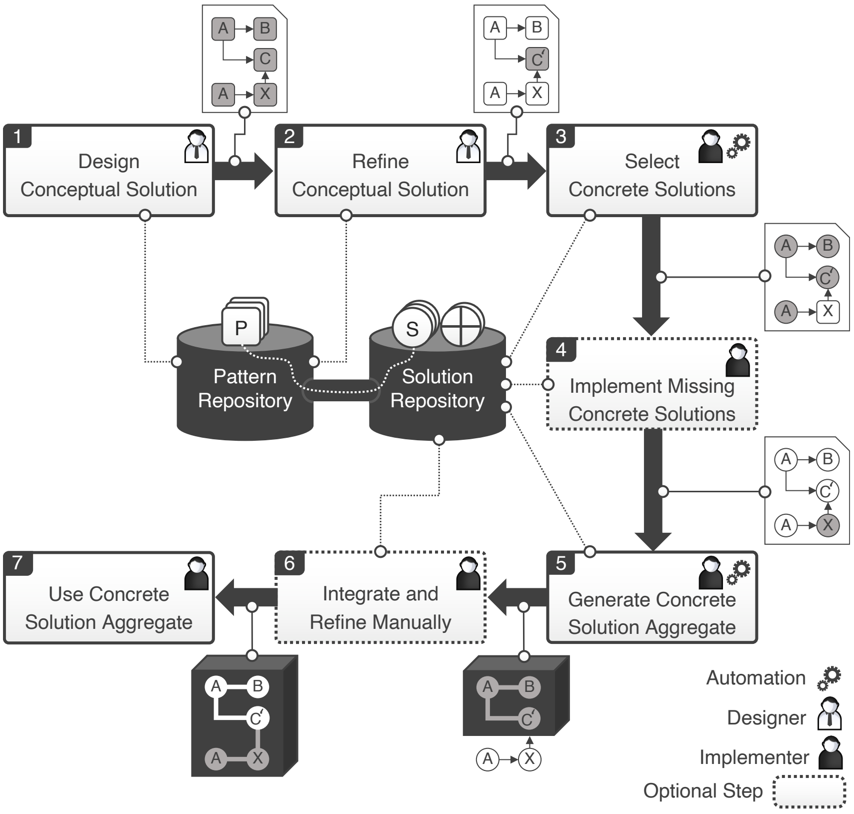

1 Design Conceptual Solution
2 Refine Conceptual Solution
3 Select Concrete Solutions
4 Implement Missing Concrete Solutions
5 Generate Concrete Solution Aggregate
6 Integrate and Refine Manually
7 Use Concrete Solution Aggregate
Pattern Repository
Solution Repository
P
S
A
B
C
C′
X
Automation
Designer
Implementer
Optional Step

Figure 3.1: Conceptual overview of the EINSTEIN-Method

guage are outdated once the programming language itself is no longer used in practice. Thus, continuous change and emerging innovations towards new technologies make it necessary for concrete solutions to be updated in the solution repository from time to time. Moreover, of course there could also patterns be used in the design model, for which no concrete solutions are available, so that they must be implemented manually. This is what the EINSTEIN-Method supports by the optional Step 4. To pick up on the example outlined in Figure 3.1, where, for example, no suitable concrete solution could be found in the solution repository for pattern *X* in Step 3, the missing concrete solution for pattern *X* is implemented by the Implementer in Step 4 and then stored in the solution repository for reuse. This is also shown in the result of Step 4, where a concrete solution is now available for pattern *X* as well. Thus, the method also opens up application areas and technologies for which solution repositories do not yet exist and, accordingly, concrete solutions do not yet exist for systematic reuse. By applying the method, Step 4 assures that the solution repository is successively filled and a growing pool of concrete solutions can be accessed for new use cases.

Of course, the concrete solutions only result in a functioning whole once they are integrated. This is achieved in Step 5 through their aggregation. The aggregation of concrete solutions is not arbitrary but based on the conceptual interplay of patterns as expressed in pattern languages and the design model at hand. Thus, the principles of the interaction of patterns in the design model and pattern languages must also be projected to the level of concrete solutions. In this work, the approaches of *Solution Languages*, *Aggregation Operators*, and *Solution Algebras* (cf. Chapter 6 and Chapter 7) are presented to systematically support and automate this step for Implementers. In this process, the previously isolated concrete solutions are integrated by means of *Aggregation Operators* ($\oplus$) to form a concrete solution that functions as a whole. This is shown in Figure 3.1 by the connected concrete solutions as part of a coherent block as a result of Step 5. If aggregation operators for the selected concrete solutions do not yet exist in the solution repository, aggregation must be performed manually by the Implementer, what is intended in the optional Step 6.

Finally, the concrete solution aggregate is ready for use as indicated in Step 7 of the EINSTEIN-Method. In many cases, the generated solution aggregate still needs to be adapted for the intended use and function. This is essentially due to the fact that the concrete solution generated up to that point is based on the interaction of general solution principles from patterns. Of course, these do not yet specifically take into account the semantics of the actual use case to be realized, i.e, the business logic. Using the

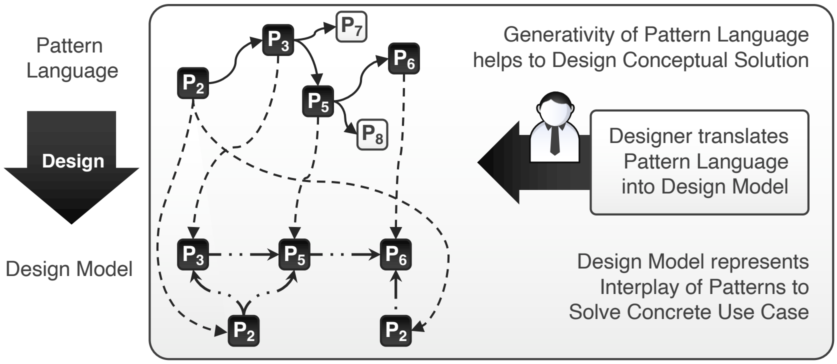

Pattern Language
Design
Design Model
P_2
P_3
P_7
P_6
P_5
P_8
Generativity of Pattern Language helps to Design Conceptual Solution
Designer translates Pattern Language into Design Model
Design Model represents Interplay of Patterns to Solve Concrete Use Case

Figure 3.2: A selection of patterns is translated into a design model

example of program code it can be stated that patterns and, therefore, also concrete solutions are agnostic about business logic. Accordingly, this must still be finally incorporated into the solution aggregate. Nevertheless, it must be emphasized that the EINSTEIN-Method allows the Implementer to concentrate on exactly these important aspects because the core of the implementation of the use case can be generated in essential parts on the basis of reusable concrete solutions.

Up to this point, the method has been explained in general terms and the individual method steps have been explained in an abstract way. Therefore, all method steps will be explained and discussed in detail in the following.

3.3.1 Step 1: Design Conceptual Solution

The first step of the EINSTEIN-Method provides guidance to derive solution designs from a pattern language taking into account how pattern languages provide design knowledge and how designs for concrete problems at hand have to be layed out. Pattern languages show the interplay of patterns in a general way. Thereby, the patterns of a pattern language describe aspects of a domain that have to be interwoven to overall designs for solving recurring problems. A pattern language itself describes the dependencies between the patterns which can be grasped as rules, guidance, and best practices about how to combine the patterns to design good designs solving concrete problems at hand. However, there is a difference between a pattern language and designs derived from it. A pattern language forms a body of knowledge in a particular field that captures good design. A design, in turn, applies the patterns to a concrete use case. That is, patterns can appear multiple times in the design. For example, when designing a cloud application, the business logic is typically encapsulated in multiple independent *Processing Components* [FLR+14, p.180ff]. When designing multifamily houses, an architect has to place multiple *Main Entrances* [AIS77, p.540ff] in a building plan. When arranging a scene in a western movie, a costume designer often has to establish multiple versions of the *Wild West Outlaw* [SBLE12]. Thus, if a Designer wants to use patterns as model elements for the design solving concrete use cases at hand, they must accordingly include one and the same pattern several times in the model. Note, that *composite patterns*, such as *Content Distribution Network* [FLR+14, p.300ff] or *Scatter-Gather* [HW04, p.297ff], which explain the interplay of patterns solving more complex problems in combination, do not deal with the above. Because these interplays have proven to be good solutions they are promoted to patterns themselves and can, therefore, also be used multiple times in the model.

In this work, the resulting design is a model using patterns as the main modeling elements and is called pattern-based design model, or design model in short. The Designer navigates through the pattern language and chooses suitable patterns and arranges them in a way solving the problem at hand guided by the pattern language. The act of designing, thus, corresponds to mapping a pattern language onto a design model solving a concrete problem. This is shown in Figure 3.2. There, e.g., pattern P_2 from the pattern language at the top of the figure is used twice in the design model at the bottom. Further, the patterns P_7 and P_8 are not used at all by the Designer in the resulting Design Model. The Designer sometimes has to resolve interdependencies between patterns in a pattern language manually, e.g., if they are stated not specifically but conceptually exist transitively between a series of patterns. This scenario can occur if a pattern language contains patterns, which make coarse-grained patterns more specific to a certain solution principle or setup.

Definition 3.3.1 (Pattern-based Design Model – informal) A *Pattern-based Design Model*, or *Design Model* for the sake of brevity, is a conceptual model defining the interplay of the main building blocks of a solution, wherein patterns are the modeling elements for these building blocks. They are connected with each other so that the design model represents the structure of the solution. ■

For instance, the cloud computing pattern language by Fehling et al. [FLR+14] introduces *User Interface Component* [FLR+14, p.175ff], *Processing Component* [FLR+14, p.180ff], and *Batch Processing Component* [FLR+14, p.185ff] – all patterns describing different kinds of application components. If the Designer wants to model an application receiving user input and processing it asynchronously in batch jobs to reduce the number of processing instances, they finally want to come up with a design model connecting the *User Interface Component* pattern and the *Batch Processing Component* pattern. However, the pattern language relates the *User Interface Component* pattern just to the more general *Processing Component* pattern by describing how they act together. The *Processing Component* pattern in turn relates further to *Batch Processing Component*, which provides the solution actually required. Thus, using this pattern language the Designer has to find *User Interface Component*, then follow the relation to *Processing Component* to finally navigate to *Batch Processing Component*. This shows, that a pattern language does not necessarily provide all possible connections between the contained patterns, so that the Designer rather has to resolve such transitive relations to pick the most suitable patterns and connect them in a design model. The formal introduction of design models is layed out in Section 4.3.

3.3.2 Step 2: Refine Conceptual Solution

Once a design Model balancing conceptual trade offs is modeled, the question arises how it can be implemented. This requires taking implementation-specific considerations into account. For instance, if the design model of a cloud application has to be implemented with specific technologies, such as services provided by a specific cloud vendor, then the question arises how to implement the patterns of the design model within the specific cloud environment. This requires to refine the solution principles captured by the patterns used in the design model towards constraints and technical specifications of the cloud services and technologies of the targeted cloud environment. To draw a non-IT example: if costumes in a film have to be arranged for specific setups like a historical point in time, all circumstances coming in from the era the film is playing in have to be considered. This may mean, for example, that a sheriff's costume needs to be refined into a Wild West sheriff's costume.

However, patterns used in a design model might not cover the translation of their provided conceptual solution towards such more constrained contexts. Although patterns refer to concrete implementations of solution principles via *known uses* [MD97] or *examples* [WF12], the constant evolution of new technologies can lead to situations where a pattern is *outdated* with respect to references to actual implementations with state of the art technology. This shows that the design model created in Step 1 of the EINSTEIN-Method has to be refined in some use cases by the Designer taking into account more implementation-specific constraints. In computer science in particular, there are several pattern languages by different authors, many of which have domain-specific overlaps in terms of the problems and solutions they capture. Thus, the Designer can make use of pattern languages capturing more details on how to implement a pattern in a specific technology to refine a design model. For example, the pattern language by [Ama13] provides implementation hints for the solution concepts of the cloud computing pattern language by [FLR+14]. Thereby, the Designer replaces patterns in the design model with patterns dealing with the same problem but refining the solution principles into implementation-specific solution building blocks. Such building blocks can be, e.g., specific cloud services and technologies along with configurations describing how to wire them. Or to recap the non-IT domain of costumes in films, a Designer can exchange the general sheriff pattern with the Wild West sheriff pattern [FBB+15], thereby enriching the design model with rules how to realize the intended costume with Wild West-specific characteristics.

Since this step relies on the availability of patterns containing technology-specific hints, it is performed optionally. Nevertheless, this step can also act as a hook enabling to enrich existing patterns with new examples an known uses. For instance, if the Designer makes use of patterns from a pattern repository, this can enable to add new examples or known uses to patterns [FBFL15; Rei13]. How pattern languages can be organized to support pattern refinement within design models is detailed in Section 4.2

3.3.3 Step 3: Select Concrete Solutions

Once the Designer prepared a design model and optionally refined it via patterns providing more specifics on their implementation they can overhand it to the Implementer who does the actual implementation work. The Implementer can make use of the design model and use it as the conceptual blueprint to understand the intended overall solution. The patterns and the relations among them captured in the design model are thereby the conceptual guideline on what has to be implemented, whereas the conceptual information found in the patterns also describes which tradeoffs have to be considered and which solution principles have to be realized. However, to avoid reimplementing patterns over and over again, the Implementer can hark back to implementations of the patterns and reuse them. This can be achieved by capturing implementations of patterns over time in a solution repository [FBFL15]. There, implementations of patterns are systematically stored and linked with the implemented patterns. This links conceptual solutions of the patterns with concrete solutions in the form of implementations. This is why implementations of patterns are called concrete solutions in this work based on the following definition.

Definition 3.3.2 (Concrete Solution – informal) A *Concrete Solution* is a reusable implementation of a pattern. It implements the solution principles detailed in a pattern with a specific technology and is stored along with relevant metadata in a solution repository. ■

Concrete solutions can be, for example, programs, programming code, configuration files, components of deployment models or further IT-artifacts, which can be combined with concrete solutions related to other patterns. In the case of the non-IT domain of costumes in films, concrete solutions stored in a solution repository are digital representations of the actual tangible costumes or even artifacts of a costume themselves [FBFL15]. When dealing with tangible concrete solutions, such as costumes, it can be necessary to capture many details of a concrete solution in order to create a digital representation. Concrete solutions, which are IT artifacts, typically do not require as much detail because they are themselves digital. Nevertheless to enable the reuse for concrete use cases at hand, metadata, such as the path to the IT-artifact, or *Selection Criteria* enabling to distinguish them and make them searchable are required. Especially, the selection criteria are essential means to find and reuse relevant concrete solutions because they capture domain-specific knowledge and terms determining when to use a certain concrete solution. Thus, they allow to filter all concrete solutions attached to a pattern according to the use case-specific needs at hand. For example, selection criteria can express the programming language or specific technology a concrete solution is realized with or the costs that had to be spent based on used cloud resources if the concrete solution is used. When searching concrete solutions in terms of costumes in films, selection criteria such as the historical era or the style of a location, region, or country a costume is designed for can be of interest when selecting proper concrete solutions.

Definition 3.3.3 (Selection Criteria – informal) A *Selection Criterion* is metadata stored along with a concrete solution in a solution repository. It expresses characteristics of a concrete solution required to distinguish it from other concrete solutions in a human- and machine-readable way. The characteristics are captured using domain- and technology-specific terms, which makes searching them applicable for Implementers with respective domain knowledge. ■

A solution repository provides a schema, API, and often also an user interface to store and retrieve all relevant metadata about concrete solutions along with the concrete solutions themselves in order to support and guide the Implementer to find and reuse them. Thus, an Implementer can search a solution repository to find proper implementations of the patterns defined in a design model. Thereby, conditions can be defined that must be met by concrete solutions of all patterns in the design model. For example, if an application has to be implemented using a specific framework, such as Spring Boot[3], the Implementer can filter for relevant concrete solutions via the selection criteria specifying that a concrete solution is implemented with this technology. Likewise, when designing costumes for a film playing in a specific geographic region, costumes can be selected, which reflect the cultural circumstances of the region. Using these mechanisms, the Implementer can refine the design model further by selecting and reusing concrete solutions of the patterns.

3.3.4 Step 4: Implement Missing Concrete Solutions

If the Implementer does not find suitable concrete solutions for all patterns contained in the design model, the concrete solutions have to be implemented manually. This step is often required for new pattern languages for which a solution repository is just set up. In rare cases (cf. [Bar18]),

[3]`https://spring.io/projects/spring-boot`

when for example a completely new field of research builds upon a solution repository by starting with gathering concrete solutions and then mining patterns or even a whole pattern languages from them, this step of the EINSTEIN-Method can be skipped. In other cases, crawling for implementations in open source software repositories [cf. Wet17] can also help to identify and gather candidates for concrete solutions, which then can be added to a solution repository and being linked to patterns and enriched by selection criteria after quality assurance by an Implementer.

Since the solution principles described by patterns are typically technology-agnostic, the solution knowledge of patterns is relevant for a long time. Some authors even speak of the *timelessness of patterns* [AIS77; Rei13]. In contrast, the body of concrete solutions in a solution repository is subject to constant renewal, as technologies and frameworks evolve over time or even completely new technological approaches such as new programming languages or even completely new computing models, such as quantum computing [LBF+20], emerge. This applies not only to solution repositories from IT, but also to other domains such as costumes in films. There new films are constantly being created whose vestimentary communication must be followed by new costumes in the solution repository in order to keep up with the times. Nevertheless, the value of older concrete solutions can potentially also increase over time. In IT, for example, it is often the case that technologies are no longer actively developed and are hardly mastered by experts, but are still in use in important and long running systems, such as the IT backends of insurances. In this case, solution repositories built up over time can preserve knowledge about implementations with specific technologies and keep it accessible.

3.3.5 Step 5: Generate Concrete Solution Aggregate

The core of the EINSTEIN-Method is the generation of a solution aggregate from the selected concrete solutions. This step is intended to relieve the Implementer of time-consuming manual work by aggregating concrete solutions automatically using aggregation operators. Aggregation operators encapsulate the logic to assemble compatible concrete solutions to overall concrete solution aggregates. Since aggregation of concrete solutions must take into account the technological specifics of compatible concrete solutions, aggregation operators have to be created and maintained for different technologies. For example, typically only concrete solutions created in the same programming language or technology are compatible with each other. If the formerly created design model represents a deployment model of an application, the selected concrete solutions typically have to be implementations of deployment descriptors of the same deployment technology and, therefore, an aggregation operator is required that is capable of aggregating them to solution aggregates. In IT, often APIs are used to abstract technical details, such as programming languages, to integrate components more easily. The EINSTEIN-Method is agnostic to such abstractions, thereby, grasping APIs as a specific type of concrete solutions. In addition, aggregation operators must ensure that the generated aggregate is valid for the corresponding technology so that it is a relevant aggregation result for the Implementer.

Definition 3.3.4 (Aggregation Operator – informal) An *Aggregation Operator* encapsulates the technology-specific logic required to aggregate concrete solutions. ■

The automation aspect of this step is represented by the gear symbol in Figure 3.1. Accordingly, the Implementer triggers this step and receives either a complete solution aggregate if the automated aggregation of all concrete solutions was successful, or partial aggregates that must then be integrated manually in the next step. The formal foundation of aggregation operators, their development, and how to structure and organize sets of concrete solutions in a solution repository is detailed in Chapter 6.

3.3.6 Step 6: Integrate and Refine Solution Aggregate manually

This step of the EINSTEIN-Method is conducted optionally by the Implementer if the automated aggregation of the selected concrete solution was not completely successful in the previous step. In this case, the Implementer receives the automatically generated aggregates and then has the option to manually integrate further concrete solutions with the solution aggregates. For this purpose, the Implementer can use a solution language that provides descriptions of steps required for manually aggregating concrete solutions (cf. Section 6.2). In addition, the Implementer can also make adjustments and refinements to the generated solution aggregates in this step in order to customize them in detail for productive use. This can involve, for example, adding business logic to components of an application or making configurations relevant for productive use.

3.3.7 Step 7: Use Concrete Solutions Aggregate

In the final step of the EINSTEIN-Method, the Implementer uses the final aggregated solution for productive use. This may mean that environment-specific configurations still need to be incorporated for execution in target systems, or the generated solution still needs to be compiled in a build pipeline for production use. This last step also provides a possibility for the identification of frequently occurring solution aggregates. Frequently occurring solution aggregates can provide an indication of so-called composite patterns, which correspond to recurring proven pattern combinations. Such indications of composite patterns can be fed into existing pattern authoring processes to complement an existing pattern language and also to associate entire solution aggregates with composite patterns.

3.4 Implementation of the EINSTEIN-Method

The introduced method abstracts from specific domains. Accordingly, it can and must be adapted for different domains. In particular, it is a matter of assigning the domain-specific artifacts but also roles accordingly to those presented in the method. Therefore, the described roles represent a domain-agnostic view. For the application of the EINSTEIN-Method, domain-specific roles accordingly coincide with those described here. In this way, the roles underlying the EINSTEIN-Method have to be refined in a domain-specific manner. Inherent in the application of the method in different domains, however, is that solutions are conceptually designed and then refined to the point of implementation. It is precisely this flow, from design to concrete implementation, that is reflected in the interaction between the roles of the Designer and the Implementer. Of course, these roles can also be performed by one and the same person. For example, in software development, it is often necessary that heterogeneous teams combining different roles are formed to elaborate complex software systems. Thereby, the roles of enterprise architects, solution architects, and software engineers are the relevant IT-roles that, depending on an actual IT project can fullfil the role of the Designer. Enterprise architects are typically responsible for the overall architecture of an IT landscape and, thereby, ensure that the overall architecture is in line with the IT strategy of an organization, which involves also the governance of allowed technologies, frameworks, or vendor-specific solutions. For instance, the governance of a specific cloud vendor can be in place, which requires to use specific cloud services and technologies. Thus, enterprise architects can take on the role of the Designer and formulate constraints for the Implementer, which have to be considered when implementing a concrete solution. Solution architects, in turn, are typically responsible for the overall architecture of systems and applications. Thereby, they often rely on patterns to design the intended architecture of an application, which, consequently, makes them most closely correspondent to the abstract role of the Designer. However, depending on the size and organizational structure of an organization also experienced software engineers can take on the role of the Designer very similar to a solution architect. The responsibility to select proper technologies and frameworks for implementations depends on multiple factors, such as the size of an organization, the organizational structure, and even the architecture style. For example, in a microservice architecture, the responsibility to select technologies and frameworks is often delegated to the software engineers implementing the microservices, while in a monolithic architecture the responsibility to select technologies and frameworks is often taken by solution architects. While solution architects typically focus on the architecture of an application and tend to do less implementation work, the opposite is true for software engineers. Thus, software engineers are typically responsible for the implementation of an application and, therefore, can take on the role of the Implementer. However, since in practice no definitive sharp line can be drawn between solution architects and software engineers, both IT roles can take over the role of an Designer and an Implementer.

Furthermore, domains can differ in terms of what the concrete solutions look like. In IT, we are usually dealing with digital concrete solutions, for example, program code, configuration files, deployment models, or similar. These are correspondingly easier to store as concrete solutions and thus also to reuse than, e.g., concrete solutions that exist as tangible artifacts such as costumes [BL15]. For these, a digital representation must be created accordingly for automated processing, as illustrated by the work of Barzen [Bar18], Barzen et al. [BBE+17], and Barzen and Leymann [BL15]. In the specific case, the applicability of the EINSTEIN-Method is linked to the cost/benefit ratio, especially for the creation of the digital representations of the concrete costumes because this can involve the development of sophisticated concrete solution meta-models and solution repositories, such as described by Fehling et al. [FBFL15], along with manual work to capture the costumes. In other digital cases, at least harvesting the concrete solutions, e.g., from code repositories, can be automated and thus does not contribute to the effort [WBFL17; Wet17].

Finally it has to be emphasized that the EINSTEIN-Method is put on orthogonally to already existing domain-specific procedure models and is not intended to replace these. Agile process models and frameworks from software development, such as Scrum [Sch97] or SAFe [Sca], can be mentioned here as examples. For example, Scrum does not distinguish between different roles of architects because the focus is on the organization of the agile development of software in iterations. However, also when developing software solutions in Scrum teams architecture is an important aspect that has to be considered. This is often achieved either by members of the Scrum team who take on the role of a solution architect or the Scrum team is supported by solution architects on demand. In any case, if the development is driven based on patterns and pattern languages, the EINSTEIN-Method can be used to support the development of software solutions in Scrum teams that bring together the heterogeneous roles of architects and software engineers. This shows, that since the EINSTEIN-Method supports the pattern-based design of solutions and the implementation based on concrete solutions, it can be used in combination with existing procedure models and frameworks. Thereby, the roles Designer and Implementer might not always be exactly mappable to distinguished roles in a domain or in procedure models, but are fulfill by a domain-specific interplay of roles that are responsible to design and implement solutions. Thus, the key driver to implement the EINSTEIN-Method in a specific domain is to identify the roles and processes in such process models that are intended for creating and transferring designs into implementations and, therefore, can be mapped to the introduced roles and methodical steps as introduced in Section 3.1 and Section 3.3.

3.5 Chapter Conclusion

In this chapter, the EINSTEIN-Method has been introduced as a methodical framework for the systematic reuse of concrete solutions. The EINSTEIN-Method is based on the idea of pattern-based design and the reuse of concrete solutions. Thereby, it is designed to ease and support the generation of whole solution aggregates from an interplay of individual concrete solutions. The EINSTEIN-Method is intended to be applicable in different domains and to support the reuse of concrete solutions independently of specific procedure models. This is achieved by binding the methodical steps to the roles of the Designer and the Implementer, who are responsible for the design and implementation of solutions, respectively. The EINSTEIN-Method is designed to be open to new technologies and to support the reuse of concrete solutions in different domains. In the following chapters, the main concepts required to implement the EINSTEIN-Method are introduced and formalized, such that they can be used as a conceptual blue print to establish the EINSTEIN-Method in new domains.

Chapter 4

Pattern Languages and Pattern-based Design

The first two steps of the EINSTEIN-Method enable to model conceptual solutions via patterns in design models and to refine abstract patterns in them with patterns providing implementation hints. In this chapter, a formalization of pattern languages based on graphs is given, which builds upon the *Pattern Language* by Alexander et al. [AIS77] and the informal definition given in Definition 2.1.2. The informal definition of pattern languages is generalized and formalized via the mathematical concept of graphs, which allows to transfer the concept of a pattern language to the domains and pattern languages investigated and applied in this work. Furthermore, it is shown how existing pattern languages can be connected to enable pattern refinement as described in Section 3.3.2. Building on this, the concept of pattern-based design models as informally introduced in Definition 3.3.1 is formalized. This chapter condenses the results from the work presented by Falkenthal et al. [FBB+15] and Falkenthal et al. [FBL18] and builds upon them.

4.1 Pattern Languages as Graphs

A pattern language interweaves multiple patterns by representing how they conceptually interact in design and implementation. Alexander introduced a pattern language as a directed acyclic graph with implicit larger and smaller semantics on the edges between patterns (cf. Definition 2.1.2). These specific semantics are sufficient to express the relation of patterns in the original domain of towns, buildings, and construction investigated by Alexander [AIS77]. However, when the concept of patterns and pattern languages got transferred to other domains, such as computer science, these specific semantics have not endured and is often replaced. Among others, Zimmer [Zim95] identifies categories of relations between the patterns for object-oriented design by Gamma et al. [GHJV94]. Zimmer [Zim95] describes relations between patterns with the following semantics: *"X uses Y in its solution"*, *"X is similar to Y"*, and *"X can be combined with Y"*. Reiners [Rei13] suggests *AND*, *OR*, and *XOR* semantics to link emergency response patterns in a pattern library. Fehling et al. [FLR+14] use semantics such as *see also* or *consider after* within their cloud computing pattern language. Zdun [Zdu07] and Zdun et al. [ZHD07] additionally introduce weights with different values on edges beyond primitive lexical semantics. Further examples of additional relation semantics are captured by Falkenthal et al. [FBB+15] and Falkenthal et al. [FBL18].

As a consequence, the strict limitation to the larger and smaller semantics as stated by Alexander has to be relaxed and generalized to transfer the concept of pattern languages in other domains. In general, the semantics of relations between patterns must be extended to allow arbitrary *weights*, including text documenting how two patterns are related to each other beyond the pure meaning of the relation. To reflect this, *Edge Types* are introduced in this work allowing to map arbitrary information onto relations between patterns, which is a generalization of the formulation of pattern relation descriptors via semantic web technologies introduced by Krieger [Kri18c]. They are organized in domains reflecting different types of structures being reused to specify arbitrary weights on edges. The basic definition of a pattern language as a directed pattern graph with types as weights on which this work is based follows in Definition 4.1.1.

A domain of edge types is a set of values that can be used to describe the relation between two patterns in a pattern language. Thus, a specific domain covers a set of symbols or words that represent termini required to express the relations between patterns of a pattern language. For example, in the case of the pattern language by Alexander et al. [AIS77], the relation between patterns is expressed by the termini *larger* and *smaller*. Since Alexander introduces directed edges between patterns that alway pointing from larger patterns to smaller patterns, the relevant termini can even be reduced to *smaller*. Thus, the domain of edge types for the pattern language by Alexander et al. [AIS77] is $D_{Alexander} = \{smaller\}$.

Definition 4.1.1 (Pattern Language)

A *Pattern Language* is a directed pattern graph $\mathcal{G}$ with typed weighted edges. Thereby, a typed edge has a reusable structure assigned to represent type-specific descriptions detailing the relationship of two patterns, s.t.,

$$\mathcal{G} = (\mathcal{P}, \mathcal{E}_{\mathcal{P}}, \mathcal{W}, \mathcal{D}, \alpha, \beta)$$

with

(i) $\mathcal{P}$ being a set of patterns where $card(\mathcal{P}) \in \mathbb{N}$

(ii) $\mathcal{E}_{\mathcal{P}} \subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{W}$

(iii) $\mathcal{W} \neq \emptyset$ being a set of edge weights

(iv) $e = (p_1, p_2, w) \in \mathcal{E}_{\mathcal{P}}$, then $\pi_1(e)$ is the starting point p_1 of edge e, $\pi_2(e)$ is the endpoint p_2 of edge e and $\pi_3(e)$ is the weight w assigned to e

(v) $\forall e \in \mathcal{E}_{\mathcal{P}} : \pi_1(e) \neq \pi_2(e)$

(vi) $p_1, p_2, \ldots, p_k \in \mathcal{P}$ is a path from pattern p_1 to pattern p_k $:\Leftrightarrow$ $(p_1, p_2, w_{1,2}), (p_2, p_3, w_{2,3}), \ldots, (p_{k-1}, p_k, w_{k-1,k}) \in \mathcal{E}_{\mathcal{P}}$

(vii) A path $p_1, p_2, \ldots, p_k \in \mathcal{P}$ is a simple path $:\Leftrightarrow$ $\forall 2 \leq i, j \leq k-1 : p_i \neq p_j$

(viii) $\forall p_i, p_{i+1}, \ldots, p_m \in \mathcal{P}$ that are simple paths holds $p_i \neq p_m$

(ix) $\forall e_i, e_k \in \mathcal{E}_{\mathcal{P}} : \pi_1(e_i) = \pi_1(e_k) \Rightarrow \pi_2(e_i) \neq \pi_2(e_k)$

(x) $\mathcal{D}$ being the set of all domains of types used to specify sets of values for type-specific descriptions of edges

(xi) $\alpha : \mathcal{W} \to \wp(\mathcal{D})$

(xii) $\beta : \mathcal{E}_{\mathcal{P}} \to \bigcup_{e \in \mathcal{E}_{\mathcal{P}}} \times_{D \in \mathcal{D}_{\alpha(\pi_3(e))}} D$

(xiii) $\forall e \in \mathcal{E}_{\mathcal{P}} : \beta(e) \in \times_{D \in \mathcal{D}_{\alpha(\pi_3(e))}} D$

where α is a map that types weights by assigning subsets of all domains to weights and β is a map that assigns type-specific descriptions to edges. ■

From Definition 4.1.1 follows that a pattern language is a directed acyclic and weighted graph with patterns as nodes and edges denoting the relationship between patterns in terms of semantics and descriptions structured by types. Although in many pattern languages references between patterns exist in both directions, in this work such references are understood as just one edge between the patterns. This can be attributed to the fact that such references in pattern languages are intended for easy navigation by the reader between all patterns, but do not capture the semantic interplay of two patterns differently. Rather, the references to other patterns carried out in pattern languages are materializations of the typed weighted edges described in Definition 4.1.1 in the patterns themselves, i.e., pattern languages can be understood as renderings of pattern graphs as defined above. For instance, two patterns p_1 and p_2 from a pattern language can reference each other denoting that if pattern p_1 is applied then one is typically faced with further design decisions captured by pattern p_2. In such cases, p_2

typically also references p_1 by describing the context in which p_2 can be applied. However, to express the semantics of the relation between p_1 and p_2 in a pattern language, just one edge is required that expresses the semantics of the relation between p_1 and p_2. Since pattern languages are mostly authored in text books or wiki pages with ordinary references or hyperlinks respectively, the above defined edges are typically materialized in the plain text of the patterns themselves. Thus, if a pattern language is, for example, presented as wiki pages, the edges between the patterns are materialized as hyperlinks between the patterns.

The above concept of a pattern language can be applied to the original *Pattern Language* by Alexander et al. [AIS77] as follows. Alexander defines that the semantics of relations between patterns strictly denote that one pattern is larger than the other pattern. He defines the structure of a pattern language by the following quote.

> *"And you see then what a beautiful structure a pattern language has. Each pattern is itself a part of some larger pattern [...]. And each pattern itself gives birth to smaller patterns [...]"* [Ale79, p.322]

Applied to Definition 4.1.1, the set of patterns is connected with edges of type *smaller* allowing to navigate from a pattern to all smaller patterns, which help *"to complete it"* [Ale79, p.313]. One example among many are the patterns *29 Density Rings* [AIS77, p.156ff] and *39 Housing Hill* [AIS77, p.209ff], where *29 Density Rings* references *39 Housing Hill*. Therefore, the semantics of the relation between these two patterns is clearly defined by the edge $e = (DensityRings, HousingHill, w)$ according to Definition 4.1.1. Since the structure of this pattern language is constructed via the *smaller* semantics, w is mapped via α in this example to $\{D_{Alexander}\}$, while β maps the edge e to $\{smaller\}$. Thus, also all other edges in the pattern language by Alexander et al. [AIS77] have the same weight w while β maps all edges to $\{smaller\}$.

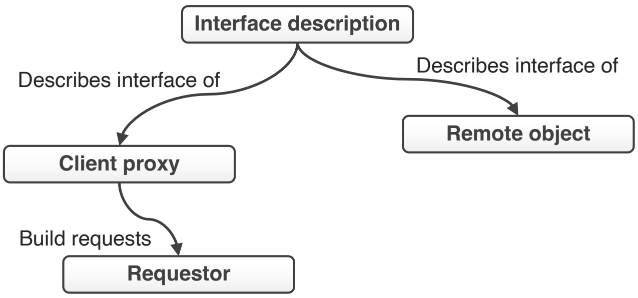


Figure 4.1: Excerpt of the Remoting Patterns by Zdun et al. [ZKV04]

The pattern language by Zdun et al. [ZKV04] can be grasped as a pattern language with more complex typed edges as introduced in this work. In their *Remoting Pattern Language*, they describe patterns for the design of distributed object middleware solutions. Thereby, among others, they introduce the so-called basic remoting patterns *Interface description*, *Client proxy*, *Marshaller*, *Remote object*, *Requestor*, *Remoting error*, *Invoker*, *Client-request handler*, and *Server-request handler*. Further, they use the termini to describe the semantics between the patterns given by the domain $D_{remoting} = \{$*Describes interface of*, *Builds requests*, *Marshalling requests*, *Demarshalling requests*, *Invocation*, *Raises*, *Send requests*, *Requests*, *Communicates with*$\}$. To show, how the above introduced formalism can be applied to the pattern language by Zdun et al. [ZKV04], the patterns *Interface description*, *Client proxy*, and *Remote object* are considered in more detail. An excerpt of the remoting patterns by Zdun et al. [ZKV04] is shown in Figure 4.1. The pattern *Interface description* is connected to the pattern *Client proxy* via the edge $e_1 = ($*Interface description*, *Client proxy*, $w_1)$ with $\alpha(w_1) = D_{remoting}$ and $\beta(e_1) = \{$*Describes Interface of*$\}$. Further, *Interface description* is connected to *Remote object* via the edge $e_2 = ($*Interface description*, *Remote object*, $w_1)$ with $\beta(e_2) = \{$*Describes Interface of*$\}$. This shows, that both, e_1 and e_2 carry the same semantics. Finally, *Client proxy* is connected to *Requestor* via the edge $e_3 = ($*Client proxy*, *Requestor*, $w_1)$ with $\beta(e_3) = \{$*Build requests*$\}$.

The remaining patterns can be connected in a similar manner utilizing the introduced formalism of Definition 4.1.1. Note, that for both of the above examples just one domain is used to describe the semantics of the relation between two patterns. However, the introduced formalism allows to use multiple domains to describe the semantics of the relation between two patterns, which can be useful, for example, if formerly isolated pattern languages are connected to an overall pattern language. Then, the semantics of the relations between patterns from the pattern languages can be expressed by different domains.

The definition of pattern languages according to Definition 4.1.1 also allows Alexander's ideas for extending pattern languages to be conceived very simply. Alexander describes that a pattern language is frequently subject to changes, adaptations, and extensions. Especially in situations when multiple pattern languages deal with aspects that belong together on a higher level, it can be beneficial to combine those pattern languages to provide an overall body of knowledge among all captured solution concepts. An example is the original pattern language by Alexander itself. It can be grasped as three individual pattern languages, one dealing with the design of towns, one with the design of buildings, and one with their construction. Nevertheless, Alexander et al. [AIS77] showed that bringing these three perspectives together in one pattern language creates one overall body of knowledge. Thereby, they formulate the process how to aggregate pattern languages by the following phrase.

> *"And, more subtly, we find also that different patterns in different languages, have underlying similarities, which suggest that they can be reformulated to make them more general, and usable in a greater variety of cases."* [Ale79, p.330]

The Definition 4.1.1 of a pattern language introduced above allows to understand their informal definition and introduction about how to aggregate pattern languages by means of set theory. Based on the concept of pattern languages as graphs containing patterns as a set of nodes and the relations among the patterns as sets of typed edges, the aggregation of pattern languages to overall ones can be fully understood by the following definition of a *Pattern Language Aggregator*.

Definition 4.1.2 (Pattern Language Aggregator) Let $\mathfrak{G}$ be the set of all pattern language graphs. Two pattern language graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are aggregated based on a set of new edges $\mathcal{E}$. Thereby,

$$\odot : \mathfrak{G} \times \mathfrak{G} \times \mathcal{E} \to \mathfrak{G}$$

is the *Pattern Language Aggregator* function that aggregates two pattern language graphs to a new single one containing new edges $\mathcal{E}$ with

(i) $\mathcal{E} \subseteq (\pi_1(\mathcal{G}_1) \cup \pi_1(\mathcal{G}_2)) \times (\pi_1(\mathcal{G}_1) \cup \pi_1(\mathcal{G}_2)) \times \mathscr{W}$

(ii) $\forall e \in \mathcal{E} : \pi_1(e) \in \pi_1(\mathcal{G}_1) \Rightarrow \pi_2(e) \in \pi_1(\mathcal{G}_2) \land \pi_2(e) \in \pi_1(\mathcal{G}_1) \Rightarrow \pi_1(e) \in \pi_1(\mathcal{G}_2)$.

$\odot$ aggregates two pattern graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ by using the fundamental union operation of set theory, s.t.

$$\mathcal{G}_3 = \odot(\mathcal{G}_1, \mathcal{G}_2, \mathcal{E}) = (\mathscr{P}_1 \cup \mathscr{P}_2, \mathscr{E}_1 \cup \mathscr{E}_2 \cup \mathcal{E})$$

with $\mathscr{P}$ being the set of patterns of $\mathcal{G}_1$ and $\mathscr{P}_2$ being the set of patterns of $\mathcal{G}_2$ as well as $\mathscr{E}_1$ and $\mathscr{E}_2$ being the edges of $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. ■

For the sake of brevity, the infix notation is $\mathcal{G}_3 = \mathcal{G}_1 \odot_{\mathcal{E}} \mathcal{G}_2$. $\odot$ is associative and commutative due to $\mathcal{G}_1 \odot_{\mathcal{E}} \mathcal{G}_2 = (\mathscr{P}_1 \cup \mathscr{P}_2, (\mathscr{E}_1 \cup \mathscr{E}_2) \cup \mathcal{E}) = (\mathscr{P}_2 \cup \mathscr{P}_1, \mathscr{E}_1 \cup \mathscr{E}_2 \cup \mathcal{E}) = (\mathscr{P}_2 \cup \mathscr{P}_1, \mathscr{E}_2 \cup \mathscr{E}_1 \cup \mathcal{E}) = (\mathscr{P}_2 \cup \mathscr{P}_1, (\mathscr{E}_2 \cup \mathscr{E}_1) \cup \mathcal{E}) = \mathcal{G}_2 \odot_{\mathcal{E}} \mathcal{G}_1$.

To solve use cases, it is often necessary to combine patterns from different pattern languages. In order to combine patterns from different pattern languages to an overall pattern language, another operator is required that allows to select a subgraph of a pattern language. For example, to implement an application that is based on asynchronous communication that has to be hosted and managed on a cloud platform, then just the subgraph of patterns from the pattern language by Hohpe and Woolf [HW04] dealing with asynchronous communication and the subgraph of patterns dealing with application hosting from the pattern language by Fehling et al. [FLR+14] are required. Thus, the operator $\trianglelefteq$ is introduced in the following to select a subgraph of a pattern language.

Definition 4.1.3 (Pattern Language Subsetting Operator) Let $\mathcal{G}$ be a pattern language graph. A subgraph of $\mathcal{G}$ is selected by the *Pattern Language Subsetting Operator*

$$\trianglelefteq : \mathfrak{G} \to \mathfrak{G}$$

with

(i) $\hat{N} \subseteq \pi_1(\mathcal{G})$

(ii) $\trianglelefteq (\hat{N})$ is the subgraph of $\mathcal{G}$ spanned by $\hat{N}$

■

To select patterns from different pattern languages $\mathcal{G}_1, \ldots, \mathcal{G}_n$ to combine them into an overall one the relevant pattern language subgraphs can be selected as $\hat{\mathcal{G}}_i = \overset{n}{\underset{i=1}{\trianglelefteq}} \mathcal{G}_i$. Then, the overall pattern language can be constructed as $\hat{\mathcal{G}} = \hat{\mathcal{G}}_1 \odot_{\mathcal{E}_1} \ldots \odot_{\mathcal{E}_{n-1}} \hat{\mathcal{G}}_n$.

The pattern language aggregator $\odot$ and the pattern language subsetting operator $\trianglelefteq$ show that the definition of pattern languages as presented in this work enables to understand and reason about the mechanisms of pattern languages informally introduced by Alexander et al. [AIS77] and Alexander [Ale79] with mathematical rigor. Further concepts described in Section 5.5 and also the prototypical implementation of the framework to support the EINSTEIN-Method described in Chapter 8 build upon the understanding and formulation of pattern languages as described in this section.

4.2 Pattern Refinement

Step 2 of the EINSTEIN-Method is about refining abstract patterns with patterns providing implementation guidance for certain technologies. To support this step, the concept of *Pattern Refinement* is introduced in this section. The formulation of pattern languages as graphs together with the concept of the pattern language aggregator (see Definition 4.1.2) show how pattern languages authored independently from each other can be connected. A special case of connecting formerly independent pattern languages is introduced in the following as pattern refinement.

Patterns are typically provided with a section about *known uses* or *examples* [MD97] illustrating the evidence that the introduced solution principles are proven in different real world use cases. Some even provide guidance via examples, e.g., in the form of programming code snippets [GHJV94]. However, broad guidance covering different technologies and implementation scenarios is typically not given. This is the outcome of pattern authoring processes (cf. [FBBL14; MD97]), where general solution principles are condensed into the *"gestalt"* of the pattern (cf. [Koh10; Koh11; Koh12]). The *gestalt* of a pattern abstracts away all implementation details and keeps the emergent qualities of a solution concept [Koh12]. According to Kohls [Koh12] this is because the process of abstraction decreases the number of constraints, which, at the same time, reduces also the guidance when applying a with specific technologies. This means that patterns following this principle are less instructive but open up more choices at realizing concrete implementations of a pattern, which corresponds to the definition by Alexander et al. [AIS77] that a pattern provides a proven solution which can be applied over and over again to many different use cases. At the same time, patterns focusing just on the *gestalt* and leaving out all technological hints lead to time-consuming efforts when it comes to applying them for use cases with specific technologies or technical constraints.

While being reusable for many use cases, such patterns open an *abstraction gap* between the represented solution principles and their actual implementations as depicted in Figure 4.2. The abstraction gap indicates that users of a pattern language have to refine the solution principles of patterns each time they want to apply them in implementations, which means they require deep knowledge about the targeted technologies and have to balance technological constraints being part of a use case at hand – all resulting in time-consuming efforts for implementing the patterns and, in case of unexperienced users, even in erroneous implementations [FBB+15]. In Figure 4.2, this relationship is indicated by the dashed arrows that connect the patterns to concrete use cases across the abstraction gap. Also if examples were kept at the point of authoring patterns, be it as links to external resources or by capturing example implementations, they get typically not updated and extended with examples from new emerging technologies. This is because most pattern languages are still published in text books or scientific papers. Those are immanently prone to abstraction gaps over time. Consequently, all this can be condensed into the following problems: (i) pattern languages often either do not provide guidance to implement the contained patterns or (ii) the actuality of examples and implementation guidance cannot be preserved because the availability of external sources cannot be ensured and patterns are not updated considering new emerging technologies.

These principles can be exemplarily revealed by investigating the pattern language by Fehling et al. [FLR+14]. In summary, the cloud computing patterns lift the captured solution principles to an architectural perspective for designing cloud-native applications leveraging capabilities and principles of cloud environments. Therefore, they are a means for modeling architectures and designs for such applications. However, they do not capture implementation examples in the pattern language itself but just provide references for further reading in external sources within the *known uses* section of the patterns. To take further examples, the implementation hints within the book on object-oriented design patterns by Gamma et al. [GHJV94] do not capture programming languages and

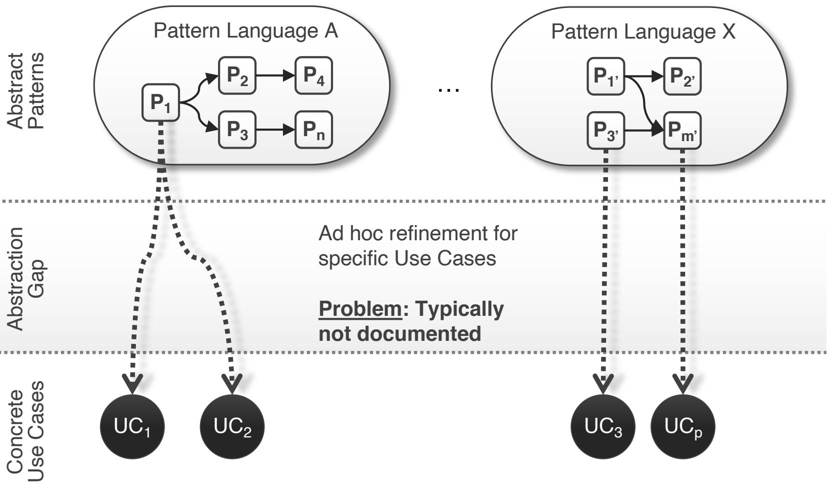


Figure 4.2: Abstracting details on implementations from patterns makes them applicable to many use cases at hand but creates an abstraction gap towards implementations

their specifics that have evolved after the time of writing the original book. The patterns on messaging-based integration of enterprise applications by Hohpe and Woolf [HW04] do not consider latest communication protocols and middlewares. The green IT patterns by Nowak et al. (cf. [NBL14; NL13; NLS+11; Now14]) do not consider new emerging technologies and paradigms, such as quantum computing, which promises energy savings in computation [JM22]. This list of examples is not exhaustive, but shows the immanent principle of arising abstraction gaps due to focussing on *gestalt* in patterns or abstraction gaps evolving over time.

A first attempt to bridge this gap is done in the EINSTEIN-Method in Step 2, in which pattern refinement takes place. Pattern refinement helps to overcome the lack of guidance towards pattern implementations arising from focussing on abstract solution principles by interweaving formerly isolated pattern languages. Thereby, abstract patterns are linked with patterns

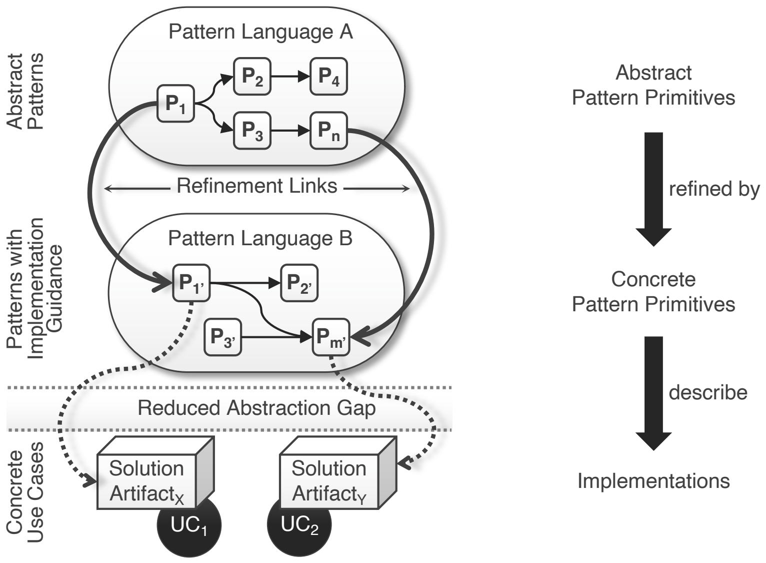


Figure 4.3: The abstraction gap can be closed via patterns providing implementation hints for specific technologies and constraints

which provide implementation guidance for specific technologies. Thus, pattern refinement relies on the presence of patterns either being authored on different levels of abstraction or providing additional implementation hints. This is conceptually depicted in Figure 4.3 where *Pattern Language B* contains patterns that help to close or at least reduce the abstraction gap as initially illustrated in Figure 4.2. Thereby, patterns just providing abstract solutions and few to no implementation hints from the depicted *Pattern Language A* are connected with those from *Pattern Language B* providing the missing implementation guidance. This principle can be applied to multiple levels of abstractions. Below, it is exemplified via two case studies in Section 4.2.1 and Section 4.2.2, respectively.

The links express the semantics that the targeted patterns *refine* the source patterns by providing implementation guidance capturing a more specific context, such as cloud services of a vendor or specific technology stacks. The targeted more concrete patterns refine the pattern primitives, which are domain-specific terminology and elements (cf. [ZAHD08]) used to describe the respective solution principles. Thus, they map abstract pattern primitives to more concrete ones, which ultimately allows users to translate the solution knowledge contained in the patterns more easily to concrete implementations. Applying the concept of the pattern language aggregator shows how an overall pattern graph can be formulated by specifying the two refinement links illustrated in Figure 4.3 as new edges connecting P_1 and $P_{1'}$ as well as P_n and $P_{m'}$ as $\mathcal{G} = \mathcal{G}_A \odot_{\{(P_1,P_{1'}),(P_n,P_{m'})\}} \mathcal{G}_B$.

4.2.1 Case Study: Pattern Refinement of Cloud Computing Patterns

The concept of pattern refinement can be applied in the domain of cloud computing. The above already mentioned pattern language by Fehling et al. [FLR+14] provides recurring problems and proven solutions in an abstract, cloud provider- and technology-independent manner for the purpose of designing cloud-native applications. However, design principles for cloud applications are also captured in pattern languages authored by the cloud vendors Amazon Web Services (AWS) [Ama13] and Microsoft [HSB+14; Mic14]. While the pattern language by Fehling et al. [FLR+14] provides patterns without any technology-specific implementation guidance this abstraction gap is filled by patterns from the other two pattern languages — each focussing on implementation guidance for the specific cloud vendor technology and service stack. Conceptually, the pattern language by Fehling et al. [FLR+14] can be seen as more abstract in comparison to the other two. Thereby, the patterns by AWS and Microsoft connect the presented solution concepts expressed by technology and vendor-agnostic primitives by Fehling et al. [FLR+14] with terminology, services, and technologies available within the respective cloud environment.

Overarching, 16 patterns can be found in the pattern languages by AWS and Microsoft, which take solution concepts by Fehling et al. [FLR+14] and refine them for the respective cloud environment [Fau16]. Exemplarily, this is shown and discussed by the patterns *Elastic Load Balancer* and *Stateless Component* by Fehling et al. [FLR+14] as well as *Scale out Pattern* and *State Sharing Pattern* by AWS [Ama13] in the following.

When designing cloud applications, it is often intended that the application is capable of processing varying amounts of workloads. Thereby, it is necessary that the number of different application components can be scaled independently from each other to match different workloads. The patterns *Elastic Load Balancer* and *Scale Out Pattern* show how this can be achieved. More precisely, the patterns describe how application components can be provisioned and no longer needed ones can be decommissioned based on monitored workload, i.e., how they can be elastically scaled. As mentioned above, because the abstract patterns by Fehling et al. [FLR+14] do not provide implementation guidance, the *Elastic Load Balancer* pattern does not give hints on how the presented solution concepts can be implemented e.g., with the cloud offerings of AWS or Microsoft. However, this guidance can be found in the *Scale out Pattern* by AWS [Ama13].

Figure 4.4 shows how the *Scale out Pattern* refines the primitives used in *Elastic Load Balancer* to AWS-specific components and services. While *Elastic Load Balancer* introduces the component capable of spreading workload among different instances of an application component being a *Load Balancer*, the *Scale out Pattern* helps to map this concept to a actual service in the AWS ecosystem, which is called *Elastic Load Balancer Service*. This service can multiplex workload among application instances hosted and managed in the AWS cloud. Thus, the *Scale out Pattern* provides concrete guidance how the concept of a *Load Balancer* can be realized within the AWS cloud. Another concept required for scaling application components due to encountered workload is *Monitoring* as depicted on the left in Figure 4.4. *Scale out Pattern* helps to translate the concept of monitoring to the *Cloud Watch* service of the AWS cloud offering. Of

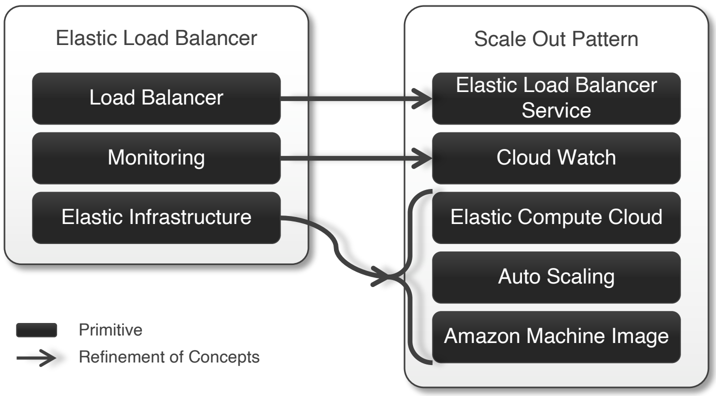


Figure 4.4: Pattern refinement allows to map abstract primitives of the *Elastic Load Balancer* pattern by Fehling et al. [FLR+14] to vendor-specific primitives of the *Scale out Pattern* of the AWS Cloud Design Patterns [Ama13]

course, also automated provisioning and decommissioning of application component instances is required to enable elastic scaling. The *Elastic Load Balancer* describes the concept of an *Elastic Infrastructure* allowing for these capabilities. In the AWS cloud, the pendant to the concept is a combination of *Elastic Compute Cloud*, which is a service to provisioning compute resources, *Auto Scaling*, which is a service configuring the elastic scaling of compute instances, and *Amazon Machine Images*, which are templates for launching virtual machines. Those are the primitives used in the *Scale out Pattern* to describe how the more abstract solution concept represented by the primitive *Elastic Infrastructure* can be realized with the AWS cloud stack. Thereby, the *Scale out Pattern* provides concrete guidance how those refined primitives have to be combined and configured to allow elastic scaling within AWS.

Further, to efficiently scale application components elastically, they ideally have to follow design principles introduced in the pattern *Stateless Component* by Fehling et al. [FLR+14]. Specifically, this pattern insists

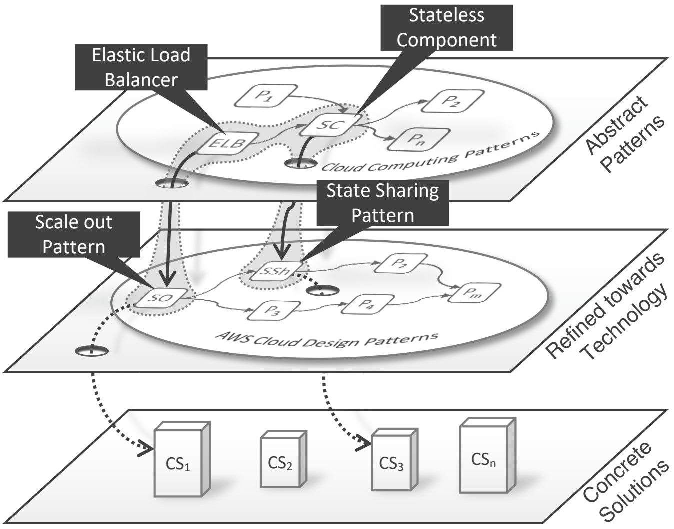


Figure 4.5: The Cloud Computing Patterns by Fehling et al. [FLR+14] can be linked with AWS Cloud Design Patterns [Ama13] to allow the refinement of solution concepts towards implementations within the AWS cloud offering. [FBB+15]

that application components should not maintain session state internally, but rather retrieve it on every execution call. This eases elastic scaling because no further management of session state is required, which finally reduces complexity and also leads to failure resiliency. Of course, when applying this pattern in real use cases this principle has to be realized within a respective cloud environment, thus, how state can be held external from an application component has to be mapped to technical solutions and services provided by the cloud. The *State Sharing Pattern* by Amazon Webservice [Ama13] helps to map this solution principles to virtual servers in combination with key-value stores, which allow to externalize the state from application components.

Figure 4.5 shows the described scenario schematically and illustrates how refinement links enable users to work with Fehling's technology-agnostic pattern language on the one hand and to dive into AWS's pattern language on the other hand to understand corresponding relevant solution concepts from the AWS cloud. The illustrated overall pattern language can be expressed as $\mathcal{G} = \mathcal{G}_{CCP} \odot_{\{(ELB,SoP),(SC,SSP)\}} \mathcal{G}_{CDP}$ with ELB, SoP, SC, and SSP being abbreviations for the respective patterns and CCP as well as CDP representing the Cloud Computing Patterns and the AWS Cloud Design Patterns.

4.2.2 Case Study: Pattern Languages for Costumes in Films

To show the applicability of pattern refinement beyond the domain of IT, in the following, patterns from the domain of costumes in films are investigated. Patterns have proven to be a valuable means to capture design elements, which are essential to communicate specific stereotypical characters via costumes in films [Bar18]. Patterns thereby capture relevant parameters of a costume, such as base elements and primitives a costume is made of, colors of the base elements and primitives, modifications of the base elements and primitives, etc., which are essential for the vestimentary communication of clothes [Bar18; FBB+15; FBB+17].

The initial patterns of the pattern language on costumes in films by Schumm et al. [SBLE12] show how design knowledge for costumes in films can be captured and normalized into a canonical pattern format. Based on this pattern format, Barzen and Leymann [BL15] suggest that pattern languages for costumes in films can be organized according to different film genres because each genre requires specific design elements and conventions, which are expressed by corresponding genre-specific patterns. For instance, costumes in a western film have to follow the clothing characteristics of the wild west era while costumes for science-fiction films typically have to show some technologically advanced parts in order to work in

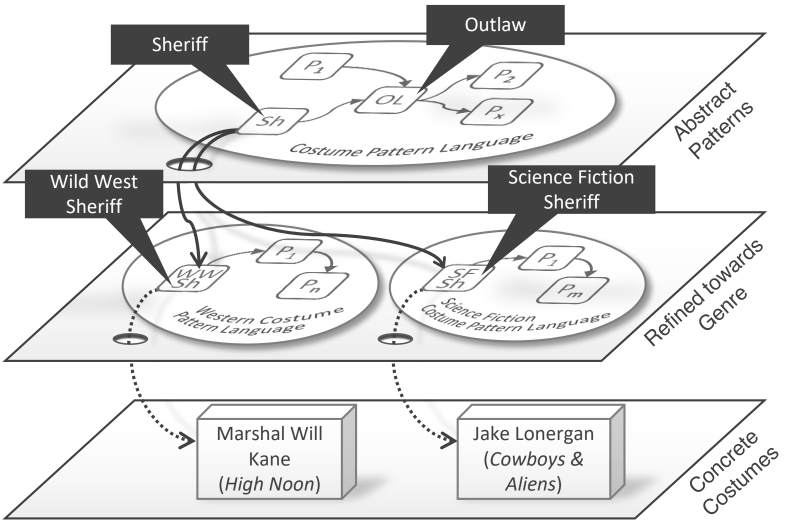


Figure 4.6: Genre-specific Costume Pattern Languages [BL15] can be connected via refinement links. [FBB+15]

this genre. Figure 4.6 shows this genre-specific organization of costume pattern languages conceptually by a western costume pattern language and a science fiction costume pattern language.

The *Wild West Sheriff* and *Science Fiction Sheriff* patterns are shown there as examples. Moreover, a closer look at the two concrete manifestations of these patterns presented – once with the role of Marshal Will Kane (Gary Cooper) in the Film High Noon (1952, Director: Fred Zinnemann) and once with the role of Jake Lonergan (Daniel Craig) in the film Cowboys & Aliens (2011, Director: Jon Favreau) – reveals that genre-specific costume elements are present, such as spurs or high-tech gear but also generic ones, such as the sheriff's star. Thus, the generic aspects of such costumes can be captured into a genre-agnostic costume pattern language, which is depicted above the genre-specific languages in Figure 4.6. This generic pattern language not only allows to capture essential design elements of a costume, such as the sheriff's star that are used in different genres, but also interactions between different costumes, which typically occur among all genres, such as the constant conflict between good and evil expressed through the contrast between a sheriff and an outlaw. Thus, this layers of abstraction allow for a clear separation of concerns among the genre-specific pattern languages on the one hand. On the other hand, in order to develop costumes starting from basic genre-agnostic essential design elements used in different film genres, such as the sheriff's star, the pattern languages can be connected using pattern refinement. The abstract pattern language on the top of Figure 4.6 thereby allows for a similar technology-agnostic perspective as the pattern language by Fehling et al. [FLR+14] as discussed above does. The refinement links connecting the more abstract patterns with the genre-specific ones then also allows to get more guidance on implementing the respective costumes for the setting of a specific genre, which corresponds to specific cloud environments from the above example of the cloud patterns and vendors. The illustrated scenario in Figure 4.6 can be expressed using the pattern language aggregator as $\mathcal{G} = (\mathcal{G}_{CP} \odot_{\{(Sh,WWSh)\}} \mathcal{G}_{WCP}) \odot_{\{(Sh,SFSh)\}} \mathcal{G}_{SFCP}$ with *Sh*, *WWSh*, and *SFSh* being abbreviations for the respective patterns and *CP*, *WCP*, as well as *SFCP* representing the Costume Pattern Language, the Western Costume Pattern Language and the Science Fiction Costume Pattern Language.

4.3 Pattern-based Designs

Alexander describes patterns being a dualism of "*things and the rules to create that things*" [Ale79, p.185, p.247]. Consequently, patterns represent entities and the knowledge about how to place those entities within the interplay of other entities in a specific domain. Thus, when designing conceptual solutions, Designers can use patterns in design models to reflect an overall solution, as intended by the EINSTEIN-Method. Many pattern languages support this by providing meaningful icons for the patterns, which can then easily be used even on whiteboards to design with the patterns of the pattern languages. Among many others, this is realized in the patterns and pattern languages languages by [EBF+17; FLR+14; HW04; NLS+11; RBF+16; Rei13; Ric18; SBLE12].

In computer science and especially in application architecture designing with patterns is a widely applied approach (cf. among others [AEK+07; AFL12; AZ05; BBKL13; BDH05; Bec07; Bec96; BMR+96; Bre16; FLR+14; HW04; RBF+16; Ric20; WBLV21]). For instance, guided by the structure of the pattern language by Fehling et al. [FLR+14], Designers can incrementally create the design of cloud-native applications. However, the architecture of more than just oversimplified example applications requires typically a sophisticated interplay of multiple patterns, which is often not already represented in the used pattern languages. For example, if a cloud application consists of multiple components all hosted on different cloud platforms, the architecture requires to express this by multiple instances of the *Processing Component* pattern connected to multiple instances of the *Elastic Platform* pattern. The same applies, for instance, to applications designed with the messaging patterns by Hohpe and Woolf [HW04] if multiple *Message Endpoints* are integrated. Or if a microservice architecture makes use of multiple *Circuit Breakers* and *API gateways* (cf.[Ric18]). Or, finally, if multiple *Mains-Powered Devices* are connected to multiple *Device Gateways* modelled using the patterns by Reinfurt et al. [RBF+16; RBF+17c]. This is not an exhaustive list but only a selection from the domain of application architecture to illustrate that for real world use cases it is often inevitable to model multiple instances of patterns and their interplay to realize the intended architecture of a complex system.

This phenomenon is not limited to computer science. Also in non-IT domains, such as designing whole towns or costumes for films, this principle can be recognized. For instance, when designing districts of a city or buildings within there, typically multiple instances of the patterns by Alexander et al. [AIS77] are required to construct sophisticated conceptual solutions. Or when arranging a whole movie scene with multiple characters it is often necessary to combine multiple instances of the costume patterns by Schumm et al. [SBLE12].

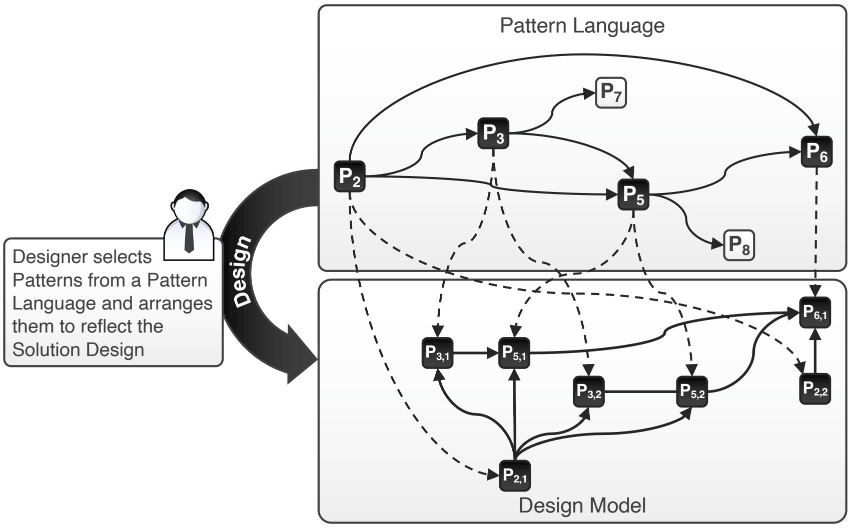

Pattern Language
P7
P3
P6
P2
P5
P8
Designer selects Patterns from a Pattern Language and arranges them to reflect the Solution Design
Design
P6,1
P3,1
P5,1
P3,2
P5,2
P2,2
P2,1
Design Model

Figure 4.7: Designing with patterns means projecting pattern languages to a design model

In conclusion, this means that designing solutions with patterns results in pattern graphs, which are not just simple subgraphs of the used pattern language. According to the EINSTEIN-Method, pattern-based design rather leads to design models that go beyond the mere structure of the pattern language used.

Figure 4.7 illustrates this principle of pattern-based design. Starting from a pattern language, patterns are selected when designing a concrete solution. However, since individual patterns are required multiple times in the intended solution design, the patterns are indexed according to their number of occurrence in the design model. The respective indices of the patterns indicate multiple instances of a pattern in the design model. This allows the Designer to express the interplay of different usages of a pattern in a design model properly. For instance, pattern P_2 is contained twice in the design model in Figure 4.7 reflected by $P_{2,1}$ and $P_{2,2}$. By connecting the instances of P_3 and P_5 with $P_{2,1}$ the Designer can express that those form an interplay in the design model, while $P_{2,2}$ and $P_{6,1}$ form another interplay. Thus, the design model unambiguously expresses the occurrence of multiple pattern usages and their respective interplay required in the final implementation of the solution design.

4.4 Pattern-based Design Models

A pattern-based design model is according to Definition 4.4.1 a pattern graph that defines the structure of an intended overall solution by specifying which patterns have to be aggregated.

Definition 4.4.1 (Pattern-based Design Model) A *Pattern-based Design Model*, in short *Design Model*, specifies an intended solution design by the interplay of patterns, which solves a concrete problem at hand.

Given a pattern language $\mathcal{G}$ and the index set I, then $(p_i)_{i \in I}$ with $p_i \in \pi_1(\mathcal{G})$ is the family of patterns from a pattern language contained in a Design Model.

A Design Model is a graph $\mathcal{G}_{dm} = (\mathcal{P}_{dm}, \mathcal{E}_{dm}, \mathcal{W}_{dm}, \mathcal{D}, \alpha_{dm}, \beta_{dm})$, s.t.,

(i) $\mathcal{P}_{dm}$ is the multiset induced by the family $(p_i)_{i \in I}$ with $p_i \in \pi_1(\mathcal{G})$ and $card(\mathcal{P}_{dm}) \in \mathbb{N}$

(ii) $\mathcal{E}_{dm} = \mathcal{P}_{dm} \times \mathcal{P}_{dm} \times \mathcal{W}_{dm}$

(iii) $\mathcal{W}_{dm}$ is a set of weights representing domain-specific semantics of the interplay of patterns

(iv) $\alpha_{dm} : \mathcal{W}_{dm} \rightarrow \wp(\mathcal{D})$

(v) $\beta_{dm} : \mathcal{E}_{dm} \rightarrow \bigcup_{e \in \mathcal{E}_{dm}} \times_{D \in \mathcal{D}_{\alpha_{dm}(\pi_3(e))}} D$

(vi) $\forall e \in \mathcal{E}_{dm} : \beta_{dm}(e) \in \times_{D \in \mathcal{D}_{\alpha_{dm}(\pi_3(e))}} D$

■

The edges within a design model are directed weighted to express specific semantics of the targeted domain of the design model along with the semantics that connected patterns have to be aggregated to form an overall solution. This work limits design models to express the structural interplay of patterns, although other work has already shown that further semantics can be introduced to express also an adaptive influence of patterns on each other [HBF+20].

4.5 Implementation of the Design Model Concept

A design model is conceptually similar to domain-specific languages, such as different modeling or design languages[1], which rely on modeling graphs. However, rather being a specific modeling language focusing and covering a domain-specific purpose, it is a modeling language independent concept that allows to model solutions with patterns. Thus, in some cases it can be beneficial to translate the concept of design models into modeling concepts within specific modeling languages. This is inevitable if a specific language is the de facto Lingua Franca in a domain. The concept of design models can be transferred to a domain-specific language under the following conditions:

- The modeling paradigm is graph-based
- Patterns can be introduced as new modeling elements
- The relations between patterns can either be specified by existing relation types, or the relation types can be extended to express that two modeling elements of type *pattern* have to be aggregated

[1] see among others ArchiMate [Ope22], UML [Ope22], or TOSCA [OAS13]

Some modeling languages are designed to be extensible. For instance, the *Unified Modeling Language (UML)* [OMG07] can be extended via UML-profiles and the *Object Constraint Language (OCL)* [Obj14] enabling to model pattern languages [Gri11]. Other approaches extend UML to map domain-specific primitives with pattern concepts, which enables to model variants of patterns in terms of elements of the patterns [DY03; MDM+18; RS09]. Harzenetter et al. [HBF+18b; HBF+20] show how patterns can get modeling elements within deployment models. All these examples show how the concept of design models can be translated into specific modeling languages. Nevertheless, this work sticks to the pure concept of design models to express the interplay of structural patterns in order to design larger solutions consisting of the interplay of multiple patterns. This is to help ensure that the concepts presented in this work remain agnostic of a domain-specific language and, thus, can be applied to domains beyond those mentioned in this work.

4.6 Chapter Conclusion

This chapter layed the foundation for the concept of design models, which are used in the EINSTEIN-Method to conceptually design solutions with patterns. The core understanding of pattern languages being graphs was introduced and the concept of the pattern language aggregator was presented. The formalization of pattern languages as graphs is the basis for further concepts introduced in the following chapters. The formalization will be refined stepwise to incorporate further concepts, such as concrete solutions, which are introduced in the next chapter. The concept of design models is kept domain-independent to allow for a broad applicability of the EINSTEIN-Method.

Chapter 5

Reuse of Pattern Implementations as Concrete Solutions

One of the main parts of this work is the reuse of pattern implementations, which are referred to as *Concrete Solutions*. This chapter builds upon the introduced concepts and formalizations of pattern languages and pattern refinement in the previous chapters. Firstly, the concept of concrete solutions is introduced and it is shown how concrete solutions can be conceptually understood and how they can be represented digitally. Secondly, the formalizations of pattern languages as graphs of patterns as well as pattern refinement as presented in Chapter 4 are extended to the conceptual model laying the basis for the reuse of concrete solutions. This chapter builds upon the work on concrete solutions and patterns by Falkenthal et al. [FBB+14a; FBB+14b] and the work on representing concrete solutions based on semantic web technologies by Krieger [Kri18c].

5.1 Concrete Solutions

Today, pattern languages are usually *harvested* by experts [Rei13]. By abstracting the problems and solutions from their expertise and experience, these experts determine the content of the patterns. This approach is referred to as the *pattern guru approach* by Reiners [Rei12; Rei13] because the captured patterns rely on the consensus of usually a small group of experts. This results in two caveats. Firstly, patterns created in this way are difficult to verify whether they actually represent proven solution knowledge for recurring problems. Even though the section on *known uses* of a pattern shows that it is used in practice, it is difficult to trace the concrete characteristics of the pattern in detail, as one often does not have access to the actual implementation. This leads to the fact that *pattern provenance* can only be specified to a limited extent at the time of formulating a pattern. Moreover, once a pattern is applied to further new use cases, the pattern itself can just hardly be extended by adding new applications as evidence for the pattern. Secondly, the abstraction of solution principles into patterns also detaches them from their underlying concrete implementations. Meaning that both, for known uses of patterns and also new applications, the pattern implementations get lost for later reuse or advice. Thus, manual effort and knowledge is required to apply the patterns to concrete use cases every time a pattern has to be applied.

Recent research approaches by Barzen [Bar18], Barzen et al. [BBE+17], Falkenthal et al. [FBB+17; FBD+15], and Strehl [Str15] have shown that pattern authoring processes can be supported by data analytics techniques and approaches such that harvesting proven solution principles can be systemized, partially automated, and most importantly gets repeatable and plausible based on given data. Thereby, the process of revealing costume patterns from a corpus of concrete costumes captured from movies shows that systematically storing concrete solutions enables to establish provenance and evidence for known uses of patterns and the rational why a pattern is proven in practice. In addition, it is possible to retrieve concrete solutions quickly if the derived patterns are linked to the concrete solutions.

Fehling et al. [FBFL15] demonstrate this for costume patterns and concrete costumes on the basis of a repository for costume patterns and a solution repository for concrete costumes.

Even though also other approaches exist to derive and underpin patterns by concrete evidence and concrete solution knowledge iteratively [FJZ+12; Rei13], the actual concrete solutions are typically not stored for systematic reuse. They are rather abstracted into informal evidence in the form of textual descriptions formulated within the canonical structure of the pattern documents similarly to *known uses*, or are not captured at all. In this case, the concrete solutions get also lost during the pattern authoring process, although mechanisms are foreseen in the managing pattern repository to keep evidence and known uses of patterns up-to-date [Rei13]. Nevertheless, also such approaches hinder to directly reuse concrete solutions connected to patterns and, thus, lead to manual efforts over and over again when it comes to applying a pattern.

Therefore, concrete solutions are introduced in this work as core concepts being not just informal parts of pattern descriptions but full-fledged entities of a theory connecting pattern languages and concrete implementations of the patterns. Thus, concrete solutions are reusable building blocks representing implementations of the solution principles documented in a pattern. This means, that concrete solutions are realizations of a pattern in specific contexts, such as specific technologies, cloud environments, programming languages, film genres, urban environments and construction sites, among others. The conceptual shift, from textual evidence, examples, and known uses in pattern documents, to concrete solutions being individually managed entities linked to patterns is depicted in Figure 5.1.

The approach on connecting patterns and implementations of them as introduced in this work specifically considers the aspect of reusing proven implementations. In this terms, this work extends the fundamental pattern theory by Christopher Alexander. Alexander specifically studied the process of design analyzing and identifying invariants of design concepts for buildings and construction. He layed the focus on reusing proven con-

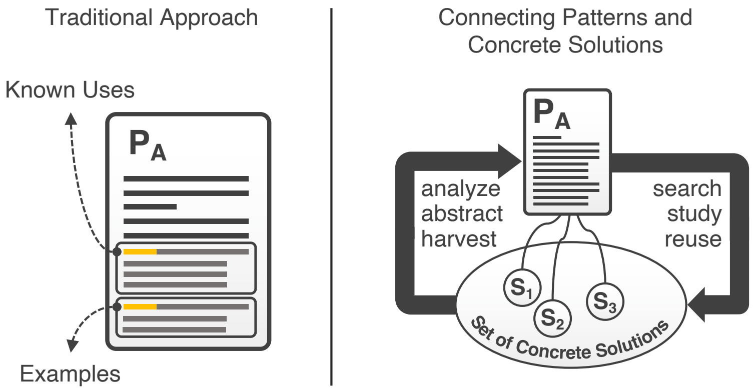


Figure 5.1: Keeping concrete solutions connected to patterns enables their reuse while establishing evidence and provenance for patterns

ceptual design principles. This work extends the core idea to also make implementations, which have proven to work in actual real life scenarios, to be reusable. The proposed inherent connection between patterns and concrete solutions enables the latter to be reused, and conversely, patterns to be derived and underpinned. It is important to point out that in general no pattern can exist without concrete solutions of the pattern being part of some implementations. This is because the set of concrete solutions form the evidence that a particular design principle manifested in the concrete solutions is actually a pattern due to the definition of a pattern being a proven solution for often recurring problems in a specific context (cf. Definition 2.1.1) – which requires at least three concrete solutions to meet the *rule of three* [Cop96; KU09].

Accordingly, a variety of concrete solutions exist for each pattern in the form of context-specific implementations. Thereby, the concrete solutions range from being helpful to understand how a pattern can be implemented in a specific context to directly reusable concrete solutions, which can be applied in a specific context without any further adaption.

Consider, for example, the Model-View-Controller pattern (MVC) [Ree03], which describes how user interactions can be implemented via a graphical user interface loosely coupled with the system logic. In practice, realizations of this pattern exist in applications implemented in many different programming languages. Once systematically captured, stored, and connected with the MVC pattern, concrete solutions in the form of programming code in different languages can be reused when implementing new applications. Since the MVC implementations typically have to be adapted to the specific context to realize the look and feel of the application at hand, the concrete solutions can be used as implementation hints providing skeletons to learn directly from. Thus, they can be used as a starting point for implementing the MVC pattern in a new context. In contrast, for example, the *Wire Tap* pattern [HW04] is a pattern that does not require many adaptations to realize it in a new context. It describes how to intercept point-to-point channels in a message-based system to inspect the transmitted messages for monitoring, testing, and troubleshooting purposes. The pattern describes that this can be achieved by inserting a generic *Wire Tap* that forwards a copy of the messages to a monitoring system. Thus, a concrete solution of the *Wire Tap* pattern can be realized as a generic implementation for a message-based system, which can be configured with the specific channel to intercept and an endpoint to forward the intercepted messages to. This concrete solution can be reused in different contexts without any further adaptation of the interception and forwarding logic.

The systematic capturing and organization of concrete solutions is a vital aspect of the proposed approach in this work. In order to capture concrete solutions along with patterns there are in general three possibilities: (i) the concrete solutions can be captured and stored systematically as part of the pattern authoring process (cf. [Bar18]), (ii) they are captured and stored when applying a pattern in a new context, such as an implementation with a new programming language, or (iii) they are explicitly developed for easing the reuse of a pattern because they are identified as recurring implementation parts where direct reuse promises time savings or assures proper functionality of critical system parts. Especially, in security critical systems the last aspect plays an important part. In IT, it is generally seen as good practice to reuse proven open-source libraries, such as creating pseudo random numbers or encrypting or decrypting messages instead of implementing such functionality themselves.

Since concrete solutions are independent entities, they are stored and organized in solution repositories (cf. [FBFL15]). In the following, it is introduced and described how concrete solutions can be grasped formally to design solution repositories capturing the domain-specific details and aspects of concrete solutions.

5.2 Formalization of Concrete Solutions

The concept of concrete solutions can be implemented differently in different domains concrete solutions. Even in a specific domain, concrete solutions typically vary with respect to the specific context they are implemented for. For example, the design patterns by Gamma et al. [GHJV94] can be implemented in different object-oriented programming languages, thereby, resulting in different concrete solutions. However, although implemented in different programming languages the resulting artifacts have some characteristics in common. For example, assuming they are programming code, they have in common that they can be versioned in a version control system such as git[1] or subversion [2]. They are software and, therefore, underlay copyrights, which are typically specified by a software license. Further, they are compatible with specific versions of a programming language compiler or runtime. The list of characteristics can be continued, i.e., those mentioned are examples of many others.

[1] https://git-scm.com
[2] https://subversion.apache.org

In contrast, concrete solutions from other domains have completely different specifics. For example, looking at the domain of costumes in films, it is notable that concrete solutions in the form of costumes must be stored completely differently for systematic reuse than the previously described programming code. Here the first essential difference to artifacts from IT is already the fact that costumes must first be represented digitally in a solution repository. The research on vestimentary communication of costumes in films by Barzen [Bar18] shows that characterizing properties with underlying taxonomically organized domains containing more than 3000 concepts are necessary to describe a costume (cf. [BBL22]).

And so it is for concrete solutions from further domains and contexts. Thereby, the structure and information content of a stored concrete solution are essential for its systematic reuse, because this is what a user enables to systematically search, understand, and reapply a concrete solution in a specific use case at hand. Thus, in order to systematically reuse concrete solutions they have to be stored along with structured metadata and domain-specific semantics describing them. Most importantly, concrete solutions require *Identity*. This means, that concrete solutions are raised from arbitrary solution snippets to entities connected to a pattern language by assigning a specific unique identifier. This identifier enables to distinguish them and making them specific entities, which can be connected to patterns the same time. In contrast to known uses or exemplary text passages in pattern documents, this mechanism enables to manage concrete solutions independently from patterns. The second invariant, which all concrete solutions share, is that they have a specific *Type*. Thereby, different types reflect that the concrete solutions are non-compatible, e.g., being implemented in different technologies such as different programming languages, or even being artifacts from completely different domains as shown above. The third invariant, which all concrete solutions share, is that a domain-specific *Structure* is required to describe them and make them reusable. This structure covers the semantics to locate the concrete solution and all domain-specific metadata required by experts to systematically reuse the concrete solution. In case of programming code, e.g., the location can be a

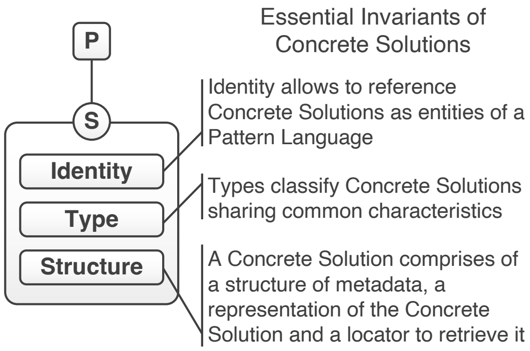

P
S
Identity
Type
Structure
Essential Invariants of Concrete Solutions
Identity allows to reference Concrete Solutions as entities of a Pattern Language
Types classify Concrete Solutions sharing common characteristics
A Concrete Solution comprises of a structure of metadata, a representation of the Concrete Solution and a locator to retrieve it

Figure 5.2: A concrete solution has an identity, a type, and captures all relevant metadata required for its systematic reuse

code repository, thus, the structure describing the concrete solution has to support to store a link to code repositories. In case of tangible costumes in films, the location can be expressed by a descriptor pointing to a specific wardrobe or compartment within it, depending on how and where the concrete costumes are kept. However, since the actual physical costumes might sometimes not be kept for direct reuse, costumes can be expressed and represented in a solution repository digitally. Representing tangible concrete solutions digitally enables to preserve all aspects about it omitting the necessity to handle the actual physical objects such as the costumes but, at the same time, keeping the essence for reproducing them when needed. The same applies for scores in the domain of music [BBE+17] and, considering the same principles, can be extended to construction drawings addressed by the pattern language by Alexander et al. [AIS77].

These structural invariants among concrete solutions are captured as depicted in Figure 5.2. Domain-specific concrete solutions refine this concept by structures relevant for the domain under consideration, as exemplarily described above. The metadata can be developed according to the needs of a certain domain, which assures that the concept of a concrete solution itself is domain-independent. As a consequence, this structure enables to describe concrete solutions digitally. Thereby, concrete solutions can be operationalized in solution repositories, which are IT-systems introduced later in this work. Solution repositories can be implemented with different technologies. A first example, that can be mentioned is the repository for storing and analyzing costumes in films by Fehling et al. [FBFL15]. It is based on a relational database and a single page application implementing the specifics to enter detailed descriptions of costumes. In this case, the structure of the concrete solutions of costumes is translated to a relational database schema. To analyze the corpus of costumes, data mining systems and machine learning pipelines are connected, which are used to process the metadata of the concrete solutions [FBD+15; Str15].

Another example that shows the realization of a solution repository via semantic web technology is shown by Krieger [Kri18c]. The solution repository is implemented as a Resource Description Framework (RDF) triple store. Using the capabilities of RDF to specify *subjects*, *objects*, and *relations* among them, concrete solutions are operationalized as sets of RDF-triples (cf. [Kri18c, p.47]). This enables to formulate concrete solutions based on the semantic web stack, which allows to form a whole ontology of concrete solutions, patterns, and connections among them.

Independently from specific domains, in the following, concrete solutions are formally introduced and defined. The informal definition of concrete solutions Definition 3.3.2 can be refined to a formal definition of concrete solutions as follows.

Definition 5.2.1 (Concrete Solution) Let $\mathcal{T}$ be the set of all concrete solution types, $\mathcal{D}_{PT}$ be the set of all domains of property types used to specify value ranges for type-specific descriptions of concrete solution properties, and $\mathfrak{S}$ be the set of all concrete solutions.

Then, a *Concrete Solution* is a tuple $s = (id, type, Props, \mathcal{D}_{PT}, \gamma, \delta) \in \mathfrak{S}$, s.t.,

(i) $id \in ID$, where ID is the set of all identifiers of concrete solutions

(ii) $type \in \mathscr{T}$

(iii) $Props$ is the set of all properties

(iv) $\forall s_i, s_j \in \mathfrak{S} : \pi_1(s_i) \neq \pi_1(s_j)$

(v) $\gamma : Props \rightarrow \wp(\mathscr{D}_{PT})$

(vi) $\delta : Props \rightarrow \bigcup_{prop \in Props} \times_{D \in \mathscr{D}_{PT_{\gamma(prop)}}} D$

(vii) $\forall prop \in Props : \delta(prop) \in \times_{D \in \mathscr{D}_{PT_{\gamma(prop)}}} D$

where γ is a map that assigns schemas to properties of concrete solutions and δ is a map that assigns values to properties from the respective domain of the property. The set of all properties *Props* specifies the schema and values of metadata describing a concrete solution. ■

While $\mathscr{T}$ allows to distinguish, for example, concrete solutions that are python programs from those that are costumes, the domains of property types $\mathscr{D}_{PT}$ provide domain-specific vocabulary to express the meta data of concrete solutions. For example, a domain $D_{color} \in \mathscr{D}_{PT}$ can be used to specify the valid set of colors to describe a costume, while a domain $D_{license} \in \mathscr{D}_{PT}$ can be used to specify the set of software licenses that can be assigned to concrete solutions. In summary, this structure enables to describe concrete solutions and managing them in solution repositories, such as those by Fehling et al. [FBFL15] or Krieger [Kri18c]. Thereby, they establish identity among all concrete solutions and capture all relevant metadata about them in structured properties. The properties are essentially data structures, which have to be implemented in solution repositories to capture and manage concrete solutions, which then ultimately enables to connect them later with patterns in a pattern repository.

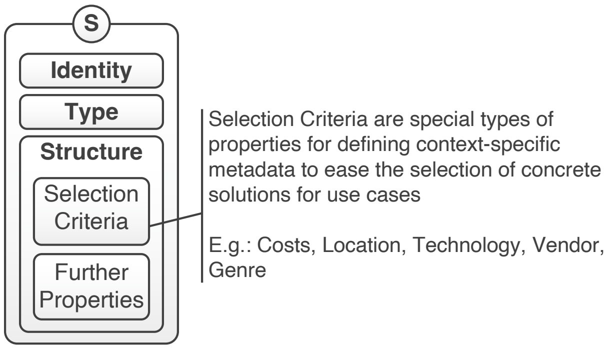

S
Identity
Type
Structure
Selection Criteria
Further Properties
Selection Criteria are special types of properties for defining context-specific metadata to ease the selection of concrete solutions for use cases
E.g.: Costs, Location, Technology, Vendor, Genre

Figure 5.3: Selection criteria of concrete solutions

5.3 Accessing Concrete Solutions via Selection Criteria

After a user selects a pattern, they encounter the challenge of determining which concrete solution adequately addresses their issue in their specific context. The context encapsules all aspects of the situation at hand, which are relevant to design and implement a solution consisting of multiple concrete solutions. This means, in order to balance all requirements and constraints present in a users context, selecting an appropriate and working set of concrete solutions is inevitable. To facilitate the appropriate selection of concrete solutions for a pattern, *Selection Criteria* are introduced, which help to identify when a particular concrete solution can be used.

Selection criteria are special types of properties of the concrete solutions and, therefore, incorporate additional metadata to them. They offer a way to guide the selection process using supplementary meta-information not included in the concrete solution itself. Selection criteria can be either human-readable or software-interpretable descriptions that indicate when to choose a specific concrete solution. Thus, they enable to add a layer of domain-specific metadata to concrete solutions aiming to ease their selection for use cases at hand, as depicted in Figure 5.3.

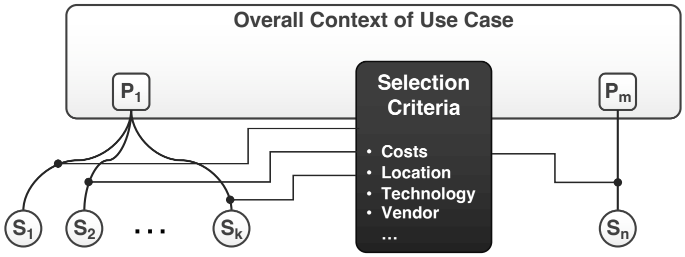


Figure 5.4: Selection criteria allow to map concrete solutions to use case-specific and, therefore, implementation-specific contexts

To illustrate the concept, the following example shows concrete solutions from the field of building architecture enriched by exemplary selection criteria. In this domain, as discussed by Alexander et al. [AIS77] and Alexander [Ale79], a concrete solution could be, e.g., an actual entrance of a building or a specific room layout of a floor, which can be described in detail through blueprints. Such blueprints capture all relevant aspects required to rebuild a concrete solution as an implementation of a pattern. Thus, connected to the corresponding patterns [AIS77; Ale79], blueprints can act as concrete solutions in this domain. To determine the most suitable concrete solution for a specific use case, selection criteria, such as the cost of the architectural concrete solution or the materials used, can be taken into account.

These criteria function as special properties of the concrete solutions. For instance, two concrete solutions for the pattern mentioned above, dealing with room layouts, might vary in the historical style of their construction and the used materials. In the field of cloud computing, selection criteria can specify the ability of a concrete solution being deployed on a specific cloud platform or even if the concrete solution is bound to a certain region, which might inflict compliance issues if not properly considered when realizing an IT-system in a specific use case. In the domain of costumes in films, selection criteria can be used, e.g., to specify the genre or a specific style of a film a costume is most suitable to.

Overall, selection criteria can be used to reflect characteristics of a concrete solution that are not directly included in the concrete solution itself, but are important for the selection of concrete solutions in the context of a specific use case, where typically multiple patterns and, therefore, also multiple concrete solutions are required (cf. Figure 5.4). In contrast to the abstract solution descriptions in patterns, selection criteria are used to make concrete solutions manageable and accessible for users via solution repositories. Selection criteria enable to add a layer of metadata to the pure description of a concrete solution, which can also incorporate information from the context in which a concrete solution is applied.

5.4 Requirements and Capabilities

In order to enable the combined use of concrete solutions it must be assured that compatible concrete solutions can be identified. More specifically, due to the fact that concrete solutions of a pattern can be implemented in many technologies, concrete solutions connected to different patterns in a pattern language can end up being not aggregatable at all. For instance, concrete solutions written in a programming language might not be combinable with components written in another programming language. Another example are proprietary deployment model fragments for different cloud providers. Thus, as a basis for the automated aggregation of concrete solutions, the concept of requirements and capabilities (cf. among others [OAS13; Wel94]) specified as properties of concrete solutions as depicted in Figure 5.5 can be used to represent dependencies between concrete solutions. On the one hand, requirements specify which capabilities have to be supported by other concrete solutions in order to be aggregatable with them. On the other hand, capabilities specify functionality or other properties a concrete solution supports.

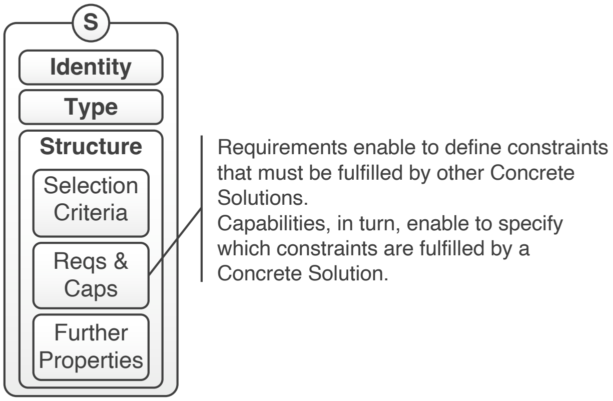

S
Identity
Type
Structure
Selection Criteria
Reqs & Caps
Further Properties
Requirements enable to define constraints that must be fulfilled by other Concrete Solutions.
Capabilities, in turn, enable to specify which constraints are fulfilled by a Concrete Solution.

Figure 5.5: Extension of the concrete solution metamodel by requirements and capabilities

For instance, considering the scenario of an *Elastic Load Balancer* that shall be used to scale *Processing Components* elastically in the environment of a specific cloud provider. In this case, a concrete solution of an *Elastic Load Balancer* can specify that it provides the capability of scaling specific *Processing Components* elastically in the environment of a cloud provider. Further, a concrete solution of *Processing Component* can specify the requirement that it in order to be launched and executed it requires another component that provides this as a capability. This concept corresponds to left-side and right-side interfaces as introduced by Reisig [Rei18] for the composition of components in workflows as well as preconditions and effects for the composition of Management Planlets by Breitenbücher et al. [BBK+13a; BBKL13] and Breitenbücher [Bre16].

This principle is illustrated conceptually in Figure 5.6. There, the concrete solution S_k is connected to pattern P_1 and the concrete solution S_n to pattern P_m. The concrete solution S_k specifies the two requirements *Req A* and *Req B*, which must be fulfilled by another concrete solution in order to be aggregatable with S_k. The concrete solution S_n specifies the

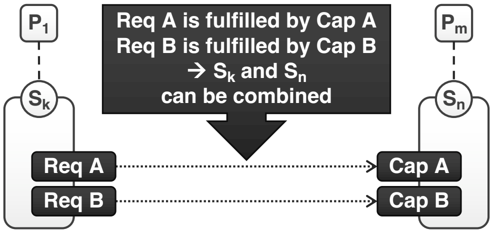


Figure 5.6: Compatibility of concrete solutions can be determined by their requirements and capabilities

respective capabilities *Cap A* and *Cap B*, thus, these two concrete solutions are compatible and can be aggregated. How the aggregation of concrete solutions can be implemented is described in Section 6.3.

5.5 Pattern Graph with Connected Concrete Solutions

Bringing patterns and concrete solutions conceptually together, the following section introduces the notion of a *Pattern Graph with connected Concrete Solutions* (PGCS). A PGCS is a pattern graph, which is refined from Definition 4.1.1 to include concrete solutions that are linked to the patterns they implement. A PGCS is defined as following.

Definition 5.5.1 (Pattern Graph with connected Concrete Solutions) Let $\mathcal{G} = (\mathcal{P}, \mathcal{E}_{\mathcal{P}}, \mathcal{W}, \mathcal{D}, \alpha, \beta)$ be a pattern graph. Then a *Pattern Graph with connected Concrete Solutions* $\mathcal{G}_{\mathcal{P}\mathcal{S}}$ is a tuple,

$$\mathcal{G}_{\mathcal{P}\mathcal{S}} = (\mathcal{P}, \mathcal{E}_{\mathcal{P}}, \mathcal{W}, \mathcal{D}, \alpha, \beta, \mathcal{S}, \mathcal{E}_{\mathcal{P}\mathcal{S}})$$

with

(i) $\mathcal{S} \subseteq \mathfrak{S}$

(ii) $\mathcal{E}_{\mathcal{P}\mathcal{S}} \subseteq \mathcal{P} \times \mathcal{S}$

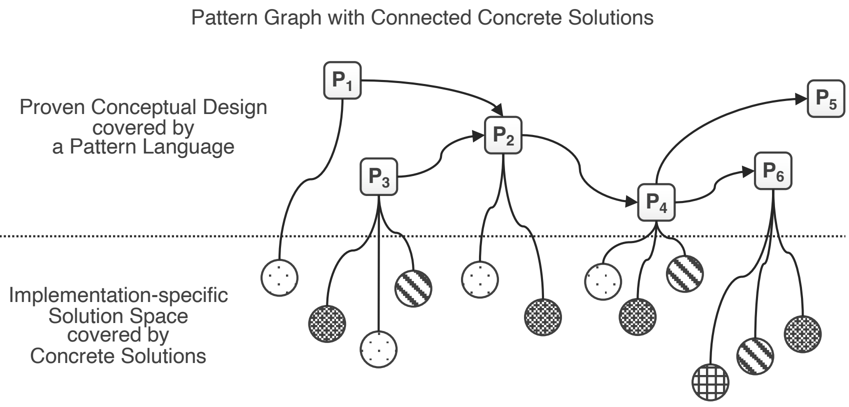

Pattern Graph with Connected Concrete Solutions
Proven Conceptual Design covered by a Pattern Language
Implementation-specific Solution Space covered by Concrete Solutions
P_1
P_2
P_3
P_4
P_5
P_6

Figure 5.7: A pattern graph with connected concrete solutions maps conceptual proven solutions in a pattern language to the implementation-centric solution space of implementations

(iii) $\forall s \in \mathcal{S} \exists! p \in \mathcal{P} : (p, s) \in \mathcal{E}_{\mathcal{P}\mathcal{S}}$

■

A PGCS connects the conceptual solutions covered by a pattern language with the implementation-centric design space covered by implementations of the patterns represented as concrete solutions. More specific, while a pattern language structures the conceptual design space into proven solution concepts, which are the patterns of the language (cf. [Zdu07]), a PGCS extends this to the *Solution Space* by connecting the patterns to proven concrete solutions, which implement them.

This means a PGCS enables to map proven conceptual solutions with reusable and proven implementations to ease the application within specific contexts and constraints. This mapping is ultimately depicted in Figure 5.7, which shows a PGCS with connected concrete solutions. There, concrete solutions of different types indicated by different hatchings are connected to the patterns they implement.

As introduced above, different types mean that the concrete solutions are implemented in non-compatible ways, e.g., in different technologies such as different programming languages. Such specifics are then captured in the respective selection criteria of the concrete solutions, which allows to exactly select those concrete solutions, which are compatible with the specific requirements a user is faced with.

5.6 Solution Models

In Section 4.4 the concept of design models was introduced, which enables a Designer to model conceptual solutions with patterns. According to Step 3 of the EINSTEIN-Method, an Implementer makes use of a design model to design the conceptual solution for a use case at hand. Thereby, they can make use of concrete solutions, which are connected to patterns as introduced in the previous section by means of a PGCS. To reuse concrete solutions of the patterns in a design model, the Implementer has to create a *Solution Model* from a design model, which is depicted conceptually in Figure 5.8 and introduced in the following. In contrast to a design model, a solution model consists of concrete solutions instead of patterns.

Thus, an Implementer has to decide which concrete solutions have to be selected to implement the patterns of the design model. In Figure 5.8, as a first step, pattern *B* is selected to by the Implementer to select a suitable concrete solution for. To find proper concrete solutions the patterns of the design model can be searched in a PGCS which then can be used to navigate from the patterns to concrete solutions implementing them – in case of the depicted scenario, an Implementer would search for pattern *B* in a PGCS to find a concrete solution that matches their needs. To limit the number of concrete solutions to a set relevant for a use case at hand, the Implementer can filter the concrete solutions by their selection criteria, type, requirements, and capabilities.

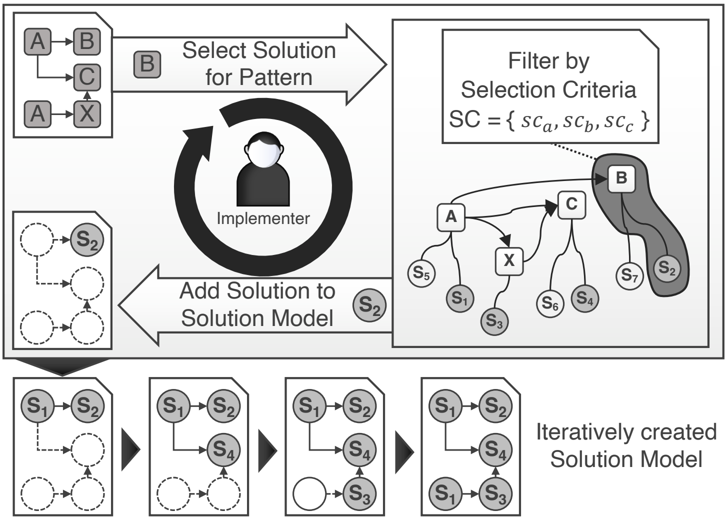

Select Solution for Pattern
Filter by Selection Criteria
SC = { sc_a, sc_b, sc_c }
Implementer
Add Solution to Solution Model
Iteratively created Solution Model

Figure 5.8: Modeling of a solution model based on a design model

As described in Section 5.3, an Implementer could, e.g, reduce all concrete solutions to only those that are implemented in a specific technology, or to those for a specific offering of a particular cloud provider. The Implementer can also omit specific filter criteria and navigate in the PGCS to the pattern of the design model to select a concrete solution.

Once the Implementer has selected a concrete solution for a pattern in the design model, the concrete solution is put in the solution model, as depicted in Figure 5.8 where concrete solution S_2 is selected for pattern B. The Implementer then can step to the next pattern in the design model to select a proper concrete solution likewise. Thereby, it can be checked if there are compatible solutions for the patterns connected to the replaced pattern by considering the requirements and capabilities of the concrete solutions. If there are no compatible solutions for the other patterns, the Implementer can either select a different concrete solution for the pattern or implement missing concrete solutions which is inline with the optional Step 4 of the EINSTEIN-Method in Figure 3.1. The Implementer creates the solution model iteratively until all patterns of the design model are represented by a concrete solution in the solution model, as illustrated as the outcome in Figure 5.8. Thereby, the Implementer also adds the respective edges between the concrete solutions in the solution model and assigns the weights from the design model which represent the semantics of the interplay of the concrete solutions. Informally speaking, this iterative selection of concrete solutions for the patterns of a design model can be grasped as *replacing* the patterns of the design model with concrete solutions resulting in a solution model.

Note, that in the case of composite patterns, the isomorphism between the design model and the solution model can get broken. This means, that the solution model is not necessarily isomorphic to the design model because a composite pattern can be replaced with a set of related concrete solutions. This corresponds to replacing graph fragments with other graph fragments, which is conceptually already solved for example in the domain of process models [Ebe14; Ma13; Sko17].

Falazi [Fal17a] shows in his work how the selection of concrete solutions from patterns can be supported by a query language working on a PGCS. He shows how a query language can be generated from a domain-specific grammar, which allows to filter concrete solutions of patterns in a PGCS by specifying constraints on their selection criteria, requirements, and capabilities. This assures, that only concrete solutions are left that provide the required characteristics to solve the use case at hand. As a result, Step 3 of the EINSTEIN-Method can be guided and semi-automated.

A solution model is a graph that consists of concrete solutions implementing the patterns of a design model. Therefore, to express the coupling of the concrete solutions in a solution model and the patterns in a design model the following definition of a the relation between the respective concrete solutions and patterns they implement is introduced. A solution model, worked out by an Implementer specifies the implementation of a design model by the interplay of concrete solutions solving a concrete problem at hand. Definition 5.6.1 shows that a solution model is implicitly related to a design model via a pattern graph with connected concrete solutions $\mathcal{G}_{\mathcal{P}\mathcal{S}}$, as already illustrated in Figure 5.8.

Definition 5.6.1 (Solution Model) Let $\mathcal{G}_{dm}$ be a design model and $\mathcal{G}_{\mathcal{P}\mathcal{S}}$ be a pattern graph with connected concrete solutions used to model $\mathcal{G}_{dm}$.

Then, a *Solution Model* is a graph $\mathcal{G}_{sm} = (\mathcal{S}_{sm}, \mathcal{E}_{sm}, \mathcal{W}_{sm}, \mathcal{D}, \alpha_{sm}, \beta_{sm})$, s.t.,

(i) $\mathcal{S}_{sm}$ is the multiset induced by the family $(s_j)_{j \in J}$ with $s_j \in \pi_7(\mathcal{G}_{\mathcal{P}\mathcal{S}})$ and $card(\mathcal{S}_{sm}) \in \mathbb{N}$ being the nodes

(ii) $\mathcal{E}_{sm} \subseteq \mathcal{S}_{sm} \times \mathcal{S}_{sm} \times \mathcal{W}_{sm}$

(iii) $\forall e \in \mathcal{E}_{sm} : \pi_1(e) \neq \pi_2(e)$

(iv) $\forall s \in \mathcal{S}_{sm} \exists! p \in \pi_1(\mathcal{G}_{dm}) \exists e \in \pi_2(\mathcal{G}_{\mathcal{P}\mathcal{S}}) : \pi_1(e) = p \wedge \pi_2(e) = s$

(v) $\mathcal{W}_{sm} \supseteq \pi_3(\mathcal{G}_{dm})$ is a set of weights representing domain-specific semantics of the interplay of concrete solutions

(vi) $\alpha_{sm} : \mathcal{W}_{sm} \rightarrow \wp(\mathcal{D})$

(vii) $\beta_{sm} : \mathcal{E}_{sm} \rightarrow \bigcup_{e \in \mathcal{E}_{sm}} \times_{D \in \mathcal{D}_{\alpha_{sm}(\pi_3(e))}} D$

(viii) $\forall e \in \mathcal{E}_{sm} : \beta_{sm}(e) \in \times_{D \in \mathcal{D}_{\alpha_{sm}(\pi_3(e))}} D$

■

5.7 Chapter Conclusion

This chapter introduced the concept of concrete solutions, which are implementations of patterns. Concrete solutions are reusable building blocks representing implementations of the solution principles documented in a pattern. A formalization of concrete solutions was introduced, which allows to systematically capture and manage concrete solutions in solution repositories based on an extendable metamodel. Thereby, it was defined that the structure of concrete solutions is domain-independent and, therefore, can be used to describe concrete solutions from different domains. In addition, the concept of selection criteria was introduced, which enables to add a layer of domain-specific metadata to concrete solutions aiming to ease their selection for use cases at hand. Further, requirements and capabilities were introduced as properties of concrete solutions, which enable to determine the compatibility of concrete solutions. The notion of a pattern graph with connected concrete solutions was introduced, which allows to map proven conceptual solutions with reusable and proven implementations to ease the application within specific contexts and constraints. Finally, it was shown how design models translated stepwise into solution models that represent an interplay of concrete solutions. This projects the conceptual solution designed in a design model to the level of reusable concrete solutions. As a result, the concepts presented in this chapter support Step 3 of the EINSTEIN-Method.

CHAPTER 6

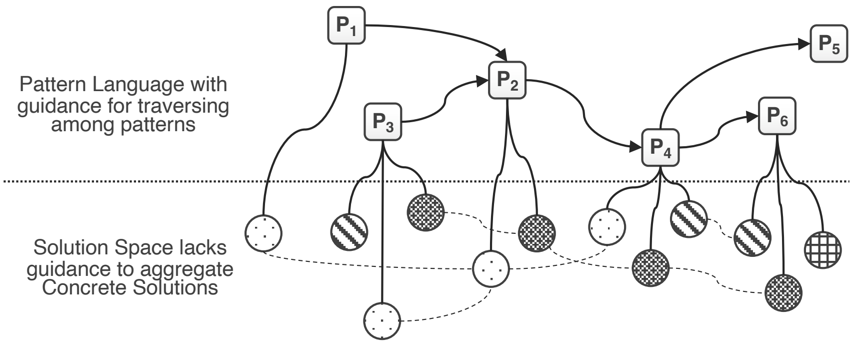
AGGREGATION OF CONCRETE SOLUTIONS

In the previous chapter, it has been shown how pattern graphs with connected concrete solutions enable to map proven conceptual solutions of pattern languages to proven implementations represented as concrete solutions. As they can be used to refine conceptual solutions captured in design models into solution models, it gets relevant to enable the *combined reuse of the concrete solutions*. This is because the selection of individual concrete solutions based on selection criteria is not enough for the development of complex systems because the individual concrete solutions are not yet integrated into a single overall solution. Therefore, to support Step 5 and Step 6 of the EINSTEIN-Method, in this chapter, the concept of aggregation is introduced in the form of *Solution Algebras*, which enable to formalize the aggregation of concrete solutions by means of *Aggregation Operators*. Then, two approaches are introduced that enable to realize the aggregation of concrete solutions. The first one is based on *Solution Languages* containing *Concrete Solution Aggregation Descriptors*, which enable to provide guidance to manually aggregate concrete solutions. The second one shows how the aggregation of concrete solutions can be automated by means of *Concrete Solution Aggregation Programs*. Thus, both approaches are implementations of the concept of aggregation operators, with concrete solution aggregation descriptors being manual realizations of aggregation operators and concrete solution aggregation programs being automated realizations, respectively. Thereby, this chapter builds on the work by Beisel [Bei17], Falkenthal et al. [FBB+14a; FBB+14b; FBBL17; FBBL19], Falkenthal and Leymann [FL17], and Krieger [Kri18c].

6.1 Solution Algebras

The previous chapter described how concrete solutions can be linked with pattern languages and how they can be selected for reuse based on selection criteria. The illustration in Figure 5.7 presented a pattern language as a network of interconnected patterns. Inherent in these patterns is the conceptual solution knowledge that opens a realm of potential implementations, i.e., the solution space of the whole pattern language. Concrete solutions, being specific implementations of individual patterns in the pattern language, are thus contained within this solution space. They are linked to the patterns they implement facilitating their reuse upon selection of a pattern from the pattern language for application. However, as the interplay of patterns in a pattern language and ultimately also in a design model reflect that solution principles described by the patterns have to be combined to solve an overall problem, this has to be also projected to the solution models elaborating the interplay of concrete solutions. Thus, an approach is required that guides users through the solution space to aggregate concrete solutions with overall ones.

This is depicted in Figure 6.1, where the solution space of a pattern language is shown as a set of concrete solutions, which are connected to the patterns they implement. However, currently guidance is missing which enables to aggregate the concrete solutions as indicated by the dotted lines



Pattern Language with guidance for traversing among patterns
Solution Space lacks guidance to aggregate Concrete Solutions
P_1
P_2
P_3
P_4
P_5
P_6

Figure 6.1: The solution space of a pattern language lacks guidance to aggregate concrete solutions as indicated by the dashed lines

between the concrete solutions. Therefore, to enable the combination of concrete solutions, it is necessary to support aggregations among them. For this purpose, it is necessary to apply a structure on the solution space of a pattern language by means of aggregations that enable to combine concrete solutions. This is achieved by introducing the notion of *Solution Algebras*, which are used to treat concrete solutions as mathematical objects on which *Concrete Solution Aggregation Operators*, or *Aggregation Operators* in short, can be applied. Aggregation operators are the means to operationalize the aggregation of concrete solutions. In order to enable the aggregation of a multitude of concrete solutions, typically different aggregation operators are required that are capable of aggregating different types of concrete solutions.

Concrete solutions sharing common characteristics, such as computer programs that are implemented in the same programming language, can be grouped into sets based on their types. Thereby, concrete solutions with the same type can potentially be combined based on compatible aggregation operators. For example, code snippets written in the same programming language share this characteristic, which makes them likely combinable to a larger code base. Concrete solutions of different types often differ in their essential characteristics, such as the technology they are implemented with or the domain which they stem from. For example, code snippets (domain *software development*), costumes (domain *costume design*), or building architecture sketches (domain *building architecture*) are not compatible at all and, therefore, are of different types. Types of concrete solutions provide a first structure on the solution space of a pattern language in terms of sets of concrete solutions.

Definition 6.1.1 (Types of Concrete Solutions) The set of concrete solutions $\mathfrak{S}$ can be partitioned into a family of sets $(A_t)_{t\in\Theta}$ with $\forall t \in \Theta : A_t \subseteq \mathfrak{S}$. Thereby, Θ is a set whose elements identify all types of concrete solutions. Then each A_t represents a *Type* of concrete solutions. ■

To express that concrete solutions can be combined with each other, the concept of aggregation operators is combined with the concept of types of concrete solutions. By utilizing aggregation operators, a structure of composable concrete solutions emerges as a solution algebra.

Definition 6.1.2 (Solution Algebra and Aggregation Operators) A *Solution Algebra* is a pair $((A_t)_{t\in\Theta}, (\oplus_j)_{j\in J})$ consisting of a family of concrete solutions $(A_t)_{t\in\Theta}$ and a family $(\oplus_j)_{j\in J}$ of *Aggregation Operators*, s.t.,

(i) $J \subseteq \bigcup\limits_{k=2}^{\infty} \underbrace{\Theta \times \ldots \times \Theta}_{(k+1)-times}$

(ii) $\oplus_j : A_{t_1} \times \ldots A_{t_k} \rightarrow A_{t_m}$ with $j = (t_1, \ldots, t_k, t_m) \in J$

■

Typically multiple solution algebras are required to cover the entire solution space of a pattern language. Thereby, solution algebras structure the solution space of a pattern language into subsets of concrete solutions called types for which aggregation operators exist that are capable of aggregating them. Thus, by means of solution algebras it is possible to structure the solution space into cohesive parts. This is conceptually illustrated in Figure 6.2, where concrete solutions are grouped into three sets of concrete solutions, which form two different solution algebras with their respective aggregation operators. The solution algebra $(\{\{s_1, s_2\}_\alpha\}, \{\oplus_{(\alpha,\alpha,\alpha)}\})$ specifies a pair, consisting of the family $\{\oplus_{(\alpha,\alpha,\alpha)}\}$ containing just one aggregation operator, which can aggregate the concrete solutions s_1 and s_2 as $\oplus_{(\alpha,\alpha,\alpha)} : \{s_1, s_2\}_\alpha \times \{s_1, s_2\}_\alpha \rightarrow \mathfrak{S}$. Note that the types of the concrete solutions are omitted in Figure 6.2 for brevity but are specified by the index of the operator. The second depicted solution algebra $(\{\{s_4, s_6\}_\beta, \{s_7, s_8\}_\gamma\}, \{\oplus_{(\beta,\beta,\beta)}, \oplus_{(\gamma,\gamma,\gamma)}\})$ specifies a separate family of aggregation operators on the types β and γ of concrete solutions. In this case, two aggregation operators are defined, whereby $\oplus_{(\beta,\beta,\beta)} : \{s_4, s_6\}_\beta \times \{s_4, s_6\}_\beta \rightarrow \mathfrak{S}$ and $\oplus_{(\gamma,\gamma,\gamma)} : \{s_7, s_8\}_\gamma \times \{s_7, s_8\}_\gamma \rightarrow \mathfrak{S}$. Note that Figure 6.2 also shows that no aggregation operator is defined that can aggregate s_6 and s_7. This can be due to the fact that the solution algebra is still under development or if the development of new technologies lead to new types of concrete solutions for which then also new aggregation operators have to be defined.

To grasp this concept more intuitively, at this point, the example of scaling processing components elastically to meet the requirements of actual workloads, as introduced in Section 4.2.1, will help. To recap, the scenario is described by the two patterns *Elastic Load Balancer* and *Stateless Component* by Fehling et al. [FLR+14]. The pattern *Elastic Load Balancer* describes the concept of a load balancer, which distributes incoming requests to a set of components, which are elastically scaled based on the monitoring of actual workloads. AWS provides an implementation of this pattern as a service, which can be managed and instrumented via CloudFormation templates [Ama23b], which is the Infrastructure-as-Code language by AWS that enables to specify cloud resources using JSON. A component implementing the business logic to be executed can be realized as described in the pattern *Stateless Component* and implemented as an Amazon Ma-

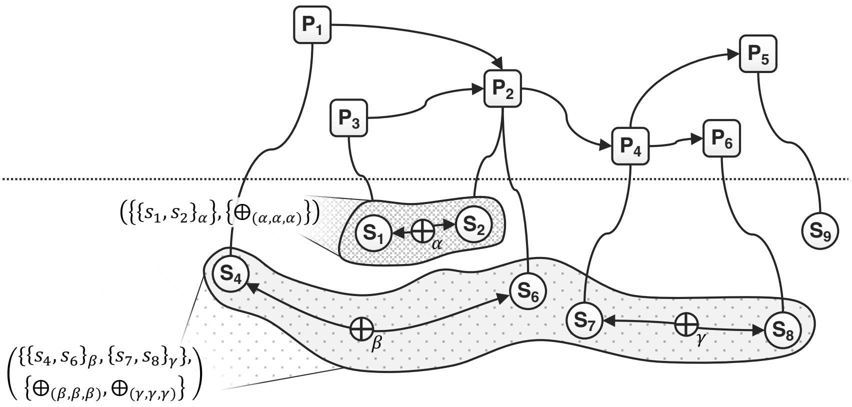

P_1
P_2
P_3
P_4
P_5
P_6
$(\{\{s_1, s_2\}_\alpha\}, \{\oplus_{(\alpha,\alpha,\alpha)}\})$
S_1
$\oplus_\alpha$
S_2
S_9
S_4
S_6
S_7
$\oplus_\gamma$
S_8
$\oplus_\beta$
$\begin{pmatrix}\{\{s_4, s_6\}_\beta, \{s_7, s_8\}_\gamma\}, \\ \{\oplus_{(\beta,\beta,\beta)}, \oplus_{(\gamma,\gamma,\gamma)}\}\end{pmatrix}$

Figure 6.2: Solution algebras structure aggregation operators and the set of concrete solutions they are capable to operate on

chine Image (AMI) [Ama23a]. The AMI can be referenced in a so-called *LaunchConfiguration*, which is a CloudFormation snippet allowing to reference AMIs as the source of the scaled components. As a consequence, both, a concrete solution of the *Elastic Load Balancer* as well as a concrete solution of the *Stateless Component*, can be implemented as CloudFormation snippets, which are valid JSON-documents. To combine both concrete solutions, an aggregation operator $\oplus_{(cf,cf,cf)} : A_{cf} \times A_{cf} \rightarrow A_{cf}$ can be defined that is capable of combining CloudFormation JSON-documents to a single CloudFormation JSON-document. Thereby, the aggregation operator understands the domain-specific semantics of CloudFormation templates and can combine the CloudFormation snippets accordingly by placing both snippets in a single JSON-document and aggregating them by adding a so-called *Autoscaling Group* which allows to reference both, the Elastic Load Balancer snippet as well as the LaunchConfiguration snippet. The resulting CloudFormation template, thus, combines the formerly disconnected concrete solutions and represents a concrete solution aggregate that can be deployed to the AWS cloud environment.

However, this example can also be slightly modified to show that aggregation operators can also operate on different sets of concrete solutions. It is also possible to define another aggregation operator $\oplus_{(cf,ami,cf)} : A_{cf} \times A_{ami} \rightarrow A_{cf}$ that understands the domain-specific semantics of CloudFormation and AMIs as well to combine both. In this case, the implementation of the Stateless Component can also be realized as an AMI, without referencing it in a LaunchConfiguration snippet, which means the concrete solution is a pure AMI. Then, the aggregation operator can combine the AMI with the CloudFormation snippet of the Elastic Load Balancer by wrapping the AMI itself into a LaunchConfiguration snippet, merging it into the resulting CloudFormation template, and referencing it in the mentioned Autoscaling Group accordingly. This scenario shows that aggregation operators can implement domain-specific semantics that allow to aggregate concrete solutions of different types, which means that they can work on different carrier sets, which is inline with Definition 6.1.2. Of course, this is only possible if the aggregation operator understands the domain-specific semantics of the concrete solutions it operates on, which shows that aggregation operators have to consider domain knowledge. In conclusion, solution algebras represent exactly the coupling and cohesion of domain-specific concrete solutions with aggregation operators that incorporate knowledge of the domains the concrete solutions stem from, which is required to aggregate concrete solutions.

Algebraic properties, such as commutativity and associativity, of aggregation operators can not be assumed in general, but are a result of the structure of the solution space of a pattern language and the domain semantics of the concrete solutions. This is exemplified by the following examples. Firstly, the aggregation operator $\oplus_{(cf,cf,cf)}$ from the example above is commutative, because the order of the CloudFormation documents as operands does not matter when referencing both in the Autoscaling Group. But it is not associative because when aggregating multiple CloudFormation documents the overall achieved semantics can be different depending on the order of aggregation steps. For example, if two Elastic Load Balancers are available and each of them shall scale a different Stateless Component, then the aggregation order has to assure that the each Elastic Load Balancer has to be aggregated with the corresponding Stateless Component.

In contrast, the discussion of commutativity and associativity is irrelevant for the aggregation operator $\oplus_{(cf,ami,cf)}$, because the operator is defined on two Sorts and, therefore, has to understand which operand is the AMI to wrap it in a LaunchConfiguration. By extending the discussion to other domains, further concrete solutions and aggregation operators impose further algebraic properties. For instance, in the domain of cloud application management (details are discussed in Section 8.2.2) special single-entry-single-exit (SESE) workflows are used to manage the lifecycle of cloud applications. Due to the SESE characteristics, an aggregation operator can be defined that aggregates those workflows, which is associative but not commutative. This is because the order of the workflows has to be preserved to reflect their intention which can only be assured if the workflows are aggregated in the right order. All these examples show that the algebraic properties of aggregation operators are highly domain-specific and, therefore, must be reflected in solution algebras according to each domain.

Nevertheless, the general guideline — to strive for *associativity* for aggregation operators — can be deduced from other works, which investigate the aggregation of system components [FR21; FR23] or system management flows [Bre16]. This is mainly, because associativity guarantees that the order of aggregation steps does not matter, which assures that the aggregation among all concrete solutions in a solution model always results in the same overall solution aggregate. But, as seen by the above examples, this can only be suggested as a guideline in order to not limit the general applicability of the presented concepts to different domains. When the concepts of this work are applied to a specific domain, the aggregation operators can be analyzed for associativity and the algebraic properties can be defined accordingly, which allows to implement the required semantics from a specific domain in a solution algebra.

Finally, solution algebras do not necessarily partition the solution space completely, since there may be concrete solutions that are not contained in any carrier set of specified solution algebras for a pattern language, as exemplified by concrete solution S_9 on the very right in Figure 6.2. This can be the case, for example, when concrete solutions are implemented in newly emerging technologies and do not fit to already present ones in the solution space or if a pattern language is still in its authoring process.

In the next two sections, it is shown how the conceptual approach of solution algebras can be realized. Firstly, in the next section, a manual approach is introduced, which is based on concrete solution aggregation descriptors and solution languages. After that, in the following section, an automated approach is introduced in the form of concrete solution aggregation programs.

6.2 Solution Languages and Concrete Solution Aggregation Descriptors

The previous section has shown that projecting the combined use of patterns to the level of concrete solutions requires the concept of Aggregation of concrete solutions. In the following, the concept of *Solution Languages* is introduced as a manual approach to implement the concept of concrete solution Aggregation. The primary objective of solution languages is to bring the navigation capabilities of pattern languages to the level of concrete solutions while supporting their manual aggregation, thus, they materialize possible aggregations in the solution space by text. First, solution languages introduce *semantically typed links* between concrete solutions annotated with meta data to aid users in determining the relevance of linked concrete solutions for resolving their use case at hand. This is particularly crucial when multiple patterns and, therefore, several concrete solutions need to be combined. The semantics of a link can, e.g., determine that concrete solutions related to different patterns can be combined, that specific concrete solutions are implementation variations of the same pattern, or if only one of several alternative concrete solutions can be used in conjunction with another.

If required, additional link semantics can be integrated into a solution language. For instance, semantic links that specifically indicate that chosen concrete solutions must not be combined can be included as well. This is beneficial, for example, when concrete solutions could technically be combined, but their implementation of non-functional attributes hinders the creation of a suitable combined solution. Such scenarios might occur, e.g., in the domain of cloud computing, where applications can be deployed among different cloud providers worldwide. Restrictions to distribute certain application components to specific countries can be due to legal regulations or company compliance policies [BBK+13b]. Therefore, it is useful to document these limitations at the level of concrete solutions via the mentioned link semantics preventing users from unnecessarily navigating to irrelevant concrete solutions.

Linking concrete solutions by means of semantically typed links already enables users to identify and understand how concrete solutions are related with each other. However, this does not provide any guidance on how to combine concrete solutions. Therefore, the concept of a *Concrete Solution Aggregation Descriptors* (CSADs) is introduced, which enables to annotate links between concrete solutions with additional documentation outlining in a human-readable way how to aggregate the linked concrete solutions. This documentation can range from a detailed description of the steps needed to combine the linked concrete solutions, to sketches of the overall solution resulting from the aggregation supporting the user. As a result, CSADs enable the inclusion of any relevant documentation into a solution language regarding the aggregation of concrete solutions. This enables the iterative creation and enhancement of a solution language over time preserving the expert knowledge of a domain at the implementation level, similarly to how pattern languages capture conceptual solution knowledge by interrelated patterns. In situations where technologies become obsolete and experts need to maintain systems implemented in those technologies

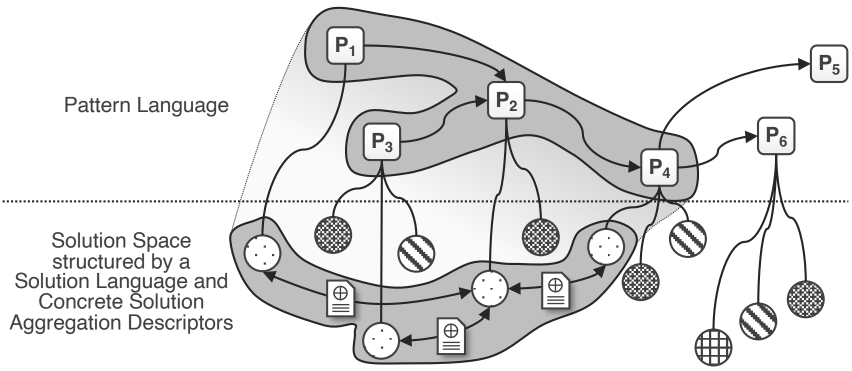

Pattern Language
Solution Space structured by a Solution Language and Concrete Solution Aggregation Descriptors
P1
P2
P3
P4
P5
P6

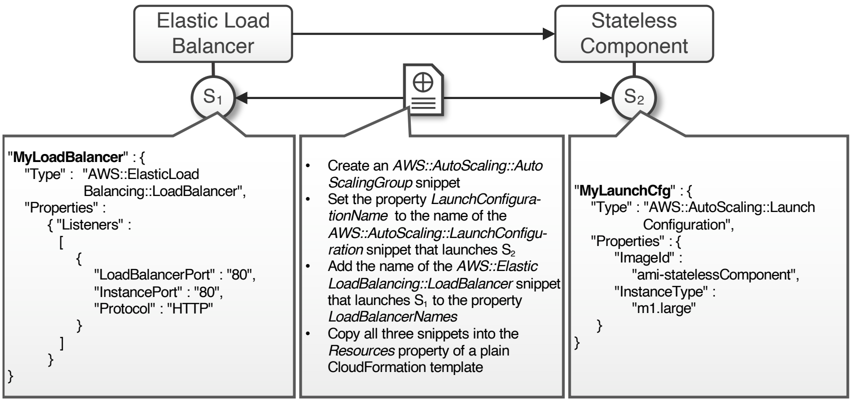
Figure 6.3: Pattern graph with connected solution language

are rare, such as in the case of the outdated programming language Cobol that is still the basis of many enterprise applications, solution languages are a candidate to preserve technology-specific implementation expertise and aggregation documentation over time.

Figure 6.3 illustrates the overall concept of a solution language, where concrete solutions are linked to the patterns they implement facilitating user navigation from patterns to reusable concrete implementations. Besides, concrete solutions are also related enabling navigation at the level of concrete solutions. Although, for the sake of brevity, Figure 6.3 focuses on links representing *can be aggregated with* semantics, the relations between the concrete solutions can capture further semantics in the CSADs attached to the relations in order to ease and guide the reuse of concrete solutions and their aggregation by Implementers as discussed above.

To give an example, the scenario described above of an *Elastic Load Balancer* scaling a *Stateless Component* is revisited. Figure 6.4 shows an exemplary CSAD, which describes how to combine the concrete solutions linked to the two patterns *Elastic Load Balancer* and *Stateless Component* of the pattern language by Fehling et al. [FLR+14]. To realize this setup by means of the Infrastructure-as-Code language CloudFormation [Ama23b] for AWS, concrete solutions for both patterns can be implemented as follows. Firstly, a concrete solution for *Elastic Load Balancer* can be implemented as a CloudFormation snippet as depicted on the left in Figure 6.4. The snippet specifies that a load balancer service instance of AWS has to be configured to consume HTTP traffic on port 80. Secondly, a concrete solution for *Stateless Component* can be implemented as a CloudFormation snippet as depicted on the right in Figure 6.4. The snippet specifies a LaunchConfiguration that describes how to launch instances of a stateless component in the AWS cloud by defining properties of virtual machines to be launched. Thereby, the property *ImageId* references a virtual machine image in the form of an Amazon Machine Image (AMI) [Ama23a], which implements the *Stateless Component*. How both concrete solutions can be combined to an concrete solution aggregate is described by the CSAD in the middle of Figure 6.4. The depicted CSAD details the steps an Implementer has to conduct to combine the concrete solutions of the two patterns using the provided CloudFormation snippets. The result is an overall CloudFormation template that can be used to automatically deploy the whole setup via AWS CloudFormation. Of course, if the combined concrete solutions result in an aggregate, one could think about saving this final aggregate also in the CSAD for reuse. However, if one of the combined concrete solutions needs to be manually changed significantly then it might be easier to re-execute the CSAD instead of adapting the final result. Therefore, the CSAD as well as the final aggregates are efficiently reusable depending on the concrete use case at hand. Further examples of CSADs can be found in the work by Falkenthal et al. [FBBL17].

As a result, a solution language projects the combinability of solution concepts of patterns to the level of concrete solutions, thereby, it structures and organizes the collection of available concrete solutions. Thus, it is an approach that implements the concept of concrete solution aggregation by providing guidance for manual aggregations of concrete solutions by means of concrete solution aggregation descriptors. They can be used to support the structuring of the solution space of a pattern language and to guide users through the solution space, which especially provides support



```
"MyLoadBalancer" : {
   "Type" :  "AWS::ElasticLoad
             Balancing::LoadBalancer",
   "Properties" :
      { "Listeners" :
         [
            {
               "LoadBalancerPort" : "80",
               "InstancePort" : "80",
               "Protocol" : "HTTP"
            }
         ]
      }
}
```

- Create an *AWS::AutoScaling::AutoScalingGroup* snippet
- Set the property *LaunchConfigurationName* to the name of the *AWS::AutoScaling::LaunchConfiguration* snippet that launches S_2
- Add the name of the *AWS::ElasticLoadBalancing::LoadBalancer* snippet that launches S_1 to the property *LoadBalancerNames*
- Copy all three snippets into the *Resources* property of a plain CloudFormation template

```
"MyLaunchCfg" : {
   "Type" : "AWS::AutoScaling::Launch
             Configuration",
   "Properties" : {
      "ImageId" :
         "ami-statelessComponent",
      "InstanceType" :
         "m1.large"
   }
}
```

Figure 6.4: An example of a CSAD describing how to combine CloudFormation snippets for an *Elastic Load Balancer* that scales a *Stateless Component*

for creating solution models from design models and aggregating the contained concrete solutions manually. Prototypical implementations of a solution repository that enable to create and manage solution languages were elaborated in the works by Beisel [Bei17] and Krieger [Kri18a]. However, this approach is limited to the manual aggregation of concrete solutions, which is not efficient in cases when the aggregation of concrete solutions has to be conducted over and over again. Therefore, the next section introduces concrete solution aggregation programs, which enable to automate the aggregation of concrete solutions.

6.3 Concrete Solution Aggregation Programs

As described above, CSADs are a means to provide the necessary documentation to aggregate concrete solutions in a manual way. However, in cases when the aggregation of concrete solutions has to be conducted over and over again, it is beneficial to add a mechanism, which enables to automate the aggregation. In order to automate the aggregation of concrete solutions, the aggregation logic needs to be formalized and also operationalized. To achieve this, the concept of *Concrete Solution Aggregation Programs*, in short *Aggregation Programs*, is introduced below for the automated aggregation of concrete solutions, which corresponds to the automation of CSADs to automatically aggregate concrete solutions in a solution model.

According to Definition 5.6.1, a solution model is a graph of concrete solutions, thus, the aggregation of contained concrete solutions can be mapped to the problem of identifying subgraphs in the solution model that can be aggregated by an aggregation program. Thereby, the set of all possible subgraphs having at least two nodes in a solution model is called the *Solution Aggregation Space* (*SAS*) of the solution model. Thus, to aggregate concrete solutions in a solution model, an aggregation program must (i) identify subgraphs of the solution model, which can be aggregated by the program, and (ii) provide the logic to aggregate the concrete solutions in these subgraphs.

An aggregation program that can aggregate concrete solutions in a solution model is defined as following, whereby, the elements used in the definition are subsequently introduced and explained.

Definition 6.3.1 (Concrete Solution Aggregation Program) Let $\mathfrak{AP}$ be the set of all aggregation programs. A *Concrete Solution Aggregation Program* $\oplus \in \mathfrak{AP}$, or *Aggregation Program* in short, is a tuple $\oplus = (DG, \kappa, o, \sigma)$, where

(i) $DG \in \mathfrak{DG}$ is a Detector Graph (Definition 6.3.3)

(ii) κ is an Aggregatability Analysis Function (Definition 6.3.5)

(iii) o is an Aggregation Operation Function (Definition 6.3.6)

(iv) σ is a Solution Model Update Function (Definition 6.3.7)

■

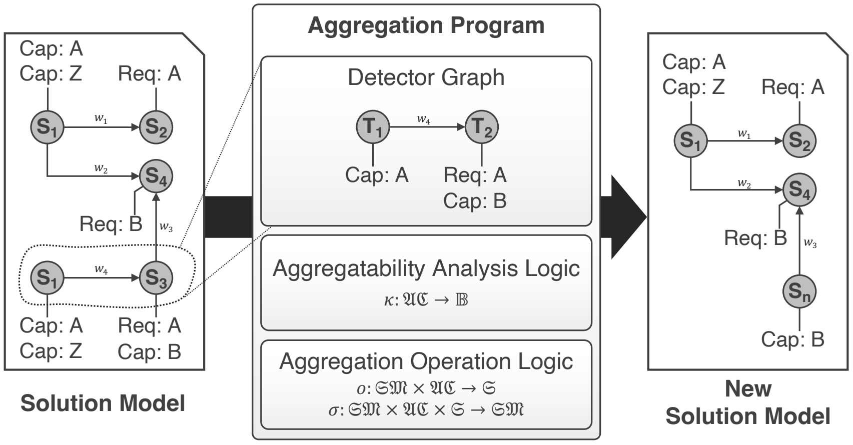

Cap: A
Cap: Z
Req: A
S_1
w_1
S_2
w_2
S_4
Req: B
w_3
S_1
w_4
S_3
Cap: A
Cap: Z
Req: A
Cap: B
Solution Model
Aggregation Program
Detector Graph
T_1
w_4
T_2
Cap: A
Req: A
Cap: B
Aggregatability Analysis Logic
$\kappa\colon \mathfrak{AC} \rightarrow \mathbb{B}$
Aggregation Operation Logic
$o\colon \mathfrak{SM} \times \mathfrak{AC} \rightarrow \mathfrak{S}$
$\sigma\colon \mathfrak{SM} \times \mathfrak{AC} \times \mathfrak{S} \rightarrow \mathfrak{SM}$
Cap: A
Cap: Z
Req: A
S_1
w_1
S_2
w_2
S_4
Req: B
w_3
S_n
Cap: B
New Solution Model

Figure 6.5: Concept of a concrete solution aggregation program

Figure 6.5 summarizes the principle of an aggregation program conceptually. On the left, a solution model is shown on which the aggregation program depicted in the middle has to be applied. The identification of subgraphs in the solution model that can be aggregated by the aggregation program is performed in two steps: firstly, the aggregation program identifies subgraph candidates in the SAS based on the graph structure of the solution model and, secondly, the aggregation program validates for each identified candidate whether it can successfully perform the aggregation. To identify subgraph candidates in the solution model, an aggregation program defines a *Detector Graph* with placeholders for concrete solutions as the nodes. The nodes of a detector graph are called *Operation Targets* and enable to specify a filter for concrete solutions based on the requirements, capabilities, and types of the concrete solutions in the solution model, as depicted exemplarily by the detector graph in Figure 6.5 consisting of the operation targets T_1 and T_2.

Definition 6.3.2 (Operation Target) Let $\mathcal{OT}$ be the set of all operation targets. An *Operation Target* is a concrete solution placeholder represented as a tuple $ot = (R, C, t)$, s.t.,

(i) R is a set of requirements

(ii) C is a set of capabilities

(iii) $t \in \mathcal{T}$ ■

Definition 6.3.3 (Detector Graph) A *Detector Graph* $DG = (OT, \mathcal{E}_{DG})$ is a graph, s.t.,

(i) $OT \subseteq \mathcal{OT}$

(ii) $\mathcal{E}_{DG} \subseteq OT \times OT \times \mathcal{W}_{sm}$

(iii) $\forall e \in \mathcal{E}_{DG} : \pi_1(e) \neq \pi_2(e)$

(iv) $\forall ot \in OT\ \exists e \in \mathcal{E}_{DG} : \pi_1(e) = ot \vee \pi_2(e) = ot$

(v) $ot_1, ot_2, \ldots, ot_k \in OT$ is a path (as symbol $\omega(ot_1, ot_k)$) from operation target ot_1 to operation target ot_k $:\Leftrightarrow$
$(ot_1, ot_2, w_{1,2}), (ot_2, ot_3, w_{2,3}), \ldots, (ot_{k-1}, ot_k, w_{k-1,k}) \in \mathcal{E}_{DG}$

(vi) $\forall ot, ot' \in OT\ \exists ot_1, \ldots, ot_k \in OT : ot_1 = ot \wedge ot_k = ot' \wedge \omega(ot_1, ot_k)$

■

A detector graph is used to identify subgraphs of a solution model, which are candidates to be aggregated by an aggregation program. Thereby, a concrete solution matches with an operation target if the types correspond and the concrete solution provides at least the requirements and capabilities defined by the operation target. The edges of the detector graph are weighted to reflect the domain-specific semantics describing the relations between concrete solutions as a further filter criteria. Thus, a detector graph of an aggregation program acts as a selector for subsets of the whole

SAS of a solution model, which can be potentially aggregated by the aggregation program. Compatible subsets of the SAS can be identified by searching for colored and weighted subgraph isomorphisms of the detector graph in the solution model. In Figure 6.5, the detector graph is used to identify the subgraph containing S_1 and S_3 in the solution model, because both provide at least the requirements and capabilities defined by T_1 and T_2 of the detector graph and also the weight of the edge connecting S_1 and S_3 corresponds to the weight connecting T_1 and T_2 in the detector graph. Thus, the subgraph containing S_1 and S_3 is isomorphic to the detector graph and, therefore, is identified as a candidate to be aggregated by the aggregation program. Such candidates are called *Aggregation Context*.

Definition 6.3.4 (Aggregation Context) Let $\mathfrak{SM}$ be the set of all solution models, let $\mathcal{G}_{sm} \in \mathfrak{SM}$ be a solution model, let $ap \in \mathfrak{AP}$ be an aggregation program, and let $\mathfrak{DG} \ni DG = \pi_1(ap)$ be the detector graph of ap.

Let $\mathfrak{F}(DG, \mathcal{G}_{sm})$ be the set of all subgraph isomorphisms of DG in $\mathcal{G}_{sm}$.

Then, an *Aggregation Context ac* is the set of pairs

$$\{(ot, s) | ot \in \pi_1(DG), s = f(ot)\}$$

induced by a subgraph isomorphism $f \in \mathfrak{F}(DG, \mathcal{G}_{sm})$ that maps the operation targets of DG to concrete solutions in $\mathcal{G}_{sm}$. ■

An aggregation context represents a subgraph isomorphism of a detector graph in a solution model. An example of an aggregation context is depicted in Figure 6.5, where the detector graph is mapped to the subgraph containing S_1 and S_3 in the solution model, which is then identified as the aggregation context the aggregation program can operate on. To validate if an aggregation program can aggregate the subgraph of a solution model identified in an aggregation context *ac*, it must investigate if the contained concrete solutions provide all necessary properties the aggregation program requires to perform the aggregation, which is necessary because the automated aggregation of concrete solutions is highly domain-specific and requires sophisticated domain knowledge. For this validation, an aggregation program provides an *Aggregatability Analysis Logic* κ that validates if an aggregation program can aggregate an aggregation context of a given solution model. Therefore, the logic maps the aggregation context to *true* if the analysis concludes that the aggregation program can aggregate the aggregation context and to *false* otherwise.

Definition 6.3.5 (Aggregatability Analysis Function) Let $\mathfrak{AC}$ be the set of all aggregation contexts. Then, an *Aggregatability Analysis Function* $\kappa : \mathfrak{AC} \rightarrow \mathbb{B}$, is a function, that provides domain-specific logic to validate if an aggregation program can aggregate an aggregation context of a solution model. ■

Although this could be achieved as well by extensively modeling concrete solutions and their relations to each other via requirements and capabilities and, at the same time, by also expressing the compatibility of aggregation programs to solution model subgraphs by sophisticated detector graphs, this would be tedious and would move the complexity of the aggregation logic to the modeling of the concrete solutions, solution models, and the detector graph of aggregation programs. Thus, to encapsulate this in aggregation programs, where also the logic to aggregate the concrete solutions resides makes particularly sense because the aggregation logic is domain-specific and operates on the concrete artifacts of the concrete solutions and not only on their representations in the solution model as already indicated by the CloudFormation snippets above (cf. Section 6.2). As a result, logic must be written that aggregates the concrete artifacts of the concrete solutions, whereby the aggregation logic must check whether it can process the available artifacts of the concrete solutions. For example, aggregation logic that is capable of aggregating CloudFormation snippets would check if the given inputs are actually valid CloudFormation snippets. Thus, exactly this validation logic, which is needed anyway, can also be used and extended to validate the compatibility of the aggregation program with an aggregation context and, thereby, is then also given in the same domain-specific coding as the aggregation logic, which makes both understandable with the same expertise.

Finally, as already mentioned above, an aggregation program provides the *Aggregation Operation Logic* to aggregate the concrete solutions in the identified aggregation context. The aggregation logic is implemented in the *Aggregation Operation Function o* and the *Solution Model Update Function σ*. The aggregation operation function o aggregates the artifacts of concrete solutions and creates a new concrete solution, while the solution model update function σ updates the solution model with the new concrete solution.

Definition 6.3.6 (Aggregation Operation Function) An *Aggregation Operation Function o* : $\mathfrak{SM} \times \mathfrak{AC} \rightarrow \mathfrak{S}$, is a function that provides domain-specific logic to aggregate concrete solutions identified by an aggregation context in a solution model to a concrete solution aggregate, which itself is a new concrete solution. ■

Definition 6.3.7 (Solution Model Update Function) A *Solution Model Update Function σ* is a function that provides domain-specific logic to update a solution model after a successful aggregation of concrete solutions by replacing the aggregated concrete solutions with the new concrete solution aggregate $\dot{s}$.

Thereby, $\sigma : \mathfrak{SM} \times \mathfrak{AC} \times \mathfrak{S} \rightarrow \mathfrak{SM}$, $(\mathcal{G}_{sm}, ac, \dot{s}) \mapsto \dot{\mathcal{G}}_{sm}$ with

(i) $\hat{S} = \bigcup_{(ot,s) \in ac} \{s\}$ is the set of concrete solutions to be replaced by the aggregate

(ii) $\bar{E} = \left\{e \in \mathcal{G}_{sm} \mid \pi_1(e) \in \hat{S} \wedge \pi_2(e) \in \hat{S}\right\}$ is the set of edges between aggregated concrete solutions

(iii) $E_{\pi_1} = \left\{e \in \mathcal{G}_{sm} \mid \pi_1(e) \in \hat{S}\right\} \setminus \bar{E}$ is the set of edges where the start of the edges has to be replaced by the aggregate

(iv) $E_{\pi_2} = \left\{e \in \mathcal{G}_{sm} \mid \pi_2(e) \in \hat{S}\right\} \setminus \bar{E}$ is the set of edges where the target of the edges has to be replaced by the aggregate

(v) $\xi_{\pi_1} : E_{\pi_1} \times \{\dot{s}\} \longrightarrow \mathfrak{E}_{\mathfrak{S}\mathfrak{M}}, ((s_1, s_2, w), \dot{s}) \mapsto (\dot{s}, s_2, w)$, where $\mathfrak{E}_{\mathfrak{S}\mathfrak{M}}$ is the set of all edges of solution models

(vi) $\xi_{\pi_2} : E_{\pi_2} \times \{\dot{s}\} \longrightarrow \mathfrak{E}_{\mathfrak{S}\mathfrak{M}}, ((s_1, s_2, w), \dot{s}) \mapsto (s_1, \dot{s}, w)$

(vii) $\dot{\mathcal{G}}_{sm} = (\pi_1(\mathcal{G}_{sm}) \setminus \hat{S} \cup \{\dot{s}\}, \pi_2(\mathcal{G}_{sm}) \setminus \bar{E} \cup \bigcup_{e \in E_{\pi_1} \times \{\dot{s}\}} \left\{\xi_{\pi_1}(e)\right\} \cup \bigcup_{e \in E_{\pi_2} \times \{\dot{s}\}} \left\{\xi_{\pi_2}(e)\right\}, \pi_3(\mathcal{G}_{sm}), \pi_4(\mathcal{G}_{sm}), \pi_5(\mathcal{G}_{sm}), \pi_6(\mathcal{G}_{sm}))$

■

Thereby, the domain-specific logic of σ also decides what properties, and specifically which requirements and capabilities, the concrete solution aggregate provides. Whenever an aggregation program successfully performs the aggregation of an aggregation context, the aggregated concrete solutions in the solution model are replaced with the concrete solution aggregate. Thereby, edges between the aggregated concrete solutions are removed from the solution model and edges that connected the aggregated concrete solutions with other concrete solutions are connected to the concrete solution aggregate, such that a new solution model results. Please note that this processing of requirements, capabilities, edges, and properties is highly domain-specific and, thus, requires this update function in the aggregation program. The new solution model can then be used to apply further aggregation programs until all concrete solutions in it are aggregated as long as only a single remaining concrete solution is present in the solution model, then the processing finishes. This principle is depicted in Figure 6.5, where the aggregation program aggregates S_1 and S_3 to S_n and replaces the aggregated concrete solutions in the solution model with S_n, as depicted on the right.

The application of aggregation programs on solution models is discussed in more detail in the next chapter where it is shown how aggregation programs can be utilized to support and guide the stepwise aggregation of a solution model to an overall concrete solution.

6.4 Chapter Conclusion

In this chapter, the concept of concrete solution aggregation was introduced. To support the aggregation of concrete solutions it is necessary to formalize aggregations among them. Therefore, the concept of aggregation operators as a means to formalize the aggregation of concrete solutions were introduced. It was shown that they enable to create structures among sets of concrete solutions, which are called solution algebras. solution languages were introduced, which allows to structure the solution space of a pattern language by means of concrete solution aggregation descriptors, which provide the necessary documentation to manually aggregate concrete solutions in a human-readable way. However, in cases when the aggregation of concrete solutions has to be conducted over and over again, it is beneficial to add a mechanism, which allows to automate the aggregation. For this purpose, aggregation programs were introduced, which correspond to the automation of CSADs. Moreover, the concept of aggregation programs was developed based on established techniques for identifying the application of a certain kind of logic on graph-based models. For example, Breitenbücher [Bre16] uses detector fragments and subgraph isomorphisms to detect the applicability of management patterns on topology models, while Harzenetter et al. [HBF+18b; HBF+20] use this approach to refine patterns in deployment models by topology fragments implementing them. In this work, the general idea to identify subgraphs in a graph-based model based on isomorphisms with a detector graph along with analysis logic for refining the detector was translated and adapted to the domain of concrete solution aggregation. As a result of identifying aggregation contexts by means of subgraph isomorphism the discussion of algebraic properties of aggregation programs is relaxed, because the mapping of operation Targets to concrete solutions in a solution model makes commutativity obsolete.

Chapter 7

Semi-Automated Aggregation of Concrete Solutions

This chapter presents a concept for a toolchain supporting the EINSTEIN-Method. The introduced concepts of this work are set in the context of this method. Further, an algorithmic approach is presented utilizing aggregation programs to enable the stepwise aggregation of concrete solutions in a solution model, where the Implementer can gradually incorporate their requirements for the aggregations. It is conceptually shown, how the algorithmic approach can be implemented in a toolchain to semi-automate and guide the aggregation of concrete solutions. This chapter builds on the works of Beisel [Bei17], Falazi [Fal17a], and Krieger [Kri18c]. Although the aggregation of concrete solutions requires domain-specific logic the presented concept of a toolchain to aggregate concrete solutions is generic.

7.1 Concept of a Toolchain to support the EINSTEIN-Method

In the following, a concept for a toolchain is presented that supports the EINSTEIN-Method. It is intended to present the concept in a domain-independent way, thus, it shows a blueprint that can be instantiated for different domains by incorporating the specific requirements of each domain. For example, these could be specified in terms of the domain-specific rendering of patterns in design models or the rendering of concrete solutions in solution models which might differ between domains. Thus, the overall concept as depicted in Figure 7.1 can be refined towards the requirements of specific domains and, of course, requires also specific implementations of tools to meet and support the Designers and Implementers. Although some supporting tools, such as for example a general pattern repository, can be used in different domains, even such tooling can benefit from domain-specific adaptations with respect to the visualization and rendering of patterns according to the needs of a specific domain.

The first illustrated step in Figure 7.1 shows that, based on a use case at hand, a Designer has to be supported to use a pattern language to create a design model that encodes the conceptual solution for the problem at hand by the interplay of patterns. The Designer can be supported by a tool to create a design model. Thereby, starting with an empty design model, the tool has to provide the Designer with a palette of patterns organized in different pattern languages relevant for their problem domain. Since the relevant patterns can be spread among different pattern languages, the tool has to incorporate this accordingly by supporting the Designer to navigate among different pattern languages (cf. [LB21; WBB+20]). This is conceptually supported by the notion of pattern languages as pattern graphs, as formally introduced in this work in Definition 4.1.1. Therefore, to provide pattern languages in the respective form, the tool has to be able to load pattern graphs from a pattern repository. Further, an underlying pattern repository also has to support the formulation of pattern languages

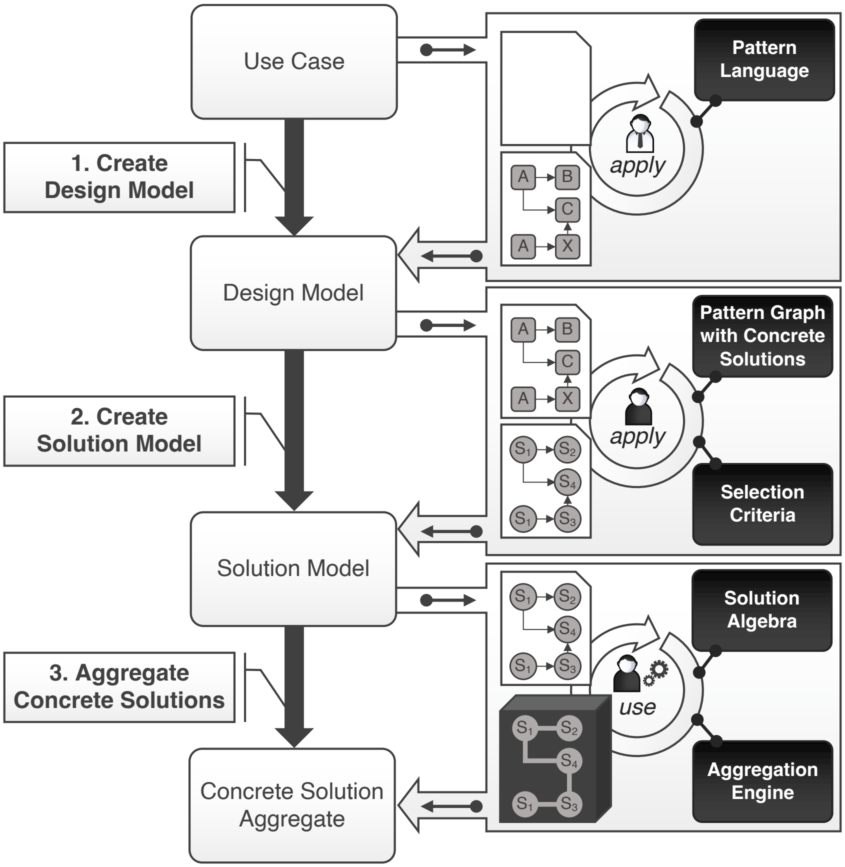


Figure 7.1: Interplay of the concepts to support the EINSTEIN-Method

as pattern graphs to realize proper inputs for the tool. The modeling tool must allow to drop patterns from the palette onto the modeling canvas and enable to create relations between the patterns reflecting the domain-specific semantics describing the interplay of the patterns to solve to use case at hand. Thereby, the patterns are interwoven by the Designer to design a conceptual solution for their problem at hand using design models as introduced in this work in Definition 4.4.1.

The resulting design model then has to be translated into a solution model by an Implementer as depicted by Step 2 in Figure 7.1. As introduced in Section 5.6, the means to reuse concrete solutions in the EINSTEIN-Method are solution models. Thus, the tool has to support the Implementer to translate the design model into a solution model stepwise. For this purpose, the tool needs to support selecting concrete solutions for the patterns of the design model. Thereby, it is required that an Implementer can search for relevant concrete solutions connected to the patterns of the design model. This can be supported by querying concrete solutions connected to the patterns of the design model utilizing selection criteria, which requires an implementation of the concept of pattern graphs with connected concrete solutions and selection criteria as specified in Definition 3.3.3 and Definition 5.5.1. As pattern languages and concrete solutions are typically managed in different repositories, this requires that patterns and concrete solutions are linked across repositories. Both, Beisel [Bei17] and Krieger [Kri18c] have validated the concept of linking patterns and concrete solutions across repositories. Further, the tool has to support the Implementer to reuse concrete solutions for the patterns of the design model and relate them according to the semantic needed to specify their interplay in the solution model, which is a new modeling canvas besides the design model.

Once the Implementer modeled a solution model they have to be supported to aggregate the concrete solutions of the solution model to an overall concrete solution aggregate as depicted by Step 3 in Figure 7.1. The tool can support the Implementer by an semi-automated approach. Solution algebras and concrete solution aggregation programs as introduced in Section 6.1 and Section 6.3 can be utilized to guide the Implementer to aggregate the concrete solutions of the solution model. For a given solution model, the tool can support the Implementer to aggregate the concrete solutions by providing a list of aggregation programs that can be applied to the solution model. Thereby, utilizing detector graphs of aggregation programs, the tool can identify for each available aggregation program on which subgraphs of the solution model it can operate and highlight the aggregatable subgraphs of the solution model. If the Implementer selects an aggregatable subgraph of an aggregation program, the aggregation itself is conducted by an *Aggregation Engine*. The aggregation engine is thereby capable of loading the artifacts of concrete solutions from a solution repository along with the aggregation programs. Further, it provides a runtime to execute the aggregatability analysis logic κ on a given aggregation context. Then, if the aggregatability analysis logic κ returns true, the aggregation engine can execute the aggregation operation function o and the solution model update function σ of the aggregation program to aggregate the concrete solutions of the solution model and provide an updated solution model as the result of the aggregation step. The resulting solution model can then be further aggregated by the Implementer by again checking which aggregation programs can be applied on the new solution model. Then, the Implementer can select again a subgraph of the solution model to be aggregated and the process can be repeated until the solution model is a single concrete solution or until no further aggregations are possible based on the available aggregation programs. Thus, the tool along with the aggregation engine implement especially the handling of aggregation programs specified by Definition 6.3.1. In the case that not all concrete solutions of a solution model could be aggregated based on aggregation programs, the Implementer can further check in a present solution language if there are manual instructions available to further aggregate the left concrete solutions of the solution model. This overall tool concept allows to semi-automate the reuse and aggregation of concrete solutions by guiding the Implementer to aggregate the concrete solutions stepwise by utilizing the aggregation programs of a solution algebra. While Step 1 and Step 2 described above are manual tasks conducted by Designers and Implementers, Step 3 also relies on automation aspects. In the next section, an algorithm is presented to the semi-automate the aggregation of concrete solutions as described in Step 3.

7.2 Algorithmic approach to implement the semi-automated Aggregation of Concrete Solutions

In the following, the assisted stepwise aggregation of concrete solutions in a solution model is described in detail and an algorithm is given that can be implemented in tools to provide semi-automated aggregation. The overall processing is depicted in Figure 7.2 where, both, a solution model to be aggregated and the available aggregation programs are the input. The depicted example in Figure 7.2 shows, how the stepwise aggregation of the solution model can be assisted. In the following, the algorithmic approach is described conceptually, whereas, the depicted example in Figure 7.2 can be used to comprehend the steps of the algorithm.

Algorithm 7.1 Aggregation Step

```
Require: Solution Model $SM$, Aggregation Programs $AP$
Ensure: Return a tuple $(SM, aggregationContext, ap, updatedSM)$ representing the aggregation step
 1: procedure PERFORMAGGREGATIONSTEP($SM, AP$)
 2:     $mappings \leftarrow []$
 3:     for all $ap \in AP$ do
 4:         $mappingsOfAp \leftarrow$ GETMAPPINGS($SM, ap$)
 5:         $mappings.append(mappingsOfAp)$
 6:     end for
 7:     $aggregation \leftarrow$ SELECTAGGREGATION($mappings$)
 8:     $aggregationContext \leftarrow aggregation[0]$
 9:     $o \leftarrow aggregation[1].o$
10:     $\sigma \leftarrow aggregation[1].\sigma$
11:     $csAggregate \leftarrow o(SM, aggregationContext)$
12:     $updatedSM \leftarrow \sigma(SM, aggregationContext, csAggregate)$
13:     return $(SM, aggregationContext, ap, updatedSM)$
14: end procedure
```

For a given solution model the procedure *performAggregationStep* specified as pseudo code in Algorithm 7.1 can be called. Firstly, *performAggregationStep* creates a list of potential aggregations supported by the aggregation programs on the given solution model (cf. lines 2 – 6 in Algorithm 7.1). Valid aggregations are identified and validated by calling the function *getMappings* specified in Algorithm 7.2 for each available aggregation program.

Algorithm 7.2 Find Subgraph Mappings

```
Require: Solution Model $SM$, Aggregation Program $ap$
Ensure: Return a list of all valid subgraph mappings of the aggregation
    program's detector graph in $SM$
 1: function getMappings($SM, ap$)
 2:     $mappings \leftarrow []$
 3:     $isos \leftarrow$ findSubgraphIsos($SM, ap.DG$)
 4:     for all $iso \in isos$ do
 5:         if $ap.\kappa(iso)$ then
 6:             $mapping \leftarrow (iso, ap)$
 7:             $mappings.append(mapping)$
 8:         end if
 9:     end for
10:     return $mappings$
11: end function
```

In turn, *getMappings* searches for all subgraph isomorphisms of the detector graph of an aggregation program in the solution model (cf. line 3 in Algorithm 7.2). The function *findSubgraphIsos* can be implemented utilizing the algorithm VF2 [CFSV01] configured with *node and edge matcher functions* that implement the matching constraints for concrete solutions and operation targets as well as the matching constraints for edges of the detector graph and the solution model as described in Section 6.3. As a result, this enables VF2 to search for node-colored and edge-weighted subgraph isomorphisms of the detector graph in the solution model. Since VF2 is a state of the art algorithm, the implementation of *findSubgraphIsos*

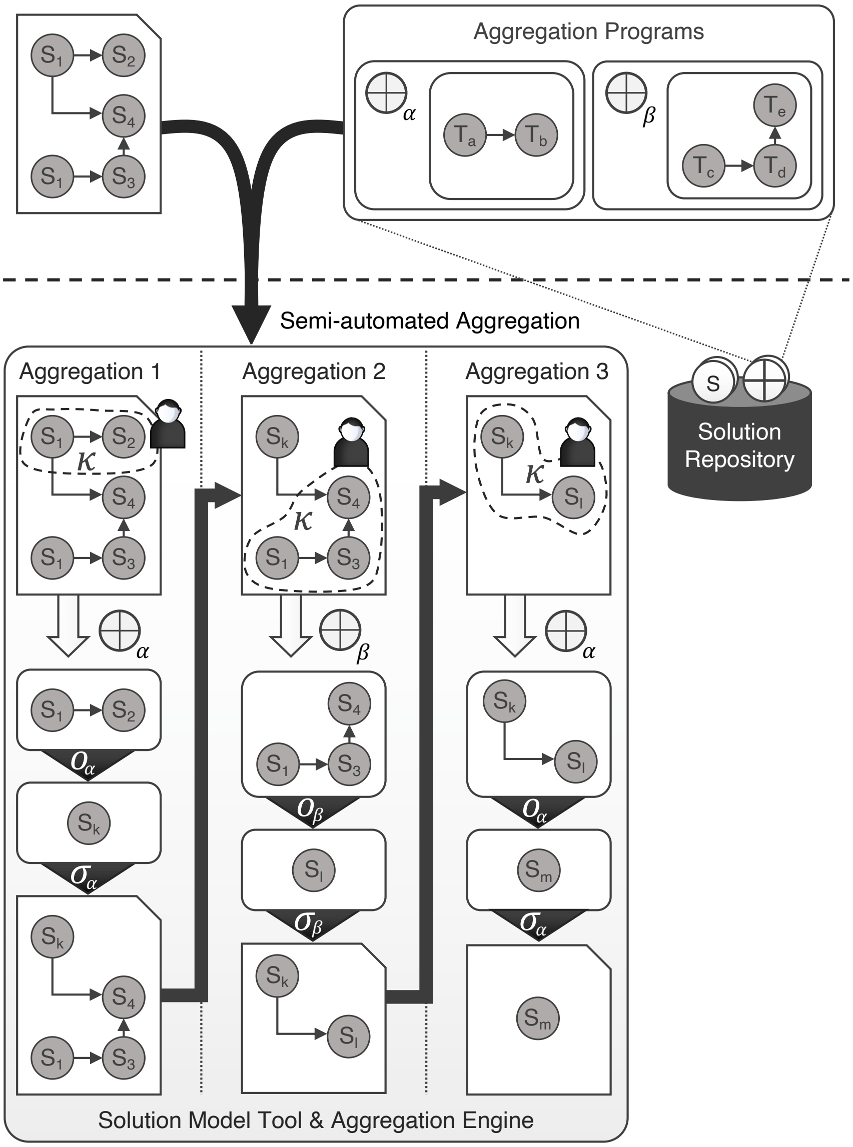


Figure 7.2: Assisted aggregation of concrete solutions

is not further detailed. Moreover, for each found subgraph isomorphism it is validated if the aggregation context identified by the subgraph isomorphism can be aggregated by the aggregation program. Thereby, the Context-specific Compatibility Analysis Function κ of the aggregation program is executed (cf. line 5 in Algorithm 7.2) for each identified subgraph isomorphism which ensures that only those subgraph isomorphisms for which κ returns *true* are kept. Finally, the valid subgraph isomorphisms are stored as a tuple together with the aggregation program in a list of mappings (cf. line 6 – 7 in Algorithm 7.2). Thus, each mapping represents an aggregation context along with the aggregation program that can operate on it.

The mappings of all available aggregation programs are merged to an overall list of potential aggregations. From this list, Implementers can then select one aggregation to be executed specified by the abstract function *selectAggregation* (cf. line 7 in Algorithm 7.1). The function *selectAggregation* represents an interaction of the Implementers with the supporting tool, whereby, they manually select the aggregation to be conducted. Thus, at this point, Implementers can bring in their domain knowledge and experience to determine the aggregation. Therefore, the function is not further specified in this work. For the given example in Figure 7.2 this results in the identification of the subgraph containing S_1 and S_2 as indicated by the dashed line for the fist aggregation illustrated in the left column.

After the Implementer selected the aggregation, it can be performed by calling the Aggregation Operation Function o and the Solution Model Update Function σ of the aggregation program on the solution model and the aggregation context (cf. line 8 – 12 in Algorithm 7.1). Thereby, line 8 represents, that the aggregation context is selected from the returned mapping tuple. Then, references to the functions o and σ are stored in variables in the lines 9 and 10, while they are called in the lines 11 and 12 to generate the concrete solution aggregate and to update the solution model. Finally, the algorithm returns a tuple representing the initial solution model, the aggregation context, the aggregation program, and the updated solution model of the aggregation step. As depicted in Figure 7.2 the resulting solution model can then be used as the input for the next aggregation step by the Implementer. For the new solution model, *performAggregationStep* can be called again to assist the Implementer to select the next aggregation and automatically aggregate the respective aggregation context. This can be iteratively conducted by the Implementer until no further aggregations are supported by the given aggregation programs.

7.3 Discussion

The presented approach tries to balance the tradeoff between automation and manual reuse of concrete solutions by incorporating user feedback and user decisions while automating the aggregation of concrete solutions in a user-controlled manner. However, a limitation of the presented approach is the lack of automating the aggregation of whole solution models and also the automated generation of solution models from design models. For the latter case, it is possible to automate the selection of concrete solutions for an entire design model by specifying a query that selects concrete solutions for all patterns of the design model as conceptually discussed by Falazi [Fal17a]. However, practical reasons, the query would have to represent a collection of constraints given by both, the Designer and the Implementer. This is because both roles typically bring in different perspectives and requirements, as discussed in Section 3.4. Further, such a query would have to assure that just one concrete solution is selected for each pattern in the design model, which is difficult for cases when multiple concrete solutions matching given constraints are available for a pattern. Otherwise, either all possible solution models would have to be generated for the set of selected concrete solutions or the query would have to be refined by the Implementer to select just one concrete solution for each pattern – which is close to the stepwise creation of a solution model as described in this work. Thereby, the focus would shift from iteratively selecting specific concrete solutions for a use case at hand, which is how systems are typically developed, to engineering a sophisticated query that selects concrete solutions for all patterns of a design model. Further, if the query is allowed to return multiple concrete solutions for a pattern, then the number of possibly generated solution models in the worst case scales exponentially with the number of patterns in the design model and the number of concrete solutions for each pattern, whereby, the Implementer finally needs to select the solution model, which matches best to solve the use case at hand. But overlooking a whole solution model is much more complex for an Implementer than dividing the whole complexity and selecting a concrete solution for each pattern individually.

Further, to automate the aggregation of overall solution models, a possible sequence of aggregations could be calculated by iteratively searching for sequences of applicable aggregation programs that collapse the solution model to a single concrete solution. This corresponds to a planning problem in the space of valid aggregation sequences. Depending on the algebraic properties of the available aggregation programs, thereby, a possibly large amount of aggregation steps and aggregation sequences could be required to be calculated. Even worse is, due to the fact that the aggregatability analysis function κ of the aggregation programs requires to investigate not only the solution model but also the artifacts of the concrete solutions, all concrete solution aggregates would also have to be generated to eventually decide if a sequence of aggregation programs completely aggregates a whole solution model. To cover the whole search space to assure that all possible aggregation sequences are investigated, also all possible orders of aggregation programs would have to be applied over and over again for each aggregation step, which again increases the complexity for the automation of overall solution models. This scales exponentially with the number of aggregation programs by $O(N!)$ with N being the set of aggregation programs. This approach quickly leads to a potentially large number of executions of the *getMappings* (cf. Algorithm 7.2) function. Thereby, *getMappings* itself searches for all subgraph isomorphisms of the detector graph of an aggregation program in a solution model. The search for subgraph isomorphisms is a problem with complexity $O(V! \times V)$ (cf. [CFSV01]) for V being the set of edges of the solution model.

As a result, based on the tradeoffs of this discussion, the presented approach focuses on the semi-automated and iterative aggregation of concrete solutions in a solution model, which also enables to integrate the EINSTEIN-Method more easily into existing procedure models (cf. Section 3.2). Thereby, an Implementer benefits from a feedback loop when selecting concrete solutions iteratively on the basis of the requirements and tradeoffs the Implementer is faced with when reusing concrete solutions to implement an use case at hand. For example, they could start with the selection of concrete solutions for the most important components as a basis, which thus act as an anchor for the further selection of concrete solutions. Then, they could complete the overall solution model by selecting concrete solutions for the remaining patterns in the design model.

7.4 Chapter Conclusion

In this chapter, an approach was presented that can be used to automate the aggregation of concrete solutions. It is based on the concepts of design models, pattern graphs with connected concrete solutions and solution algebras. To aggregate the concrete solutions of a solution model a generic concept of a toolchain was introduced as a blueprint that allows to instantiate the concepts of the EINSTEIN-Method for different domains. The aggregation of concrete solutions was further formulated as a semi-automated and iterative process that can be conducted by an Implementer. Thereby, an algorithmic approach was introduced and discussed that specifically enables an Implementer to iteratively adjust the aggregation of concrete solutions in a solution model by utilizing the aggregation programs of a solution algebra. The algorithm, enables a supporting tool to maintain an overall context for the aggregation of a solution model due to the fact that each aggregation step provides information consisting of the initial solution model, the aggregation context, the aggregation program, and the updated solution model of the aggregation step. By this means, the Implementer can be supported to always trace back the aggregation steps and the decisions made during the aggregation process, which is especially helpful if the Implementer wants to try out different implementations. Then, the Implementer can always go back to a previous aggregation to try out another aggregation. Further, the tool could be implemented to also undo mistakenly executed aggregation steps.

Chapter 8

Toolchain and Validation

To validate the technical feasibility of the presented approaches and concepts, a toolchain to support the EINSTEIN-Method as introduced as a blueprint in the previous chapter was implemented in different prototypes in the course of this work. The overall interplay of the different tools is shown in this chapter and illustrated as a coarse architecture. It is discussed how to implement aspects of the introduced blueprint. Finally, the concept of concrete solution aggregation is validated by discussing further validation scenarios in the domains *Cloud Application Design and Deployment*, *Cloud Application Management*, *Costumes in Films*, and by summarizing validation scenarios that were already published in other works.

8.1 Prototypes to support the EINSTEIN-Method

To support the EINSTEIN-Method, prototypes of tools were developed in the course of this work that implement the conceptual blueprint as introduced in the previous chapter. The different tools are depicted in Figure 8.1, which shows how they form an overall toolchain to support the EINSTEIN-Method. The tools are grouped into the categories *Supporting Tools for Pattern and Concrete Solution Authoring* as well as *Tools to support the EINSTEIN-Method*. The first group consists of a pattern repository and a solution repository, which both can be accessed via an API from corresponding user interfaces. On the one hand, the *Pattern Language Authoring UI* is designed and developed as an extendable web application based on Angular [Goo23]. It allows to author, maintain, and share pattern languages, thereby, providing the capability to implement specific renderings for the different pattern languages. This enables to adapt the look and feel of a pattern language to the requirements of the domain it is used in. Concepts and learnings from the previous works by Beisel [Bei17], Fehling et al. [FBFL15], Krieger [Kri18c], and Weigold [Wei19] influenced the development of the pattern language authoring UI. For example, to support the rendering of pattern languages according to the needs of the covered domains, a plugin-based rendering approach was implemented that enables to realized different look and feels for that match the needs of the users. In combination with the API and the pattern repository as the backend, which are implemented utilizing the Java framework Spring Boot [VMW23], it already evolved beyond being a prototype for this work. It is used on a daily basis and is under constant further development in the research project PlanQK [Pla23c] as part of the PlanQK platform [Pla23b] to author, manage, and share quantum computing patterns. They were initially started by Leymann [Ley19] and are constantly extended in the PlanQK project [Pla23a]. The source code for user interface, API, and backend is available as part of the *Pattern Atlas* project as open source soft-

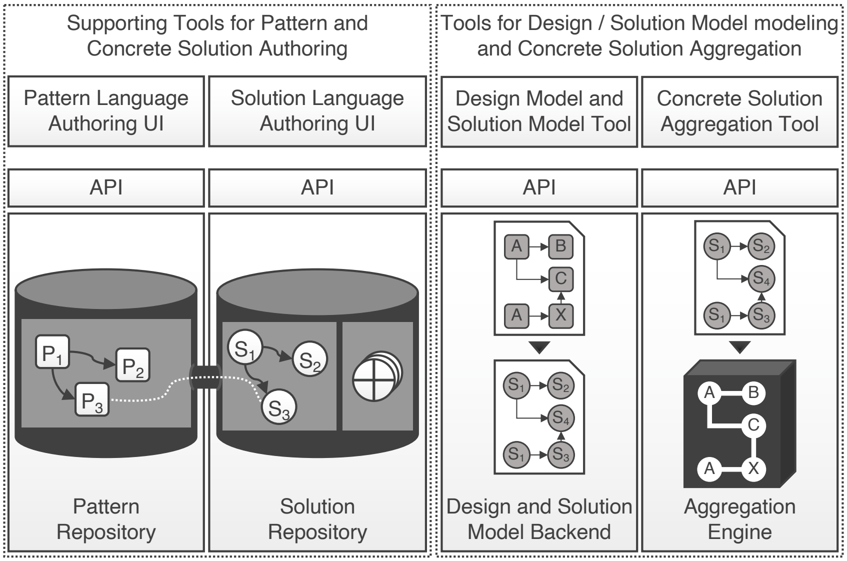

Supporting Tools for Pattern and Concrete Solution Authoring
Tools for Design / Solution Model modeling and Concrete Solution Aggregation
Pattern Language Authoring UI
Solution Language Authoring UI
Design Model and Solution Model Tool
Concrete Solution Aggregation Tool
API
API
API
API
P1
P2
P3
S1
S2
S3
A
B
C
X
S4
Pattern Repository
Solution Repository
Design and Solution Model Backend
Aggregation Engine

Figure 8.1: Architecture of a Toolchain to support the EINSTEIN-Method

ware [Dev23f][Dev23e]. The Pattern Atlas project bases on the metaphor of an atlas of patterns as detailed in [LB21]. A container-based setup to run and test the system is also available [Dev23d].

On the other hand, multiple solution repositories were developed providing validation of the concepts presented in this work. The first one to mention is a domain-specific solution repository for the domain of costumes in films. It is tailored to support the requirements of the domain and was developed and validated in the course of the MUSE project [Bar18] to capture costumes and their parts as concrete solutions. It enables to represent and digitalize tangible concrete solutions in the form of costumes in an IT system. It was also connected to a pattern repository to link costume patterns with concrete solutions in the solution repository [FBFL15]. It is available as open source software [Dev23c][Dev23a] and can be run and tested via a container-based setup [Dev23b].

Further, Krieger [Kri18c] implemented a combined pattern and solution repository based on semantic web technologies as a supervised work supporting this thesis. He shows, how patterns and solution languages can be represented in an ontology and how to query them using SPARQL [W3C23]. Thereby, concrete solution descriptors reference the implemented patterns, which allows to query for concrete solutions based on patterns. This concept is depicted in Figure 8.1 via the connection between the solution repository and the pattern repository. The source code is available as open source software [Kri18a][Kri18b]. Finally, the work by Beisel [Bei17] shows the implementation of a solution repository for implementations of the Cloud Computing Patterns by Fehling et al. [FLR+14].

The second group – *Tools to support the EINSTEIN-Method* – consists of the *Design Model and Solution Model Tool* and the *Concrete Solution Aggregation Tool*, which all are implemented as extensions of the *Pattern Atlas UI*, which is available as open source code [Dev23f]. Those tools support the aggregation of concrete solutions and were by Graf [Gra20]. Firstly, the *Design Model and Solution Model Tool* provides a modeling canvas to create design models based on a pattern language. Thereby, patterns can be selected from the pattern repository and added to the design model, such that instances of a pattern are created in the design model (cf. Section 4.3). They further can be linked to create relations between the patterns reflecting the intended design to solve a use case at hand by a Designer. Secondly, the extensions enable to filter and select concrete solutions from the solution repository that are related to the patterns in the design model. This way, a solution model can be created. The filtering of concrete solutions to replace patterns in a design model was further detailed by Falazi [Fal17a] by means of a grammar-based approach. Thereby, Falazi [Fal17a] shows how a lexer and a parser can be generated from a formal language that enables to specify queries on the set of concrete solutions in the solution repository considering selection criteria provided by a user. The source code is available as open source [Fal17b].

A prototypical aggregation engine was developed by Graf [Gra20] as a proof of concept to aggregate concrete solutions of the Cloud Computing Patterns by Fehling et al. [FLR+14] and the Messaging Patterns by Hohpe and Woolf [HW04]. This scenario is available as an exemplary input for Pattern Atlas and is detailed as a validation scenario below in Section 8.2.4.

8.2 Validation

To validate the presented approach, different scenarios leveraging the concepts presented in this work are discussed in the following. The validation scenarios show the feasibility of the presented approaches in the domains *Cloud Application Design and Deployment*, *Cloud Application Management*, *Costumes in Films*, and *Message-oriented Application Integration*. Each of the following subsections introduces a scenario as a basis to apply the concepts of this work. These discussed validation scenarios show the practical impact and feasibility of the application of concrete solutions and their aggregation in the mentioned domains. Section 8.2.4 further discusses a validation scenario that was also realized with the prototypical implementation of a design / solution model modeling tool and an aggregation engine.

8.2.1 Cloud Application Design and Deployment

The Cloud Computing Patterns by Fehling et al. [FLR+14] offer established solutions for the design of cloud applications. These patterns describe the conceptual components required at an abstract level but do not provide technical specifics regarding (i) the specific components to use or (ii) deployment methods. Considering for example the *User Interface Component* [FLR+14] pattern, it does not assist in selecting appropriate technologies, such as programming languages, web servers, or cloud infrastructures for deploying the entire component in a cloud.

In this context, the concept of concrete solutions and their aggregation is useful to obtain proven application structures that define specific technologies that work seamlessly together. The idea is to associate *Topology Models* describing technical components and dependencies among them with the abstract patterns by Fehling et al. [FLR+14]. When different patterns with topology models as concrete solutions need to be combined, these concrete solutions can provide a blueprint for the technical integration of the different components contained in the topology models. Ideally, when aggregating topology models the resulting models can be directly deployed using a deployment system.

In the following, the discussion delves into this concept further, primarily based on the OASIS standard *Topology and Orchestration Specification for Cloud Applications* (*TOSCA*) [OAS13]. TOSCA is an OASIS standard aimed at automating application deployment and management. This standard outlines a declarative deployment metamodel for describing topology models, which can be processed automatically by deployment systems to deploy and manage the modeled application. TOSCA allows for the precise definition of technical blueprints called topology models, specifying the technical components and their relationships within an application. This forms a solid foundation for capturing technical knowledge in the form of concrete solutions. Additionally, components specified in a TOSCA topology model can express requirements and capabilities similar to concrete solutions as represented in this work. E.g., a virtual machine component in a topology model may specify a requirement for deployment on a hypervisor, even if that hypervisor is not part of the topology model. In such cases, the deployment system must fulfill the open requirement and inject a hypervisor and associated management logic to provision virtual machine instances. These concepts play a crucial role in developing aggregation programs for aggregating topology models. To illustrate this concept, a scenario is considered where a user interface requires hosting. In this case, the *User Interface Component* pattern is combined with the *Elastic Infra-*

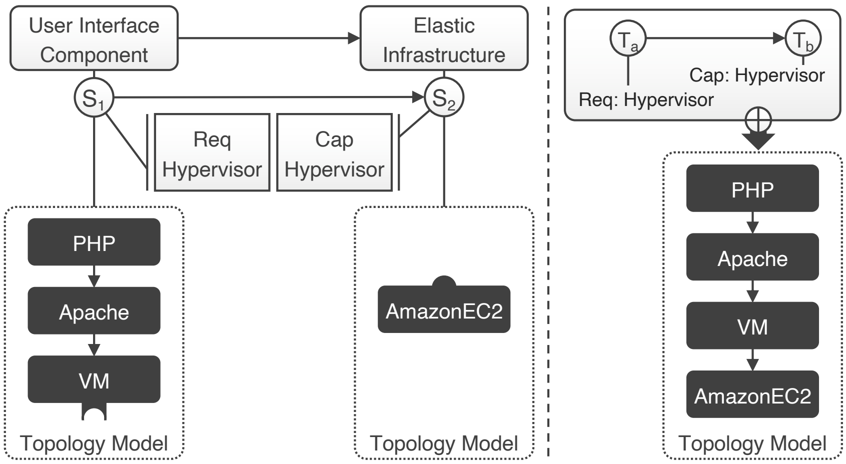


Figure 8.2: An aggregation operator aggregates topology models based on open requirements and provided capabilities into a complete topology model (adapted from [FBBL19]).

structure pattern [FLR+14]. The *Elastic Infrastructure* pattern describes a scalable deployment solution based on virtual machines in an elastic cloud.

Figure 8.2 illustrates two concrete solutions for these patterns. The *User Interface Component* pattern is linked with a concrete solution in the form of a TOSCA topology model. The topology consists of a user interface implemented as a PHP component hosted on an Apache Web Server, running on a Virtual Machine (VM). While this is a typical stack for hosting user interfaces, the virtual machine requires a hypervisor for execution. This can be represented as a requirement of the concrete solution. Thus, this topology model cannot be deployed directly but requires the injection of a hypervisor component.

In turn, the *Elastic Infrastructure* pattern defines a concrete solution in the form of another topology model containing an AmazonEC2 component, which represents Amazon Web Services' elastic infrastructure cloud offering. This concrete solution specifies its capability to provide a hypervisor. By matching the requirements and capabilities specified by the two concrete solutions and the detector graph of the aggregation program on the right in Figure 8.2, both topology models can be combined into a complete topology model by the aggregation program. The logic to validate if different topology models can be aggregated based on requirements and capabilities as well as the actual aggregation of the topology models can be automated as demonstrated by Saatkamp et al. [SBK+18] and Wild [Wil22].

8.2.2 Cloud Application Management

After deploying an application, the operations phase commences, during which various management processes are executed. One essential task is the scaling of application components [BBKL13]. Another prominent management use case involves migrating a component from an overloaded on-premise hosting environment to a scalable cloud provider. The primary advantage of such migration lies in the cloud's elasticity, where the number of component instances automatically scales based on actual workloads. Given the frequency of this scenario, the Cloud Computing Patterns detailed by Fehling et al. [FLR+14] encompass Application Management Patterns aimed at resolving such management challenges. Specifically, the *Forklift Migration* pattern addresses migration, while the *Elasticity Management Process* pattern guides the scaling of application instances based on workload. The combination of these two patterns offers a solution to the previously mentioned problem of overloaded on-premise components. However, it's crucial to automate these actions since manual executions are error-prone. Unfortunately, applying management patterns is challenging, as transforming an abstract pattern solution into an executable process demands extensive technical expertise in various technologies [FBB+14a]. Hence, having reusable, executable concrete solutions for these patterns presents a significant advantage compared to manual execution or writing complex, automatically executable scripts.

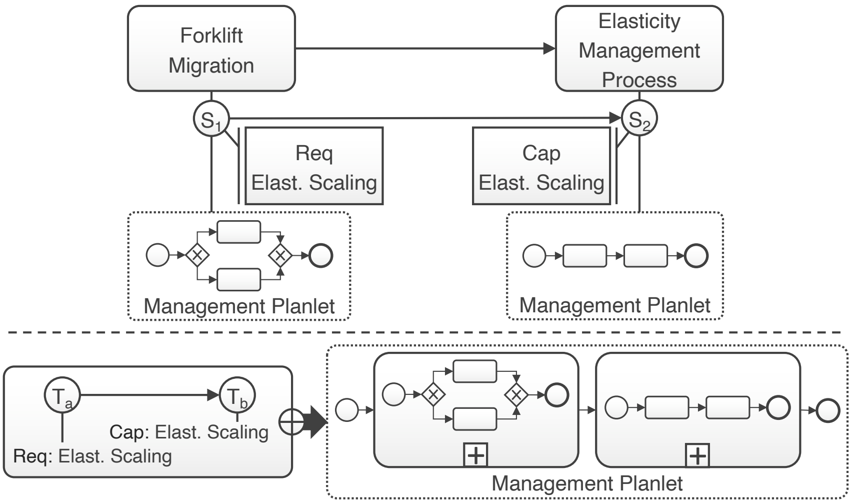


Figure 8.3: An aggregation program aggregates management planlets into subprocesses of an overall management planlet (adapted from [FBBL19]).

Falkenthal et al. [FBB+14a] demonstrated the use of management planlets as concrete solutions for management patterns. A management planlet is a generic building block for management tasks, structured as a Single-Entry-Single-Exit workflow (SESE). These management planlets implement specific management functions, such as component migration or scaling [BBKL13; Bre16]. Thereby, the concept of management planlets can be used on different granularity levels: for example, a management planlet could execute a simple task such as starting a virtual machine, but also a complex task such as migrating a component from one cloud provider to another as described by the higher-level management patterns by Fehling et al. [FLR+14]. Since management planlets follow the SESE-semantics, they can be easily aggregated into more complex workflows, which makes them appropriate for the concept of concrete solutions and their aggregation. Thus, management planlets can be used to implement concrete solutions for management patterns. As illustrated in Figure 8.3, management planlets can be linked as concrete solutions with the *Forklift Migration* pattern or the *Elasticity Management Process* pattern, serving as technical implementations for particular use cases [FBB+14a; FBB+14b]. For instance, migrating a PHP component from a local Apache Web Server to Amazon EC2 can be realized using a management planlet implementing the Forklift Migration pattern. Another management planlet can implement the *Elasticity Management Process* pattern to scale the PHP application on EC2. Thus, when migrating an application to an elastically scaling infrastructure, these two management planlet can be aggregated. This aggregation can be achieved by copying the corresponding workflows as subprocesses into a new workflow model and connecting them with a control flow. The resulting workflow again is a SESE-workflow, as depicted in Figure 8.3. The aggregation can be performed because the detector graph of the aggregation program matches with the requirements and capabilities of the concrete solutions. Further, in this case the aggregation program can also provide an aggregatability analysis function κ that checks if the workflows to be aggregated provide SESE-semantics, which is omitted in the figure for the sake of simplicity.

8.2.3 Costumes in Films

In the following, the concept of concrete solution aggregation is translated to the domain of costumes in films. Since, designing costumes for films is a creative process, it is not possible to fully automate the creation of costumes based on patterns and concrete solutions. However, the concept of concrete solutions and their aggregation can be applied to this domain conceptually to give a baseline to support the creative process of costume design. Besides that, costume patterns are still a relatively new concept and, therefore, not yet widely used in the domain of costume design. Thus, the following validation scenarios discuss the application of concrete solutions and their aggregation in the domain of costumes in films on a conceptual level by means of abstract aggregation operators.

Costumes in films play an important role in conveying specific character attributes to the audience through visual clues. They offer a means to communicate information about a character's role in a film, including their social status, their character traits, or even the mood of a character in a specific film scene. This communication is, for example, achieved by utilizing various materials of differing quality, such as using ragged cotton for portraying a penniless character or shiny silk for a well-dressed bank employee. Furthermore, moods and emotions of a character in different scenes can be conveyed through the proper choice of colors and tones in their attire. Additionally, typical stereotypes within a film genre, such as a sheriff or an outlaw in a western movie, can be represented through the clothing choices, which viewers anticipate due to their exposure to multimedia conventions. Consequently, these conventions can be systematically abstracted and authored into costume patterns, preserving the core principles of vestimentary communication [Bar18; BL15; SBLE12].

concrete solutions in the domain of film costumes can be captured in two ways. Firstly, they can be embodied in the actual physical costumes worn by characters during film production. Secondly, they can be systematically documented by providing detailed descriptions in a solution repository as outlined in Section 8.1 with the MUSE solution repository. In this approach, individual clothing items, referred to as *base elements* are cataloged along with their attributes, such as for example colors, material properties, or functions relevant to costume design. This documentation aids costume designers in exploring numerous concrete solutions related to costume patterns when crafting character wardrobes for film productions. A costume can be defined as a composition of base elements, with each base element representing a specific garment type, like a shirt, trousers, skirt, t-shirt, or jacket etc. Barzen [Bar18] gives a sophisticated ontology-based meta model for the domain of costumes in films, which is reduced and simplified for the discussions in this work as following.

Definition 8.2.1 (Set of Costumes, Base Elements, and Base Element Types) Let C be the set of costumes, $\mathcal{B}$ the set of base elements, and $T_{\mathcal{B}}$ the set of base element types. Each costume $c \in C$ is composed of a set

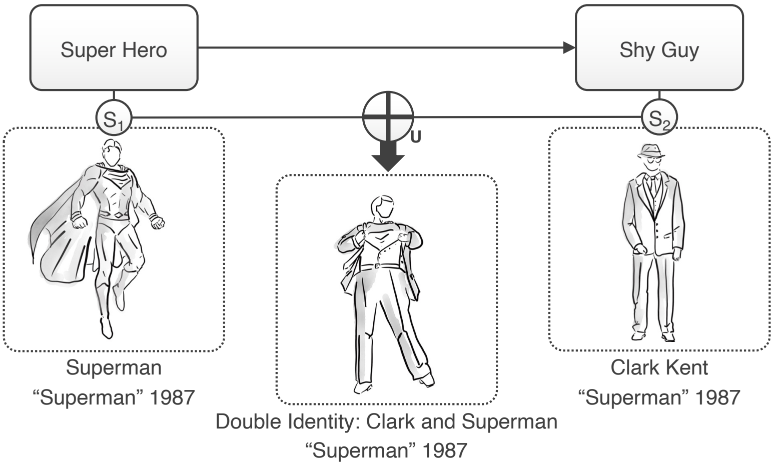


Figure 8.4: An aggregation operator forms the Double Identity Costume by combining the sets of base elements found in the two costumes of Superman and Clark Kent, both of which represent specific solutions for the *Super Hero* and *Shy Guy* patterns, respectively (adapted from [FBBL19]).

of base elements in $\mathcal{B}$, such that $\forall c \in C : c \in \wp(\mathcal{B})$. Additionally, the function ψ maps base elements in $\mathcal{B}$ to specific base element types in $T_{\mathcal{B}}$, s.t., $\psi : \mathcal{B} \longrightarrow T_{\mathcal{B}}, b_i \mapsto t_j$. ■

For example, a costume designer can navigate from the *Super Hero* pattern to Superman's costume, as depicted on the left in Figure 8.4, to systematically analyze the clothing of a superhero. Superheroes often conceal their true identities by assuming unassuming personas in their daily lives, illustrated in Figure 8.4 by Clark Kent's costume, which serves as a concrete solution for the *Shy Guy* pattern. While the creative process of costume design cannot be fully formalized and automated, costumes can be regarded as collections of base elements. In this framework, the combination of the Superman and Clark Kent costumes is essentially the union of their

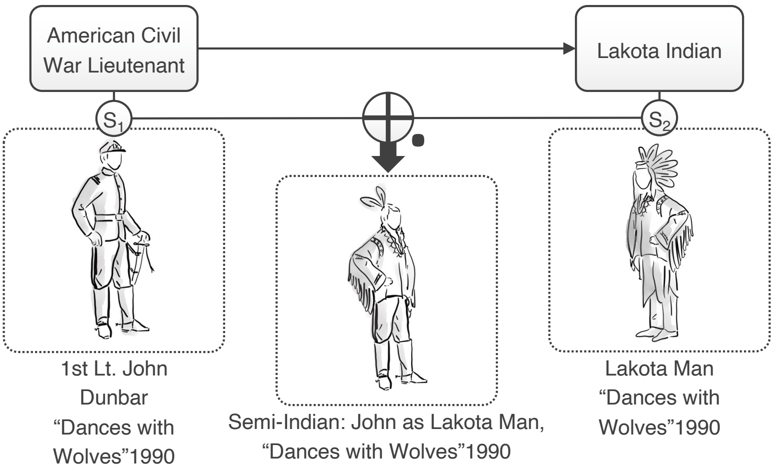

American Civil War Lieutenant
Lakota Indian
S1
S2
1st Lt. John Dunbar "Dances with Wolves"1990
Semi-Indian: John as Lakota Man, "Dances with Wolves"1990
Lakota Man "Dances with Wolves"1990

Figure 8.5: An aggregation operator merges the costume of First Lt. John Dunbar with that of the Lakota Man, resulting in an ensemble that represents a character with a Semi-Indian identity (adapted from [FBBL19]).

respective base element sets. The arrangement of these base elements, such as wearing the superhero costume under the shy guy attire, is based on the creativity of the costume designer and cannot be formalized exhaustively. Therefore, the properties of this aggregation operator are assumed analogous to the basic set union operation. This can be defined as follows.

Definition 8.2.2 (Aggregation Operator $\oplus_\cup$) The aggregation operator $\oplus_\cup : C \times C \to C$, $(c_1, c_2) \mapsto c_1 \cup c_2$, aggregates two costumes by uniting their sets of base elements. ■

The operator is commutative and associative, satisfying $c_1 \oplus_\cup c_2 = c_2 \oplus_\cup c_1$ and $(c_1 \oplus_\cup c_2) \oplus_\cup c_3 = c_1 \oplus_\cup (c_2 \oplus_\cup c_3)$.

In other scenarios, the combination of costumes can occur differently. As shown in Figure 8.5, costumes can also be merged to highlight the transformation of a character from one stereotype to another. For example, First Lt. John Dunbar, a soldier from the American Civil War era represented by the costume on the left in Figure 8.5, undergoes a character and attitude transformation during the movie *Dances with Wolves* (1990). He befriends members of the Lakota tribe and expresses his connection with them by adopting elements of their clothing. Specific costumes worn by Lakota men can be regarded as concrete solutions of the *Lakota Indian* pattern, depicting this common stereotype. Consequently, the aggregation of both costumes represents a selection from their base element sets. This aggregation operator requires ensuring that if base elements of the same type exist in both costumes, only one instance of each type is included in the aggregated costume. This operator can be defined as follows:

Definition 8.2.3 (Aggregation Operator $\oplus_\bullet$) The aggregation operator $\oplus_\bullet : C \times C \to C$, $(c_1, c_2) \mapsto c_1 \cup c_2$ with $\forall b_j \neq b_k \in c_1 \cup c_2 : (\psi(b_j) \neq \psi(b_k)) \vee (b_j \in c_1 \wedge b_k \in c_1) \vee (b_j \in c_2 \wedge b_k \in c_2)$ ensures the inclusion of distinct types of base elements in the aggregated costume, unless a base element type exists in both original costumes.

Based on these aggregation operators, a solution algebra can be defined as $(\{\{C\}, \{\oplus_\cup, \oplus_\bullet\}\})$. While $\oplus_\cup$ and $\oplus_\bullet$ are not distributive, they can be applied sequentially. Costume designers should carefully consider the application order that aligns with their requirements.

8.2.4 Validation in other Works

The former subsections discussed validation scenarios that show the application of the introduced concepts conceptually. To validate the prototypical implementation of a modeling tool for design models and solutions models as well as an aggregation engine, Graf [Gra20] shows the application of

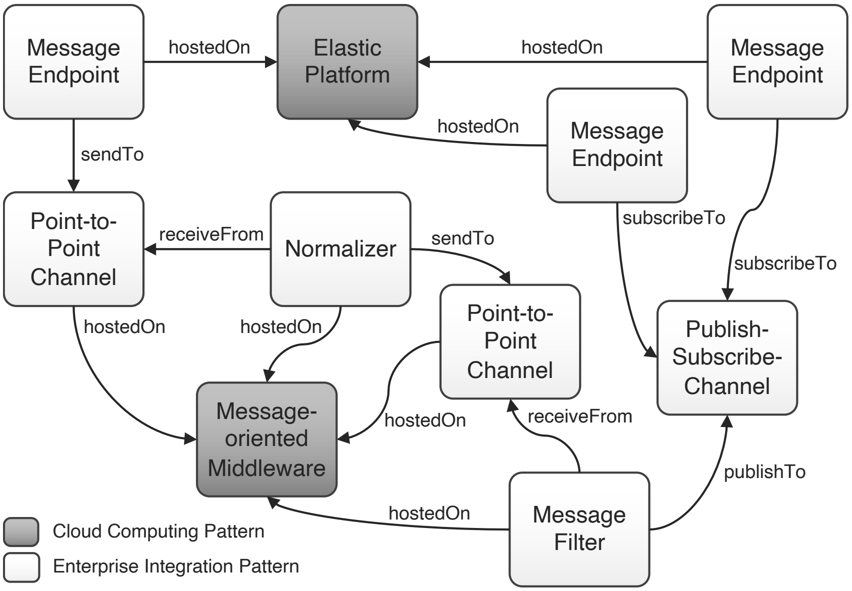


Figure 8.6: Aggregation Scenario with Cloud Computing Patterns and Enterprise Integration Patterns (adapted from [Gra20, p.54]).

the presented concepts in the domain of messaging-based application integration. He shows how the concepts can be applied to aggregate concrete solutions of the Cloud Computing Patterns by Fehling et al. [FLR+14] and the Messaging Patterns by Hohpe and Woolf [HW04]. Thereby, he designed and implemented a comprehensive validation scenario [Gra20, p.54] that is depicted in Figure 8.6.

Figure 8.6 shows the overall design model of the scenario. The integration of multiple applications is represented by patterns from the pattern language by Hohpe and Woolf [HW04]. Thereby, messages of the upper left *Messaging Endpoint* are passed through *Point-to-Point Channel* to a *Normalizer*, which transforms the messages into a common format. Subsequently, the scenario shows that the messages are passed through another *Point-to-Point Channel* to a *Message Filter*, which filters all messages according to a specific business logic. Messages that pass the *Message Filter* are published to a *Publish-Subscribe Channel*, from which the *Messaging Endpoints* on the upper right consume new messages. All these messaging components are hosted on a *Message-oriented Middleware*, while the respective *Messaging Endpoints* are hosted on an *Elastic Platform*, whereby both, *Message-oriented Middleware* and *Elastic Platform* are patterns from the language by Fehling et al. [FLR+14]. This overall interplay is specified by the respective domain-specific semantics on the edges as depicted in the design model in Figure 8.6. Graf [Gra20] shows how concrete solutions for these messaging patterns can be implemented as string templates to generate configuration files for the message-oriented middleware ActiveMQ [Apa23] or Java programs implementing the *Message Endpoints*, respectively. Further, the concrete solutions for the Cloud Computing Patterns are implemented as CloudFormation snippets. He further, shows how aggregation programs can be implemented, which are capable of aggregating the concrete solutions of the validation scenario. Finally, he describes how the prototypically implemented aggregation engine can execute the aggregation programs to aggregate concrete solutions implementing the patterns of the design model. Thus, he validates the feasibility of the presented concepts in the domain of messaging-based application integration.

8.3 Chapter Conclusion

In this chapter, it was argued that the conceptually introduced blueprint for a toolchain to support the EINSTEIN-Method can be implemented. The different components and their interplay show the feasibility of the concepts presented in this work. The toolchain is a prototypical starting point to implement the concepts and the EINSTEIN-Method in further domains beyond those discussed in this work. The applicability of the presented concepts was further validated by different validation scenarios. Beyond those presented in this section, further validation scenarios from the domains of *User Interaction Design* and *Object-oriented Software Engineering* are detailed in the works by Falkenthal et al. [FBB+14a; FBB+14b] that were elaborated in the course of this thesis as well.

Chapter 9

Conclusion and Outlook

The present work introduces an approach for the utilization of concrete solutions and their aggregation. The central focus lies on the reuse of concrete solutions, which correspond to implementations of patterns. This is facilitated by the EINSTEIN-Method, which is formulated as a domain-agnostic framework for applying the concepts presented in this work. To enable domain-independent modeling of conceptual solutions, design models have been introduced. These models serve as a generalization of pattern graphs, as they can represent multiple instances of a pattern to model complex solutions where a pattern must be used more than once. To refine patterns in a design model using patterns at different abstraction levels, the concept of pattern refinement has been discussed. For the reuse of pattern implementations, concrete solutions have been introduced as the key elements of the presented approach. It was shown how concrete solution descriptors can be defined and implemented as structures that support the description of concrete solutions. They are a domain-independent concept but can be refined to capture domain-specific properties of concrete solutions to ensure the general applicability of this approach. Selection criteria were introduced as special extensions of the properties of concrete solution descriptors, thereby, enabling users to search for concrete solutions selectively.

To support the manual aggregation and organization of concrete solutions, the concept of pattern languages has been extended to the level of concrete solutions by means of solution languages. These languages, along with concrete solution aggregation descriptors, enable the creation of a knowledge base that assists in organizing and reusing concrete solutions in combination. As a fundamental basis to formulate the aggregation of concrete solutions, the concept of aggregation operators has been introduced. Aggregation operators operate on the structures of concrete solution descriptors, specifically the requirements and capabilities, to ensure the compatibility of concrete solutions. They create a structure on the set of concrete solutions, dividing concrete solutions into cohesive subsets on which the aggregation operators can be applied. Such structures were introduced as solution algebras.

To implement the patterns in a design models by reusing concrete solutions, it has been demonstrated how design models can be iteratively transformed into solution models, which represent a translation of design models into graphs of concrete solutions. It was shown that solution algebras can be applied to these graphs to check whether the selected concrete solutions can be aggregated into an overarching solution. Additionally, solution aggregation programs were introduced as the implementations of aggregation operators that enable the semi-automated and assisted aggregation of concrete solutions in a solution model. To support Designers and Implementers to establish the EINSTEIN-Method in their work, a conceptual tool environment has been presented that can be grasped as a blueprint for a toolchain supporting the EINSTEIN-Method. Thereby, an algorithm for implementing the assisted aggregation of concrete solutions by an Implementer utilizing aggregation programs was introduced.

To validate the presented concepts, an architecture for a tool environment implementing the conceptual tool environment has been presented. Thereby, the feasibility of the presented concepts has been demonstrated through prototypes of tools for pattern and concrete solution authoring as well as the aggregation of concrete solutions. Finally, various validation scenarios across different domains were discussed to illustrate the applicability of the approach. Besides the many examples in the different chapters to illustrate the presented concepts, these validation scenarios underpin the applicability of the approach in different domains.

As a conclusion, this work presents a domain-independent approach for the utilization of concrete solutions and their aggregation. However, the applicability of the approach depends on its implementation in tools tailored to specific domains, as each domain has unique requirements for handling patterns and representing concrete solutions. Such tools often need to be integrated into existing toolchains to be effectively used. Therefore, this work can be grasped as a general framework for the utilization of concrete solutions and their aggregation that can be incorporated into tools and procedure models across various domains.

Several interesting directions for future work can be identified as following. Since this work bases on structural patterns, it would be interesting to also incorporate behavioral patterns into the approach, such as presented by Harzenetter et al. [HBF+20] for pattern-based deployment models. Then design models could be extended to not just allow to specify the structure of a solution but also to add patterns that adapt or configure other patterns. To do so, further semantics on edges have to be introduced in design models, which allow to distinguish between aggregating patterns and adapting patterns. Further, the concept of aggregation operators then also needs to be adjusted to support the adaption of concrete solutions, accordingly. Since the applicability of the approach to the domain of costumes in films was discussed by means of abstract aggregation operators it would be interesting to investigate their implementation and application in real film productions. The general actions required for aggregating costumes, for example, could be implemented as Human Tasks in workflows. This idea, combined with the fact that many costumes from films are already digitized in the MUSE solution repository could enable to extend the approach to also support the automated aggregation of costumes. Based on this, it can be possible to implement aggregation operators in this domain that enable to materialize the aggregation of costumes into workflows. Further, it would be interesting to investigate the automated aggregation of overall solution models. However, to find a suitable approach for this, it is necessary to investigate the aggregation of overall solution models in specific domains by defining solution models and aggregation operators representative for the domain. In this context, it would also have to be investigated how much semantics must be contained in the solution models to enable their automated aggregation. A good balance must be found between the representation of details in solution models and the analysis of the artifacts of concrete solutions by the aggregation programs. The difficulty is that if too much semantics has to be present in the solution models, mapping this to a full representation of implementations in the solution model easily renders the approach practically unusable. Finally, although some of the tools developed in the coarse of this work are used on a daily basis beyond this thesis, they are still in an early stage of development. So, especially to investigate the applicability and usability of the presented approaches in further domains, it would be beneficial to increase the maturity of these tools and to integrate them into further existing toolchains.

Bibliography

[AEK+07] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, A. A. Totok. “Pattern Based SOA Deployment”. In: *Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*. Springer, Sept. 2007, pp. 1–12 (cit. on p. 99).

[AFL12] V. Andrikopoulos, C. Fehling, F. Leymann. “Designing for CAP - The Effect of Design Decisions on the CAP Properties of Cloud-native Applications”. In: *Proceedings of the 2nd International Conference on Cloud Computing and Service Science (CLOSER 2012)*. SciTePress, Apr. 2012, pp. 365–374 (cit. on p. 99).

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977 (cit. on pp. 13–15, 35–38, 58, 60, 66, 72, 79–81, 83, 85, 87, 88, 99, 112, 116).

[Alb20] J. Albano. *Design Patterns in the Spring Framework*. 2020. URL: https://www.baeldung.com/spring-framework-design-patterns (cit. on p. 46).

[Ale64] C. Alexander. *Notes on the Synthesis of Form*. Oxford University Press, 1964 (cit. on pp. 34, 36, 41).

[Ale79] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979 (cit. on pp. 13–16, 36–38, 40, 41, 59, 60, 83, 85, 87, 98, 116).

[Ama13] Amazon Webservice. *AWS Cloud Design Patterns*. 2013. URL: http://en.clouddesignpattern.org/index.php/Main_Page (cit. on pp. 18, 49, 51, 68, 92–95).

[Ama23a] Amazon. *Amazon Machine Images*. 2023. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html (cit. on pp. 132, 138).

[Ama23b] Amazon. *AWS CloudFormation*. 2023. URL: https://aws.amazon.com/de/cloudformation/ (cit. on pp. 131, 137).

[Ant96] D. L. G. Anthony. “Patterns for Classroom Education”. In: *Pattern Languages of Program Design 2*. Addison-Wesley, 1996, pp. 391–406 (cit. on p. 14).

[Apa23] Apache Software Foundation. *Apache ActiveMQ*. 2023. URL: https://activemq.apache.org (cit. on p. 178).

[App97] B. Appleton. “Patterns and Software: Essential Concepts and Terminology”. In: *Object Magazine Online* 3.5 (1997) (cit. on pp. 13, 18).

[AZ05] P. Avgeriou, U. Zdun. “Architectural Patterns Revisited – A Pattern Language”. In: *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. UVK - Universitaetsverlag Konstanz, July 2005 (cit. on pp. 59, 60, 99).

[Bar18] J. Barzen. “Wenn Kostüme sprechen - Musterforschung in den Digital Humanities am Beispiel vestimentärer Kommunikation im Film (in English: When Costumes Speak - Pattern Research in the Digital Humanities Using the Example of Vestimentary Communication in Films)”. Dissertation. University of Cologne, 2018, p. 280 (cit. on pp. 14, 71, 76, 96, 106, 109, 111, 165, 173).

[BBB+23] F. Bühler, J. Barzen, M. Beisel, D. Georg, F. Leymann, K. Wild. “Patterns for Quantum Software Development”. In: *Proceedings of the 15th International Conference on Pervasive Patterns and Applications (PATTERNS 2023).* Xpert Publishing Services (XPS), 2023, pp. 30–39 (cit. on pp. 14, 38).

[BBE+17] J. Barzen, U. Breitenbücher, L. Eusterbrock, M. Falkenthal, F. Hentschel, F. Leymann. “The vision for MUSE4Music – Applying the MUSE method in musicology”. In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 323–328 (cit. on pp. 23, 30, 76, 106, 112).

[BBK+13a] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. “Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies”. In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013).* Springer, Sept. 2013, pp. 130–148 (cit. on p. 118).

[BBK+13b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, M. Wieland. “Policy-Aware Provisioning of Cloud Applications”. In: *Proceedings of the Seventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2013).* Xpert Publishing Services, Aug. 2013, pp. 86–95 (cit. on p. 136).

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. “Pattern-based Runtime Management of Composite Cloud Applications”. In: *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013).* SciTePress, May 2013, pp. 475–482 (cit. on pp. 44, 99, 118, 170, 171).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. “Automating Cloud Application Management Using Management Idioms”. In: *Proceedings of the Sixth International Confer-*

ences on Pervasive Patterns and Applications (PATTERNS 2014). Xpert Publishing Services, May 2014, pp. 60–69 (cit. on p. 44).

[BBL+22] M. Beisel, J. Barzen, F. Leymann, F. Truger, B. Weder, V. Yussupov. “Patterns for Quantum Error Handling”. In: *Proceedings of the 14th International Conference on Pervasive Patterns and Applications (PATTERNS 2022)*. Xpert Publishing Services, 2022, pp. 22–30 (cit. on pp. 14, 38).

[BBL22] J. Barzen, F. Bühler, F. Leymann. “Der MUSE Datensatz (in English: The MUSE Dataset)”. In: *ZfdG - Zeitschrift für digitale Geisteswissenschaften. Fabrikation von Erkenntnis: Experimente in den Digital Humanities* (Sept. 2022) (cit. on p. 111).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Apr. 2003 (cit. on p. 56).

[BDH05] A. Barros, M. Dumas, A. H. M. ter Hofstede. “Service Interaction Patterns”. In: *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*. Springer, Sept. 2005, pp. 302–318 (cit. on p. 99).

[BDR21] O. Bibartiu, F. Dürr, K. Rothermel. “Clams: A Cloud Application Modeling Solution”. In: *2021 IEEE International Conference on Services Computing*. 2021, pp. 1–10 (cit. on p. 44).

[Bec07] K. Beck. *Implementation Patterns*. 1st ed. Addison-Wesley, Nov. 2007 (cit. on p. 99).

[Bec96] K. Beck. *SmallTalk Best Practice Patterns*. Prentice Hall, Oct. 1996 (cit. on p. 99).

[Bei17] M. Beisel. “Concept and Implementation of a Solution Language and a Solution Repository for Cloud Computing Patterns”. Bachelor’s Thesis. University of Stuttgart, 2017, p. 45 (cit. on pp. 128, 139, 149, 152, 164, 166).

[BFYV96] F. Budinsky, M. Finnie, P. Yu, J. Vlissides. "Automatic Code Generation from Design Patterns". In: *IBM Systems Journal* 35.2 (1996), pp. 151–171 (cit. on pp. 41, 42).

[BGd10] P. Bottoni, E. Guerra, J. de Lara. "A language-independent and formal approach to pattern-based modelling with support for composition and analysis". In: *Information and Software Technology* 52.8 (2010), pp. 821–844 (cit. on p. 39).

[BHS07a] F. Buschmann, K. Henney, D. C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Vol. 4. Wiley & Sons, 2007, p. 636 (cit. on p. 41).

[BHS07b] F. Buschmann, K. Henney, D. C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. Vol. 5. Wiley & Sons, 2007, p. 490 (cit. on p. 41).

[Bib23] O. Bibartiu. "Architecture-based availability prediction and service recommendation for cloud computing". Dissertation. University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, 2023 (cit. on p. 44).

[BL15] J. Barzen, F. Leymann. "Costume Languages as Pattern Languages". In: *Pursuit of Pattern Languages for Societal Change (PURPLSOC) - The Workshop 2014: Designing Lively Scenarios With the Pattern Approach of Christopher Alexander*. 2015, pp. 88–117 (cit. on pp. 76, 96, 97, 173).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Oct. 1996 (cit. on pp. 14, 37, 41, 59, 99).

[Bos96] J. Bosch. "Design Patterns as Language Constructs". In: *Journal of Object-Oriented Programming* 11.2 (1996) (cit. on pp. 40–42).

[Bos98a] J. Bosch. “Design Patterns & Frameworks: On the Issue of Language Support”. In: *Object-Oriented Technologys*. Springer, 1998, pp. 133–136 (cit. on p. 47).

[Bos98b] J. Bosch. “Specifying Frameworks and Design Patterns as Architectural Fragments”. In: *Proceedings of Technology of Object-Oriented Languages*. IEEE, 1998, pp. 268–277 (cit. on p. 47).

[Bre16] U. Breitenbücher. “Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements (in English: A Pattern-based Method for Application Management Automation)”. Dissertation. University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, 2016 (cit. on pp. 44, 99, 118, 134, 147, 171).

[Bul02] A. Bulka. “Design Pattern Automation”. In: *Proceedings of the 3rd Asia-Pacific Conference on Pattern Languages of Programs*. Vol. 13. 2002 (cit. on p. 47).

[BZ11] I. Bayley, H. Zhu. “A Formal Language for the Expression of Pattern Compositions”. In: *International Journal On Advances in Software* 4.3&4 (2011), pp. 342–353 (cit. on p. 39).

[CFSV01] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. “An Improved Algorithm for Matching Large Graphs”. In: *Proceedings of the 3rd International Workshop on Graph-Based Representations in Pattern Recognition (IAPR-TC15)*. Jan. 2001, pp. 149–159 (cit. on pp. 155, 159).

[CKK11] P. Clements, R. Kazman, M. Klein. *Evaluating software architectures : methods and case studies*. Addison-Wesley, 2011, p. 323 (cit. on p. 56).

[CM13] W. Cunningham, M. Mehaffy. “Wiki as Pattern Language”. In: *Proceedings of the 20th Conference on Pattern Languages of Programs*. ACM, 2013, Artivel No. 32 (cit. on p. 50).

[Cop96] J. O. Coplien. *Software Patterns*. SIGS Books & Multimedia, 1996 (cit. on pp. 37, 40, 108).

[Cun] Cunningham, Ward. *Portland Pattern Repository*. URL: http://c2.com/ppr/ (cit. on p. 51).

[Dev23a] Developers of the MUSE Server. *Project of the MUSE Server*. 2023. URL: https://github.com/Muster-Suchen-und-Erkennen/muse-server (cit. on p. 165).

[Dev23b] Developers of the MUSE Tool. *Project of the MUSE Docker Compose Setup*. 2023. URL: https://github.com/Muster-Suchen-und-Erkennen/muse-docker (cit. on p. 165).

[Dev23c] Developers of the MUSE UI. *Project of the MUSE UI*. 2023. URL: https://github.com/Muster-Suchen-und-Erkennen/muse-ui (cit. on p. 165).

[Dev23d] Developers of the Pattern Atlas. *Docker Compose Setup to run the Pattern Atlas*. 2023. URL: https://github.com/PatternAtlas/pattern-atlas-docker (cit. on p. 165).

[Dev23e] Developers of the Pattern Atlas. *Pattern Atlas API and Backend*. 2023. URL: https://github.com/PatternAtlas/pattern-atlas-api (cit. on p. 165).

[Dev23f] Developers of the Pattern Atlas. *Pattern Atlas UI*. 2023. URL: https://github.com/PatternAtlas/pattern-atlas-ui (cit. on pp. 165, 166).

[DF06] A. Dearden, J. Finlay. “Pattern languages in HCI : a critical review”. In: *Human Computer Interaction* 21.1 (2006), pp. 49–102 (cit. on p. 36).

[DF23] J. T. Duarte Maia, F. Figueiredo Correia. “Service Mesh Patterns”. In: *Proceedings of the 27th European Conference on Pattern Languages of Programs*. ACM, 2023 (cit. on p. 38).

[DY03] J. Dong, S. Yang. "Visualizing design patterns with a UML profile". In: IEEE Computer Society, 2003, pp. 123–136 (cit. on p. 103).

[Ebe14] H. Eberle. "Prozessbausteine (in English: Process Building Blocks)". Dissertation. University of Stuttgart, 2014 (cit. on p. 123).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications". In: *Proceedings of the Ninth International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services, Feb. 2017, pp. 22–27 (cit. on pp. 29, 99).

[Fal17a] G. Falazi. "A Concept for Describing Concrete Solutions to Support their Automated Selection from Patterns". Master's Thesis. University of Stuttgart, 2017, p. 103 (cit. on pp. 123, 149, 158, 166).

[Fal17b] G. Falazi. *Solution Selection Prototype*. 2017. URL: https://github.com/PatternAtlas/solution-selection (cit. on p. 166).

[Fau16] J. Fauser. "Pattern Refinement in the domain of Cloud Computing". Master's Thesis. University of Applied Sciences Reutlingen, 2016, p. 141 (cit. on p. 93).

[FBB+14a] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. "Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains". In: *International Journal On Advances in Software* 7.3&4 (Dec. 2014). IARIA, pp. 710–726 (cit. on pp. 16, 23, 24, 28, 41, 105, 128, 170–172, 179).

[FBB+14b] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. "From Pattern Languages to Solution Implementations". In: *Proceedings of the Sixth International Con-*

ferences on Pervasive Patterns and Applications. Xpert Publishing Services, May 2014, pp. 12–21 (cit. on pp. 16, 23, 24, 28, 41, 105, 128, 172, 179).

[FBB+15] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, H. Schulze. “Leveraging Pattern Application via Pattern Refinement”. In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change*. epubli, June 2015, pp. 38–61 (cit. on pp. 16, 22, 28, 69, 79, 80, 89, 95–97).

[FBB+17] M. Falkenthal, J. Barzen, U. Breitenbücher, S. Brügmann, D. Joos, F. Leymann, M. Wurster. “Pattern research in the digital humanities: how data mining techniques support the identification of costume patterns”. In: *SICS Software-Intensive Cyber-Physical Systems* 32.3-4 (2017). Springer, pp. 311–321 (cit. on pp. 23, 28, 96, 106).

[FBBL14] C. Fehling, J. Barzen, U. Breitenbücher, F. Leymann. “A Process for Pattern Identification, Authoring, and Application”. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP 2014)*. ACM, Jan. 2014 (cit. on pp. 40, 88).

[FBBL17] M. Falkenthal, J. Barzen, U. Breitenbücher, F. Leymann. “Solution Languages : Easing Pattern Composition in Different Domains”. In: *International Journal On Advances in Software* 10.3&4 (2017). IARIA, pp. 263–274 (cit. on pp. 24, 28, 128, 138).

[FBBL19] M. Falkenthal, U. Breitenbücher, J. Barzen, F. Leymann. “On the Algebraic Properties of Concrete Solution Aggregation”. In: *SICS Software-Intensive Cyber-Physical Systems* (2019). Springer (cit. on pp. 28, 128, 169, 171, 174, 175).

[FBD+15] M. Falkenthal, J. Barzen, S. Dörner, V. Elkind, J. Fauser, F. Leymann, T. Strehl. “Datenanalyse in den Digital Humanities - Eine Annäherung an Kostümmuster mittels OLAP

Cubes (in English: Data Analysis in the Digital Humanities - An Approach to Costume Patterns Using OLAP Cubes)". In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS), 02. - 06.3.2015 in Hamburg, Germany.Proceedings.* Lecture Notesin Informatics (LNI). Gesellschaft für Informatik e.V. (GI), 2015, pp. 1–4 (cit. on pp. 106, 113).

[FBFL15] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. "PatternPedia – Collaborative Pattern Identification and Authoring". In: *Proceedings of Pursuit of Pattern Languages for Societal Change. The Workshop 2014*. epubli, Aug. 2015, pp. 252–284 (cit. on pp. 29, 50, 69, 70, 76, 107, 110, 113, 114, 164, 165).

[FBL18] M. Falkenthal, U. Breitenbücher, F. Leymann. "The Nature of Pattern Languages". In: *Pursuit of Pattern Languages for Societal Change*. Edition Donau-Universität Krems, 2018, pp. 130–151 (cit. on pp. 22, 29, 79, 80).

[Feh17] Fehling, Christoph. *Cloud Computing Patterns*. 2017. URL: http://www.cloudcomputingpatterns.org (cit. on p. 51).

[FEL+12] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rütschlin, D. Schumm. "Capturing Cloud Computing Knowledge and Experience in Patterns". In: *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD 2012)*. IEEE, June 2012, pp. 726–733 (cit. on p. 58).

[FHM+15] H. Finidori, T. Henfrey, N. McLaren, K. Laitner, S. Borghini, V. Puig, T. Iba, M. Pruvostcbeaurain, H. Leitner, R. Reiners, F. Leymann, M. Falkenthal. "The PLAST Project – Pattern Languages for Systemic Transformations". In: *International Journal of the Spanda Foundation* VI.1 (2015). Spanda Foundation, pp. 205–218 (cit. on pp. 14, 29).

[FJZ+12] M. Falkenthal, D. Jugel, A. Zimmermann, R. Reiners, W. Reimann, M. Pretz. “Maturity Assessments of Service-oriented Enterprise Architectures with Iterative Pattern Refinement”. In: *Lecture Notes in Informatics - Informatik 2012*. 2012, pp. 1095–1101 (cit. on p. 107).

[FL17] M. Falkenthal, F. Leymann. “Easing Pattern Application by Means of Solution Languages”. In: *Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications*. Xpert Publishing Services, 2017, pp. 58–64 (cit. on pp. 24, 28, 128).

[FLR+13] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, S. Verclas. “Service Migration Patterns - Decision Support and Best Practices for the Migration of Existing Service-based Applications to Cloud Environments”. In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2013)*. IEEE, Dec. 2013, pp. 9–16 (cit. on p. 44).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Jan. 2014, p. 367 (cit. on pp. 14, 18, 38, 41, 49, 59, 66–68, 80, 87, 89, 92–95, 98, 99, 131, 137, 166–171, 177, 178).

[Fow03] M. Fowler. *Catalog of Patterns of Enterprise Application Architecture*. 2003. URL: https://martinfowler.com/eaaCatalog/ (cit. on p. 51).

[FR21] P. Fettke, W. Reisig. “Modelling Service-Oriented Systems and Cloud Services with Heraklit”. In: *Advances in Service-Oriented and Cloud Computing*. Springer, 2021, pp. 77–89 (cit. on p. 134).

[FR23] P. Fettke, W. Reisig. *Handbook of Heraklit*. 2023. URL: https://heraklit.dfki.de/assets/documents/HERAKLIT_Handbuch_Teil_I_EN.pdf (cit. on p. 134).

[FSS+13] T. Furukawazono, I. Studies, S. Seshimo, I. Studies, D. Muramatsu, T. Iba. “Survival Language : A Pattern Language for Surviving Earthquakes”. In: *Proceedings of the 20th Conference on Pattern Languages of Programs*. ACM, 2013, Article No. 30 (cit. on p. 14).

[GBB+23] D. Georg, J. Barzen, M. Beisel, F. Leymann, J. Obst, D. Vietz, B. Weder, V. Yussupov. “Execution Patterns for Quantum Applications”. In: *Proceedings of the 18th International Conference on Software Technologies*. SciTePress, 2023, pp. 258–268 (cit. on p. 38).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Oct. 1994 (cit. on pp. 37, 40–43, 48, 80, 88, 89, 110).

[Goo23] Google. *Angular Web Framework*. 2023. URL: https://angular.io (cit. on p. 164).

[Gra20] M. Graf. “Automatisierte Aggregation von Musterimplementierungen (in English: Automated Aggregation of Pattern Implementations)”. Bachelor’s Thesis. University of Stuttgart, 2020, p. 83 (cit. on pp. 166, 167, 176–178).

[Gri11] P. Grimm. “Metamodell und Plattform für Mustersprachen und Musterkataloge (in English: Metamodel and Platform for Pattern Languages and Pattern Catalogs)”. Diploma Thesis. University of Stuttgart, 2011, p. 96 (cit. on p. 103).

[HBF+18a] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. “Pattern-based Deployment Models and Their Automatic Execution”. In: *11th IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE, 2018, pp. 41–52 (cit. on pp. 30, 44, 45).

[HBF+18b] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. “Pattern-based Deployment Models and Their Automatic Execution”. In: *Proccedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, Dec. 2018, pp. 41–52 (cit. on pp. 103, 147).

[HBF+20] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, F. Leymann. “Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration”. In: *Proceedings of the Twelfth International Conference on Pervasive Patterns and Applications (PATTERNS 2020)*. Xpert Publishing Services, Oct. 2020, pp. 40–49 (cit. on pp. 45, 102, 103, 147, 183).

[HBF+21] L. Harzenetter, U. Breitenbücher, G. Falazi, F. Leymann, A. Wersching. “Automated Detection of Design Patterns in Declarative Deployment Models”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility Cloud Computing (UCC 2021)*. ACM, 2021, pp. 36–45 (cit. on p. 44).

[Hee14] U. van Heesch. *Open Pattern Repository*. May 2014. URL: https://code.google.com/p/openpatternrepository/ (cit. on p. 50).

[Hen01] K. Henney. “C ++ Patterns - Executing Around Sequences”. In: *Proceedings of the Fifth European Conference on Pattern Languages of Programming*. Universitätsverlag Konstanz, 2001 (cit. on p. 41).

[Hoh17] Hohpe, Gregor and Woolf, Bobby. *Enterprise Integration Patterns*. 2017. URL: https://www.enterpriseintegrationpatterns.com/patterns/messaging (cit. on p. 51).

[HSB+14] A. Homer, J. Shar, L. Brader, M. Narumoto, T. Swanson. *Cloud Design Patterns: Prescriptive Architecture Guidance For Cloud Applications*. Microsoft, 2014, p. 232 (cit. on pp. 18, 92).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004 (cit. on pp. 14, 41, 45–47, 49, 59, 66, 87, 90, 99, 109, 167, 177).

[IM10] T. Iba, T. Miyake. “Learning patterns: a pattern language for creative learners II”. In: *Proceedings of the 1st Asian Conference on Pattern Languages of Programs (AsianPLoP 2010)*. ACM, 2010, pp. I-41–I-58 (cit. on p. 14).

[JM22] D. Jaschke, S. Montangero. *Is quantum computing green? An estimate for an energy-efficiency quantum advantage*. 2022. URL: https://arxiv.org/abs/2205.12092 (cit. on p. 90).

[KF14] D. Krleža, K. Fertalj. “A method for situational and guided information system design”. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications*. Xpert Publishing Services, 2014, pp. 70–78 (cit. on p. 43).

[Koh10] C. Kohls. “The structure of patterns”. In: *Proceedings of the 17th Conference on Pattern Languages of Programs*. New York: ACM, 2010 (cit. on pp. 13, 88).

[Koh11] C. Kohls. “The Structure of Patterns: Part II - Qualities”. In: *Proceedings of the 18th Conference on Pattern Languages of Programs*. New York: ACM, 2011 (cit. on p. 88).

[Koh12] C. Kohls. “The path to patterns - introducing the path metaphor”. In: *Proceedings of the 17th European Conference on Pattern Languages of Programs*. ACM, 2012 (cit. on pp. 40, 88).

[Kri18a] C. Krieger. *Repository of the Linked Open Patterns Client*. 2018. URL: https://github.com/PatternAtlas/linkedOpenPatternClient (cit. on pp. 139, 166).

[Kri18b] C. Krieger. *Repository of the SPARQL Service*. 2018. URL: https://github.com/PatternAtlas/sparqlService (cit. on p. 166).

[Kri18c] C. Krieger. "Semantic Querying of Distributed Pattern and Solution". Master's Thesis. University of Stuttgart, 2018, p. 75 (cit. on pp. 80, 105, 113, 114, 128, 149, 152, 164, 166).

[KU09] C. Kohls, J.-g. Uttecht. "Lessons learnt in mining and writing design patterns for educational interactive graphics". In: *Computers in Human Behavior* 25.5 (2009), pp. 1040–1055 (cit. on pp. 40, 108).

[LB21] F. Leymann, J. Barzen. "Pattern Atlas". In: *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future*. Ed. by M. Aiello, A. Bouguettaya, D. Tamburri, W.-J. van den Heuvel. Springer, 2021, pp. 67–76 (cit. on pp. 50, 53, 150, 165).

[LBF+20] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, K. Wild. "Quantum in the Cloud : Application Potentials and Research Opportunities". In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SciTePress, 2020, pp. 9–24 (cit. on p. 72).

[Ley19] F. Leymann. "Towards a Pattern Language for Quantum Algorithms". In: *Quantum Technology and Optimization Problems*. Vol. 11413. Lecture Notes in Computer Science (LNCS). Springer, 2019, pp. 218–230 (cit. on p. 164).

[Ma13] Z. Ma. "Process fragments: enhancing reuse of process logic in BPEL process models". Dissertation. University of Stuttgart, 2013 (cit. on p. 123).

[MD97] G. Meszaros, J. Doble. “Pattern Languages of Program Design 3”. In: Addison-Wesley, 1997. Chap. A Pattern Language for Pattern Writing, pp. 529–574 (cit. on pp. 15, 40, 41, 58, 68, 88).

[MDM+18] H. Marouane, C. Duvallet, A. Makni, R. Bouaziz, B. Sadeg. “An UML profile for representing real-time design patterns”. In: *Journal of King Saud University - Computer and Information Sciences* 30.4 (2018), pp. 478–497 (cit. on p. 103).

[Mic14] Microsoft. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. 2014. URL: https://msdn.microsoft.com/en-us/library/dn568099.aspx (cit. on p. 92).

[MT14] A. G. Mirnig, M. Tscheligi. “Building a General Pattern Framework via Set Theory : Towards a Universal Pattern Approach”. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services, 2014, pp. 8–11 (cit. on p. 39).

[NBL14] A. Nowak, U. Breitenbücher, F. Leymann. “Automating Green Patterns to Compensate CO2 Emissions of Cloud-based Business Processes”. In: *Proceedings of the Eighth International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2014)*. Xpert Publishing Services, Aug. 2014, pp. 132–139 (cit. on p. 90).

[NL13] A. Nowak, F. Leymann. “Green Business Process Patterns - Part II”. In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*. IEEE, Dec. 2013, pp. 168–173 (cit. on p. 90).

[NLS+11] A. Nowak, F. Leymann, D. Schleicher, D. Schumm, S. Wagner. "Green Business Process Patterns". In: *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP 2011)*. ACM, Oct. 2011 (cit. on pp. 90, 99).

[Now14] A. Nowak. "Green Business Process Management: Methode und Realisierung (in English: Green Business Process Management: Method and Implementation)". Dissertation. University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, 2014 (cit. on p. 90).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2013 (cit. on pp. 102, 117, 168).

[Obj14] Object Management Group. *Object Constraint Language Specification 2.4*. 2014. URL: https://www.omg.org/spec/OCL/2.4/PDF (cit. on p. 103).

[OMG07] OMG. *OMG Unified Modeling Language (UML)*. Object Management Group (OMG). 2007 (cit. on p. 103).

[Ope22] Open Group. *ArchiMate 3.2 Specification*. 2022. URL: https://publications.opengroup.org/archimate-library/archimate-standards/c226?_ga=2.138430195.1181970343.1667419594-1657458596.1667419594 (cit. on p. 102).

[PHPR09] L. Pavlič, M. Heričko, V. Podgorelec, I. Rozman. "Improving Design Pattern Adoption with an Ontology-Based Repository". In: *Informatica* 33 (2009), pp. 189–197 (cit. on p. 50).

[Pla23a] PlanQK Project Consortium. *PlanQK Pattern Repository*. 2023. URL: https://patterns.platform.planqk.de (cit. on p. 164).

[Pla23b] PlanQK Project Consortium. *PlanQK Platform*. 2023. URL: https://platform.planqk.de (cit. on p. 164).

[Pla23c] PlanQK Project Consortium. *The PlanQK Research Project*. 2023. URL: https://planqk.de (cit. on p. 164).

[Pra09] D. R. Prasanna. *Dependency Injection*. Manning Publications, 2009, p. 330 (cit. on p. 46).

[RBF+16] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns”. In: *Proceedings of the 21th European Conference on Pattern Languages of Programs*. ACM, 2016 (cit. on pp. 14, 29, 38, 41, 49, 99).

[RBF+17a] L. Reinfurt, U. Breitenbücher, M. Falkenthal, P. Fremantle, F. Leymann. “Internet of Things Security Patterns”. In: *Proceedings of the 24th Conference on Pattern Languages of Programs*. ACM, 2017 (cit. on pp. 14, 30).

[RBF+17b] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns for Device Bootstrapping and Registration”. In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. ACM, 2017 (cit. on pp. 14, 30).

[RBF+17c] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns for Devices”. In: *Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications*. Xpert Publishing Services, 2017, pp. 117–126 (cit. on pp. 14, 29, 99).

[RBF+17d] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns for Devices: Powering, Operating, and Sensing”. In: *International Journal on Advances in Internet Technology* (2017). IARIA, pp. 106–123 (cit. on p. 30).

[RBF+19] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns for Communication and Management”. In: *Transactions on Pattern Languages of Programming* IV (2019) (cit. on pp. 14, 30).

[Ree03] T. Reenskaug. *The Model-View-Controller (MVC) Its Past and Present*. 2003. URL: http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/HM1A93.html (cit. on p. 109).

[Rei12] R. Reiners. "A Pattern Evolution Process - From Ideas to Patterns". In: *Informatiktage 2012*. Vol. S-11. LNI. GI, Mar. 2012, pp. 115–118 (cit. on p. 106).

[Rei13] R. Reiners. "An Evolving Pattern Library for Collaborative Project Documentation". Dissertation. RWTH Aachen University, 2013 (cit. on pp. 40, 48, 50, 60, 69, 72, 80, 99, 106, 107).

[Rei17] Reinfurt, Lukas et al. *Internet of Things Patterns*. 2017. URL: http://www.internetofthingspatterns.com (cit. on p. 51).

[Rei18] W. Reisig. "Associative composition of components with double-sided interfaces". In: *Acta Informatica* (56 Oct. 2018) (cit. on p. 118).

[RFBL17] L. Reinfurt, M. Falkenthal, U. Breitenbücher, F. Leymann. "Applying IoT Patterns to Smart Factory Systems". In: *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2017, pp. 1–10 (cit. on pp. 14, 30).

[RFJZ13] R. Reiners, M. Falkenthal, D. Jugel, A. Zimmermann. "Requirements for a Collaborative Formulation Process of Evolutionary Patterns". In: *Proceedings of the 18th European Conference on Pattern Languages of Programs EuroPlop*. ACM, 2013, Article No. 16 (cit. on p. 29).

[RFL20] L. Reinfurt, M. Falkenthal, F. Leymann. "Where to Begin - On Pattern Language Entry Points". In: *SICS Software-Intensive Cyber-Physical Systems* 35 (2020). Springer (cit. on p. 30).

[Ric18] C. Richardson. *Microservices Patterns*. Manning Publications, 2018, p. 490 (cit. on pp. 38, 46, 49, 99).

[Ric20] C. Richardson. *Design Patterns in the Spring Framework*. 2020. URL: https://microservices.io (cit. on pp. 51, 99).

[RJ07] E. Razina, D. Janzen. "Effects of dependency injection on maintainability". In: *Proceedings of the 11th International Conference of Software Engineering and Applications*. ACTA Press, 2007, pp. 7–12 (cit. on p. 46).

[RS09] I. Reinhartz-Berger, A. Sturm. "Utilizing domain models for application design and validation". In: *Information and Software Technology* 51.8 (2009), pp. 1275–1289 (cit. on p. 103).

[SBF+19] K. Saatkamp, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann. "An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models using First-order Logic". In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 495–506 (cit. on p. 30).

[SBK+18] K. Saatkamp, U. Breitenbücher, K. Képes, F. Leymann, M. Zimmermann. "OpenTOSCA Injector: Vertical and Horizontal Topology Model Injection". In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Vol. 10797. Lecture Notes in Computer Science (LNCS). Springer International Publishing, Jan. 2018, pp. 379–383 (cit. on p. 170).

[SBLE12] D. Schumm, J. Barzen, F. Leymann, L. Ellrich. "A Pattern Language for Costumes in Films". In: *Proceedings of the 17th European Conference on Pattern Languages of Programs*. Article no. 7. 2012 (cit. on pp. 66, 96, 99, 173).

[SC16] A. L. Santos, D. Coelho. "Java Extensions for Design Pattern Instantiation". In: *Proceedings of the International Conference on Software Reuse*. IEEE, 2016, pp. 284–299 (cit. on p. 43).

[Sca] Scaled Agile. *SAFe - Scaled Agile Framework*. URL: `https://www.scaledagileframework.com` (cit. on p. 77).

[Sch10] T. Scheibler. “Ausführbare Integrationsmuster (in English: Executable Integration Patterns)”. Dissertation. University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, 2010 (cit. on p. 45).

[Sch97] K. Schwaber. “SCRUM Development Process”. In: *Business Object Design and Implementation*. Springer London, 1997, pp. 117–134 (cit. on p. 77).

[Sko17] M. Skouradaki. “Workload mix definition for benchmarking BPMN 2.0 Workflow Management Systems”. Dissertation. University of Stuttgart, 2017 (cit. on p. 123).

[Sou95] J. Soukup. “Implementing Patterns”. In: *Pattern Languages of Program Design*. ACM Press, 1995, pp. 395–412 (cit. on p. 41).

[Str15] T. Strehl. “Identifikation von Musterindikatoren mit Methoden des visuellen Data Mining für multivariate kategoriale Daten (in English: Identification of pattern indicators with methods of visual data mining for multivariate categorical data)”. Master’s Thesis. University of Applied Sciences Reutlingen, 2015, p. 116 (cit. on pp. 106, 113).

[SU23] M. Syed, Z. Ul Abadin. “A Pattern for Proof of Stake Consensus Algorithm in Blockchain”. In: *Proceedings of the 27th European Conference on Pattern Languages of Programs*. ACM, 2023 (cit. on p. 38).

[VMW23] VMWare. *The Spring Boot Project*. 2023. URL: `https://spring.io/projects/spring-boot` (cit. on p. 164).

[W3C23] W3C. *SPARQL 1.1 Overview*. 2023. URL: `https://www.w3.org/TR/sparql11-overview/` (cit. on p. 166).

[WBB+20] M. Weigold, J. Barzen, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Wild. “Pattern Views: Concept and Tooling for Interconnected Pattern Languages”. In: *Communications in Computer and Information Science* 1310 (2020), pp. 86–103 (cit. on pp. 22, 31, 51, 150).

[WBFL17] J. Wettinger, U. Breitenbücher, M. Falkenthal, F. Leymann. “Collaborative gathering and continuous delivery of DevOps solutions through repositories”. In: *Computer Science - Research and Development* 32.3-4 (2017) (cit. on p. 76).

[WBLS20] M. Weigold, J. Barzen, F. Leymann, M. Salm. “Data Encoding Patterns For Quantum Algorithms”. In: *Proceedings of the 27th Conference on Pattern Languages of Programs (PLoP ’20)*. ACM, 2020, pp. 1–11 (cit. on pp. 14, 38).

[WBLS21] M. Weigold, J. Barzen, F. Leymann, M. Salm. “Encoding patterns for quantum algorithms”. In: *IET QuantumCommunication* 2.4 (2021), pp. 141–152 (cit. on pp. 14, 38).

[WBLV21] M. Weigold, J. Barzen, F. Leymann, D. Vietz. “Patterns For Hybrid Quantum Algorithms”. In: *Proceedings of the 15th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2021)*. Springer, 2021, pp. 34–51 (cit. on pp. 38, 99).

[Wei19] M. Weigold. “Use of standards and technologies of the Semantic Web for the representation of pattern languages”. Master’s Thesis. University of Stuttgart, 2019, p. 67 (cit. on p. 164).

[Wel94] D. S. Weld. “An Introduction to Least Commitment Planning”. In: *AI Magazine* 15.4 (1994), pp. 27–61 (cit. on p. 117).

[Wet17] J. Wettinger. “Gathering Solutions and Providing APIs for their Orchestration to Implement Continuous Software Delivery”. Dissertation. University of Stuttgart, 2017, p. 239 (cit. on pp. 72, 76).

[WF12] T. Wellhausen, A. Fiesser. “How to Write a Pattern?: A Rough Guide for First-time Pattern Authors”. In: *Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP 2011)*. ACM, July 2012 (cit. on p. 68).

[Wil22] K. Wild. “Eine Methode zum Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen (in English: A method for distributing, adapting and deploying cross-partner applications)”. Dissertation. University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, 2022 (cit. on p. 170).

[WKWV20] S. Waseeb, W. S. Khail, H. G. Wahaj, V. Vranić. “Extracting Relations Between Organizational Patterns Using Association Mining”. In: *Proceedings of the 25th European Conference on Pattern Languages of Programs*. ACM, 2020 (cit. on p. 39).

[WV03] M. V. Welie, G. C. V. D. Veer. “Pattern Languages in Interaction Design : Structure and Organization”. In: *Human-Computer Interaction (INTERACT) ’03: IFIP TC13 International Conference on Human-Computer Interaction*. IOS Press, 2003, pp. 527–534 (cit. on p. 49).

[WV23] S. Waseeb, V. Vranić. “Toward Organizational Pattern Ontology”. In: *Proceedings of the 27th European Conference on Pattern Languages of Programs*. ACM, 2023 (cit. on p. 40).

[YBB+22] V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, J. Soldani. “Serverless or Serverful? A Pattern-Based Approach for Exploring Hosting Alternatives”. In: *Service-Oriented Computing*. Springer, 2022, pp. 45–67 (cit. on p. 38).

[ZAHD08] U. Zdun, P. Avgeriou, C. Hentrich, S. Dustdar. “Architecting as Decision Making with Patterns and Primitives”. In: *Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2008)*. ACM, May 2008, pp. 11–18 (cit. on p. 92).

[ZB15] H. Zhu, I. Bayley. “On the Composability of Design Patterns”. In: *IEEE Transactions on Software Engineering* 41 (2015), pp. 1138–1152 (cit. on p. 39).

[Zdu07] U. Zdun. “Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis”. In: *Software: Practice & Experience* 37.9 (July 2007), pp. 983–1016 (cit. on pp. 16, 80, 120).

[ZHD07] U. Zdun, C. Hentrich, S. Dustdar. “Modeling process-driven and service-oriented architectures using patterns and pattern primitives”. In: *ACM Transactions on the Web* 1.3 (Sept. 2007), 14–es (cit. on p. 80).

[Zhu14] H. Zhu. “Towards a General Theory of Patterns”. In: *Cyberpatterns: Unifying Design Patterns with Security and Attack Patterns*. Ed. by C. Blackwell, H. Zhu. Springer, 2014, pp. 57–69 (cit. on p. 39).

[Zim95] W. Zimmer. “Relationships between Design Patterns”. In: *Pattern Languages of Program Design*. Addison-Wesley, 1995, pp. 345–364 (cit. on p. 80).

[ZKV04] U. Zdun, M. Kircher, M. Völter. “Remoting patterns: Design reuse of distributed object middleware solutions”. In: *IEEE Internet Computing* 8.6 (2004), pp. 60–66 (cit. on p. 84).

All links were last accessed on 2023-06-13.

List of Figures

1.1 Current state of the art of pattern application 15
1.2 Vision of pattern application 17
1.3 Overview of the research contributions 27

3.1 Conceptual overview of the EINSTEIN-Method 63
3.2 A selection of patterns is translated into a design model . 65

4.1 Excerpt of Remoting Patterns 84
4.2 The abstraction gap 90
4.3 Closing the abstraction gap 91
4.4 Pattern refinement 94
4.5 Refinement of the Cloud Computing Patterns 95
4.6 Refinement of Costume Patterns 97
4.7 Pattern languages and design models 100

5.1 Connecting patterns and concrete solutions 108
5.2 Essential invariants of concrete solutions 112
5.3 Selection criteria of concrete solutions 115
5.4 Filtering of concrete solutions with selection criteria . . 116
5.5 Requirements and capabilities of concrete solutions . . . 118

5.6 Compatibility of concrete solutions 119
5.7 Pattern graph with connected concrete solutions 120
5.8 Modeling of a solution model 122

6.1 Lack of guidance for aggregating concrete solutions . . . 129
6.2 Solution algebras 132
6.3 Pattern graph with connected solution language 137
6.4 Example of a concrete solution aggregation descriptor . . 139
6.5 Concept of a concrete solution aggregation program . . . 141

7.1 Interplay of the concepts to support the EINSTEIN-Method 151
7.2 Assisted aggregation of concrete solutions 156

8.1 Architecture of a Toolchain 165
8.2 Aggregation program for topology models 169
8.3 Aggregation program for management planlets 171
8.4 Aggregation operator for Costumes in Films I 174
8.5 Aggregation operator for Costumes in Films II 175
8.6 Aggregation Scenario with Cloud Computing Patterns and Enterprise Integration Patterns 177

List of Symbols

Symbol	Description
$\mathscr{D}$	Set of all domains of types of edges (Definition 4.1.1)
$\mathscr{G}$	Pattern language graph (Definition 4.1.1)
$\mathscr{P}$	Set of patterns (Definition 4.1.1)
$\mathscr{E}_{\mathscr{P}}$	Set of edges among patterns in a pattern language (Definition 4.1.1)
$\mathscr{W}$	Set of pattern language edge weights (Definition 4.1.1)
α	Map that types pattern language edge weights (Definition 4.1.1)
β	Map that assigns type-specific descriptions to pattern language edges (Definition 4.1.1)
$\mathfrak{G}$	Set of all pattern language graphs (Definition 4.1.2)
$\odot$	Pattern language aggregator (Definition 4.1.2)
$\trianglelefteq$	Pattern language subsetting operator (Definition 4.1.3)

Symbol	Description
$\mathcal{E}$	Set of new edges to connect pattern languages (Definition 4.1.2)
$P(\mathcal{G})$	Map that maps $\mathcal{G}$ to the set of patterns contained in $\mathcal{G}$ (Definition 4.4.1)
$\mathcal{G}_{dm}$	Design model (Definition 4.4.1)
$\mathscr{P}_{dm}$	Set of patterns contained in a design model (Definition 4.4.1)
$\mathscr{E}_{dm}$	Set of edges among patterns in a design model (Definition 4.4.1)
$\mathscr{W}_{dm}$	Set of edge weights representing domain-specific semantics of the interplay of patterns in a design model (Definition 4.4.1)
α_{dm}	Map that types edge weights in a design model (Definition 4.4.1)
β_{dm}	Map that assigns type-specific descriptions to edges in a design model (Definition 4.4.1)
$\mathscr{T}$	Set of concrete solution types (Definition 5.2.1)
$\mathscr{D}_{PT}$	Set of all domains of concrete solution property types (Definition 5.2.1)
$\mathfrak{S}$	Set of all concrete solutions (Definition 5.2.1)
ID	Set of all concrete solution identifiers (Definition 5.2.1)
id	Id of a concrete solution in a concrete solution descriptor (Definition 5.2.1)
$type$	Type of a concrete solution in a concrete solution descriptor (Definition 5.2.1)
$Props$	Set of concrete solution properties in a concrete solution descriptor (Definition 5.2.1)
γ	Map that assigns schemas to properties of concrete solution descriptors (Definition 5.2.1)

Symbol	Description
δ	Map that assigns values to properties of concrete solution descriptors (Definition 5.2.1)
$\mathcal{G}_{\mathcal{P}\mathcal{S}}$	Pattern graph with connected concrete solutions (Definition 5.5.1)
$\mathcal{P}$	Set of patterns contained in a pattern graph with connected concrete solutions (Definition 5.5.1)
$\mathcal{S}$	Set of concrete solutions contained in a pattern graph with connected concrete solutions (Definition 5.5.1)
$\mathcal{E}_{\mathcal{P}}$	Set of edges among patterns in a pattern graph with connected concrete solutions (Definition 5.5.1)
$\mathcal{E}_{\mathcal{P}\mathcal{S}}$	Set of edges among patterns and concrete solutions in a pattern graph with connected concrete solutions (Definition 5.5.1)
$\mathcal{G}_{sm}$	Solution model (Definition 5.6.1)
$\mathcal{S}_{sm}$	Set of concrete solutions contained in a solution model (Definition 5.6.1)
$\mathcal{E}_{sm}$	Set of edges among concrete solutions in a solution model (Definition 5.6.1)
$\mathcal{W}_{sm}$	Set of edge weights representing domain-specific semantics of the interplay of concrete solutions in a solution model (Definition 5.6.1)
$(A_t)_{t\in\Theta}$	Family of types of concrete solutions (Definition 6.1.1)
Θ	Index set whose elements identify all types of concrete solutions (Definition 6.1.1)
A_t	Type of concrete solutions (Definition 6.1.1)
$(\oplus_j)_{j\in J}$	Family of aggregation operators (Definition 6.1.2)
$((A_t)_{t\in\Theta}, (\oplus_j)_{j\in J})$	Solution algebra (Definition 6.1.2)

Symbol	Description
$\mathfrak{DG}$	Set of all detector graphs (Definition 6.3.1)
$\oplus$	Aggregation operator (Definition 6.3.1)
DG	Detector graph of a solution aggregation program (Definition 6.3.1)
κ	Aggregatability analysis function of a solution aggregation program (Definition 6.3.1, Definition 6.3.5)
o	Aggregation operation function of a solution aggregation program (Definition 6.3.1, Definition 6.3.6)
σ	Solution model update function (Definition 6.3.1, Definition 6.3.7)
ot	Operation target of a detector graph (Definition 6.3.2)
R	Set of requirements of an operation target (Definition 6.3.2)
C	Set of capabilities of an operation target (Definition 6.3.2)
t	Type of an operation target (Definition 6.3.2)
$\mathfrak{OT}$	Set of all operation targets (Definition 6.3.3)
OT	Set of operation targets of a detector graph (Definition 6.3.3)
E_{OT}	Set of edges among operation targets of a detector graph (Definition 6.3.3)
$\mathfrak{SM}$	Set of all solution models (Definition 6.3.4)
$\mathfrak{AP}$	Set of all aggregation programs (Definition 6.3.4)
ap	Concrete aggregation program (Definition 6.3.4)
ac	Concrete aggregation context (Definition 6.3.4)
$\mathfrak{AC}$	Set of all aggregation contexts (Definition 6.3.5)

Symbol	Description
$\hat{S}$	Set of concrete solutions to be replaced by σ (Definition 6.3.7)
$\bar{E}$	Set of edges between aggregated concrete solutions (Definition 6.3.7)
E_{π_1}	Set of edges where the start of the edges has to be replaced by the aggregate (Definition 6.3.7)
E_{π_2}	Set of edges where the target of the edges has to be replaced by the aggregate (Definition 6.3.7)
$\mathfrak{E}_{\mathfrak{S}\mathfrak{M}}$	Set of edges of all solution models (Definition 6.3.7)
ξ_{π_1}	Function that replaces the start of an edge with the concrete solution aggregate (Definition 6.3.7)
ξ_{π_2}	Function that replaces the start of an edge with the concrete solution aggregate (Definition 6.3.7)
$\dot{s}$	Concrete solution aggregate (Definition 6.3.7)
$\dot{\mathcal{G}}_{sm}$	Updated solution model (Definition 6.3.7)
C	Set of all costumes (Definition 8.2.1)
$\mathcal{B}$	Set of all base elements of costumes (Definition 8.2.1)
$T_{\mathcal{B}}$	Set of all base element types (Definition 8.2.1)
$\oplus_{\cup}$	Aggregation operator for the union of costumes (Definition 8.2.2)
$\oplus_{\bullet}$	Aggregation operator for the union of Costumes based on selected base element types (Definition 8.2.3)

List of Algorithms

7.1 Aggregation Step 154
7.2 Find Subgraph Mappings 155

List of Definitions

2.1.1 Pattern – informal 36

2.1.2 Alexandrian Pattern Language – informal 37

3.3.1 Pattern-based Design Model – informal 67

3.3.2 Concrete Solution – informal 70

3.3.3 Selection Criteria – informal 71

3.3.4 Aggregation Operator – informal 73

4.1.1 Pattern Language 81

4.1.2 Pattern Language Aggregator 86

4.1.3 Pattern Language Subsetting Operator 87

4.4.1 Pattern-based Design Model 101

5.2.1 Concrete Solution 113

5.5.1 Pattern Graph with connected Concrete Solutions 119

5.6.1 Solution Model 124

6.1.1 Types of Concrete Solutions 130
6.1.2 Solution Algebra and Aggregation Operators 130
6.3.1 Concrete Solution Aggregation Program 140
6.3.2 Operation Target 142
6.3.3 Detector Graph 142
6.3.4 Aggregation Context 143
6.3.5 Aggregatability Analysis Function 144
6.3.6 Aggregation Operation Function 145
6.3.7 Solution Model Update Function 145
8.2.1 Set of Costumes, Base Elements, and Base Element Types . 173
8.2.2 Aggregation Operator $\oplus_{\cup}$ 175
8.2.3 Aggregation Operator $\oplus_{\bullet}$ 176