Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains

Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany, lastname@iaas.uni-stuttgart.de

Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains

Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, and Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
{falkenthal, barzen, breitenbuecher, fehling, leymann}@iaas.uni-stuttgart.de

Abstract—**Patterns are a well-known and often used concept applied in various domains. They document proven solutions to recurring problems in a specific context and in a generic way. As a result, patterns are applicable in a multiplicity of specific use cases. However, since the concept of patterns aims at generalization and abstraction of solution knowledge, it is difficult to apply patterns to specific use cases, as the required knowledge about refinement and the manual effort that has to be spent is often immense. Therefore, we introduce the concept of Solution Implementations, which are concrete solution artifacts directly associated with patterns in order to efficiently support elaboration of concrete pattern implementations. In addition, we show how Solution Implementations can be aggregated to solve problems that require the application of multiple patterns at once. We evaluate the presented approach by conducting use cases in the following domains: (i) Cloud Application Architecture, (ii) Cloud Application Management, (iii) Costumes in Films, (iv) User Interaction Design, and (v) Object-Oriented Software Engineering.**

Keywords-pattern languages, solution implementations, pattern application, cloud computing patterns, costume patterns

I. INTRODUCTION

Patterns and pattern languages are well-established concepts in different application areas of computer science and information technology (IT) [1]. Originally introduced to the domain of building architecture [2], the concept of patterns recently got more and more popular in different domains such as education [3], design engineering [4], user interaction design [5], large-scale emergeny management [6], software architecture [7], enterprise application architecture [8], enterprise architecture management [9], cloud application architecture [10], application security [11] or costumes [12]. Patterns are used to document proven solutions to recurring problems in a specific context. However, since the concept of patterns aims at generalization and abstraction, it is often difficult to apply the captured abstracted knowledge to a concrete problem. Thus, pattern application often requires immense manual effort and domain-specific knowledge to refine the abstract, conceptual, and high-level solution description of a pattern to an individual use case. These following examples show that this problem occurs in several domains due to the abstraction of solution knowledge into patterns. For example, if a PHP: Hypertext Preprocessor (PHP) [13] developer uses the patterns by Gamma et al. [14], he or she is faced with the problem that the general solution concepts of the patterns have to be translated to his or her concrete context, i.e., he or she has to implement solutions based on a given programming paradigm predefined by PHP. An enterprise architect who has to integrate complex legacy systems may use the enterprise application architecture patterns by Fowler [8] or the enterprise integration patterns by Hohpe and Wolf [15] to gain insight to proven solutions of his or her problems; but, these are still generic solutions and he or she has to create proper implementations for the systems to integrate. This can lead to huge efforts since he or she also has to consider many constraints given by the running systems and technologies besides paradigms of the used programming languages. A teacher who uses the learning patterns by Iba and Miyake [3] has to adapt them to match his or her prevailing school system with all the teaching methods. To give a final example, a costume designer could use the patterns by Schumm et al. [12] to find clothing conventions for a cowboy in a western film but he or she still has to come up with a specific solution for the specific film.

The above examples show that it is often time consuming to create concrete solutions from patterns, since patterns in general describe proven generic solutions at a conceptual level. To overcome this problem, we suggest that patterns should be linked to the (i) original concrete solutions from which they have been deduced (if available) and (ii) to individual new concrete implementations of the abstractly described solution. Therefore, we introduce the concept of *Solution Implementations* that enables users who want to apply a certain pattern to reuse already existing implementation artifacts for their use cases, which eases the application of patterns and reduces the required manual effort significantly. In addition, our concept supports avoiding errors of manual refinement, since existing solution artifacts can be looked up from patterns.

This paper is an extended version of our former work [1] in which we presented Solution Implementations at the Sixth International Conference on Pervasive Patterns and Applications (PATTERNS 2014). In this article, we now validate the approach of Solution Implementations in detail by conducting additional use cases to show that the concept is domain-agnostic and fundamental in the field of pattern research. The studies covered in this article are conducted in the following domains: (i) Cloud Application Architecture,

(ii) Cloud Application Management, (iii) Costumes in Films, (iv) User Interaction Design, and (v) Object-Oriented Software Engineering.

The remainder of this paper is structured as follows: we clarify the difference between the common concept of pattern solutions and Solution Implementations as separate concrete solution artifacts in Section II. In Section III, we discuss related work and the lack of directly usable concrete solutions in state of the art pattern research. We show how to keep patterns linked to concrete solution knowledge in the form of Solution Implementations and how to select Solution Implementations to establish concrete solution building blocks, which can be aggregated in Section IV. In Section V, we present detailed use cases to show the applicability of the presented concept. We verify the feasibility of the approach by means of implemented prototypes in Section VI and conclude this paper with an outline of future work in Section VII.

II. Motivation

Patterns are human readable artifacts, which combine problem knowledge with generic solution knowledge. Patterns are often organized as pattern languages, i.e., they are related. All patterns of a pattern language follow a canonic pattern format, which is a *template* for documenting all contained patterns. This format typically defines different sections such as "Problem", "Context", "Solution", and "Known Uses". The problem and context sections describe the problem to be solved in an abstract manner where the solution section describes the general characteristics of the solution in an abstract way. Thus, the general solution is refined for individual problem manifestations and use cases resulting in different concrete solutions every time the pattern is applied. The known uses section is the only place where concrete solutions from which the pattern has been abstracted are described. The description in the known uses section is also only textually but concrete solution artifacts are not related to patterns. Further, the known uses are commonly not extended as the pattern is applied nor do they guide pattern readers during the creation of their own solutions.

Therefore, due to the abstract nature of patterns and generalized issues, most pattern languages only contain some concrete solutions a pattern was derived from in the known uses section. This leads to the problem that the user of the pattern has to design and implement a specific solution based on his individual and concrete use case, i.e., a solution has to be implemented based on the user's circumstances considering the given pattern. However, many patterns are applied several times to similar use cases. Thus, the effort has to be spent every time for tasks, which were already performed multiple times. For example, the Model-View-Controller (MVC) [16] Design Pattern is an often-used pattern in the domain of user interface design. This pattern was, therefore, implemented for many applications in many programming languages from scratch, as patterns typically provide no directly usable concrete solutions for use cases in a concrete context. Patterns are not linked with a growing list of solutions that can be used as basis to apply them to individual use cases rapidly: each time a pattern should be applied, it has to be refined manually to the current use case. The provided sections such as "Known Uses" and "Examples", which are part of the pattern structure in most pattern languages [15][17][18], therefore, support the reader in creating new solutions only partially: they provide only partial solution refinements or solution templates as written text but not directly applicable implementations that can be used without additional effort. Thus, the reader of a pattern is faced with the problem of creation and design to elaborate a proper solution based on a given pattern each time when it has to be applied – which results in time-consuming efforts that decrease the efficiency of using patterns.

As of today, patterns are typically created by small groups of experts. By abstracting the problems and solutions into patterns relying on their expertise, these experts determine the content of the patterns. This traditional way of pattern identification, also called the "pattern guru approach" by Reiners et al. [19], creates the two issues already seen: first, the patterns are only hardly verifiable because the concrete solutions they have been abstracted from are mostly not traceable ("pattern provenance") and second, the patterns document abstracted knowledge, therefore manual effort and specific knowledge is needed to apply them to concrete problems.

Another problem occurs if multiple patterns have to be combined to create a concrete solution. Pattern languages tackle the problem of selecting and applying multiple related patterns to solve overall problems. As shown by Zdun [20], this can be supported by defining relationships between patterns within a pattern language, which assure that connected patterns match together semantically, i.e., that they are composable regarding their solutions. This means that patterns can be used as composable building blocks to create overall solutions. Once patterns are composed to create overall solutions the problem arises that concrete solutions have to be feasible in the context of concrete problem situations. Referring to the former mentioned example of a PHP developer, the overall concrete solution, consisting of the concrete solutions of the composed patterns, has to be elaborated that it complies with the constraints defined by the programming language PHP. So, the complexity of creating concrete solutions from composed patterns increases with the number of aggregated solutions, since integration efforts add to the efforts of elaborating each individual solution. Thus, to summarize the discussion above, we need a means to support the required refinement from a pattern's abstract solution description to directly applicable concrete solutions and their composition.

III. RELATED WORK

As patterns are human readable artifacts, the template documenting a pattern contains solution sections presenting solution knowledge as ordinary text [2][7][14][18]. This kind of solution representation contains the general principle and core of a solution in an abstract way. Common solution sections of patterns do not reflect concrete solution instances of the pattern. They only provide conceptual sketches of a solution or describe the essence of the solution textually. Thus, they just act like manuals to support a reader at implementing a solution proper for his issues, but they do not provide concrete solution artifacts.

Iterative pattern formulation approaches as shown by Reiners et al. [19][21] and Falkenthal et al. [22] can enable that concrete solution knowledge arising from running projects is used to formulate patterns. Patterns are not just final artifacts but are formulated based on initial ideas in an iterative process to finally reach the status of a pattern. Nevertheless, in these approaches concrete solution knowledge only supports the formulation process of patterns but is not stored in the form of concrete solution artifacts explicitly to get reused when a pattern is applied.

Porter et al. [23] have shown that selecting patterns from a pattern language is a question of temporal ordering of the selected patterns. They show that combinations and aggregations of patterns rely on the order in which the patterns have to be applied. This leads to so called pattern sequences which are partially ordered sets of patterns reflecting the temporal order of pattern application. This approach focuses on combinability of patterns, but not on the combinability of concrete solutions.

Many pattern collections and pattern languages are stored in digital pattern repositories such as presented by Reiners [3], Fehling [24] and van Heesch [25]. Although these repositories support readers in navigating through the patterns they do not link concrete solutions with the patterns. Therefore, readers have to manually recreate concrete solutions each time when they want to apply a pattern.

Zdun [20] shows that pattern languages can be represented as graphs with weighted edges. Patterns are the nodes of the graph and edges are relationships between the patterns. The weights of the edges represent the semantics of the relationships as well as the effects of a pattern on the resulting context of a pattern. These effects are called goals and reflect the influence of a pattern on the quality attributes of software architectures. While this approach helps to select proper pattern sequences from a pattern language it does not enable to find concrete solutions and connect them together.

Demirköprü [26] shows that Hoare logic can be applied to patterns and pattern languages such that patterns are getting enriched by preconditions and postconditions. By considering this conditions, pattern sequences can be connected into aggregates, respectively compositions of patterns where preconditions of the first pattern of the sequence are the preconditions of the aggregate and postconditions of the last pattern in the sequence are accordingly the postconditions of the aggregate. This approach only tackles aggregation of patterns without considering concrete solutions.

Fehling et al. [27][28] show that their structure of cloud computing patterns can be extended to annotate patterns with additional implementation artifacts. Those artifacts can represent instantiations of a pattern on a concrete cloud platform. Considering those annotations, developers can be guided through configurations of runtime environments. Although patterns can be annotated with concrete implementation artifacts, this approach is only described in the domain of cloud computing and must be extended to other domains in order to introduce a means to ease pattern usage and refinement in general.

Mirnig and Tscheligi [29] introduce a general pattern framework based on set theory. This framework provides a general theory of patterns in order to explicate knowledge in pattern structures and relate patterns into pattern languages. Their approach is general due to the definition of patterns and pattern languages by means of set theory and, therefore, provides a domain independent fundamental method to create patterns and pattern languages. Further, they introduce a conceptual mechanism by means of descriptors and targets to combine patterns from different domains, respectively pattern languages. Nevertheless, the approach only deals with abstracted solution knowledge that is captured into patterns and related into pattern languages. Hence, the approach lacks support to deal with concrete solutions. Besides, the approach only describes to combine patterns by means of descriptors and targets in general, but it does not clarify how patterns may work together in concrete use cases. So, the approach does not include a method to resolves functional and non-functional dependencies between patterns to be applied together.

Krleža and Fertalj [30] integrate the concept of patterns into the methodology of model driven architectures (MDA) to assure higher model qualities. They show that patterns can help to purposefully reduce the freedom of modeling in software projects. Patterns are provided for the several abstraction levels of the MDA approach. Further, transformation rules guide users to automatically generate artifacts of more specific levels of the MDA modeling space by considering refinements of a pattern of a more abstract level to a pattern on a more specific level. Thus, relations of patterns in different abstraction levels reduce the number of applicable transformation rules from one level to the other. Further, applicable transformation rules also reduce the number of suitable patterns to be applied on more specific levels, vice versa. So, this design method supports users to build consistent and continuous MDA models covering all abstraction layers. But while patterns and transformation rules are stored to be reused in several use cases, concrete platform specific implementations of patterns are not stored and related to their patterns to be reused directly. The approach also lacks a means to automatically select proper patterns based on criteria, which are defined by a user.

Breitenbücher et al. [31] introduce *Automated Management Idioms* as technology and implementation specific refinements of application management patterns. These idioms can be applied automatically to manage cloud applications by generating declarative descriptions of the management tasks to be executed. Thus, in general they tackle the same issues as Solution Implementations but only for the domain of application management.

Barzen and Leymann [32] show a formalism to collect concrete solution knowledge in the domain of costumes in films in a structured way to derive costume patterns from the captured concrete solutions. They introduce to use domain specific ontologies to define valid properties and values to describe concrete solutions of the domain. Concrete solutions are classified by means of an equivalence function to mine the essence of a set of concrete solutions. The so captured essence in the form of an equivalence class of concrete solutions makes up a pattern. Further, they generalize the approach that it can be applied also in other domains than costumes in films. Their approach clarifies the correspondence of patterns and concrete solutions and emphasizes the approach presented in this work.

Finally, Fehling et al. [33] show how the approach from Barzen and Leymann [32] can be implemented by means of pattern and solution repositories. Further, they show how patterns and concrete solutions can be interrelated comprehensively across both repositories. This is also a concrete implementation of the approach presented in this paper but only for the domain of costumes in films.

IV. Solution Implementations: Building Blocks for Applying and Aggregating Concrete Solutions of Patterns

In the above section, we summarized the state of the art and identified that (i) concrete solutions are not connected to patterns and that (ii) there are no approaches supporting the aggregation of concrete solutions if multiple patterns have to be applied together. Even though there are approaches to derive patterns from concrete solution knowledge iteratively [21][22], concrete solutions are not stored altogether with the actual patterns nor are they linked to them. Concrete solutions, thus, cannot be retrieved from patterns without the need to work them out manually over and over again for the same kind of use cases. Therefore, we propose an approach that (i) defines concrete, implemented solution knowledge as reusable building blocks, (ii) that links these concrete solutions to patterns, and (iii) enables the composition of concrete solutions.

A. Solution Implementations

We argue that concrete solutions are often lost during the pattern writing process since patterns capture general core solution principles in a technology and implementation-agnostic way. In addition, applications of patterns to form new concrete solutions are not documented in a way that enables reusing the knowledge of refinement. As a result, the details of the concrete solutions are abstracted away and must be worked out again when a pattern has to be applied to similar use cases. Thus, the benefits of patterns in the form of abstractions lead to effort when using them due to the missing information of concrete realizations. We suggest keeping concrete solutions linked to patterns in order to ease pattern application and enable implementing new concrete solutions for similar use cases based on existing, already refined, knowledge. These linked solutions can be, for example, (i) the concrete solutions, which were considered initially to abstract the knowledge into a pattern, (ii) later applications of the pattern to build new concrete solutions, or (iii) concrete solutions that were explicitly developed to ease applying the pattern.

Concrete solutions, which we call *Solution Implementations (SI)*, are building blocks of concrete solution knowledge. Therefore, Solution Implementations describe concrete solution knowledge that can be reused directly. In the domain of software development, Solution Implementations provide code, which can be used directly in the development of an own application. For example, a PHP developer faced with the problem to implement the Model-View-Controller Pattern (MVC pattern) [16] in an application can reuse a Solution Implementation of the MVC pattern written in PHP code. Especially, patterns may provide multiple different Solution Implementations – each optimized for a special context and requirements. So, there could be a specific MVC Solution Implementation for PHP4 and another for PHP5, each one considering the programming concepts of the specific PHP version. Another Solution Implementation could provide a concrete solution of the MVC pattern implemented in Java. Therefore, in this case also a Java developer could reuse a concrete MVC solution to save implementation efforts.

By connecting Solution Implementations to patterns, users do not have to redesign and recreate solutions every time a pattern is applied. The introduced Solution Implementations provide a means to capture existing fine-grained knowledge linked to the abstract knowledge provided by patterns. So, users can look at the connected Solution Implementations once a pattern is selected and reuse them directly. To distinguish between pattern's abstract solutions and Solution Implementations, we point out that the solution section of patterns describes the core solution principles in text format and the Solution Implementations represent the real solution objects – which may be in different formats (often depending on the problem domain), e.g., executable code in software development or real clothes in the domain of costumes. Thus, while patterns are documented commonly in natural text, their Solution Implementations depend mainly on the domain of the pattern language and can occur in various forms. Since many specific Solution Implementations can be linked to a pattern, we need a means to select proper Solution Implementations of the pattern to be applied.

B. Selection of Solution Implementations from Patterns

Once a user selects a pattern, he is faced with the problem to decide which Solution Implementation solves his problem in his context properly. To enable selecting

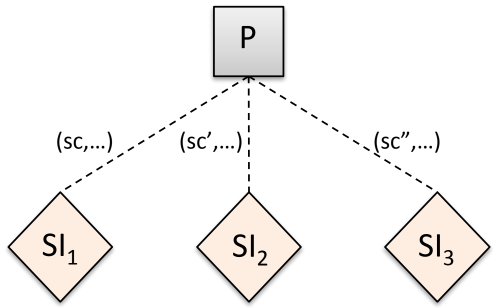


Figure 1. Solution Implementations (SI) connected to a pattern (P) are selectable under consideration of defined Selection Criteria (sc).

proper Solution Implementations of a pattern we introduce *Selection Criteria (sc)*, which determine when to use a certain Solution Implementation. The concept of keeping Solution Implementations linked to the corresponding pattern and supporting the selection of a proper Solution Implementation is shown in Figure 1. Selection Criteria are added to relations between Solution Implementations and patterns. Selection Criteria may be human readable or software interpretable descriptions of when to select a Solution Implementation. They provide a means to guide the selection using additional meta-information not present in the Solution Implementation itself.

To exemplify the concept, we give an example of Solution Implementations from the domain of building architecture. In this domain addressed by Christopher Alexander [2][34], a Solution Implementation would be, for instance, a real entrance of a building or a specific room layout of a real floor, which are described in detail, e.g., by blueprints, and linked to the corresponding pattern [2][34]. To find the most appropriate Solution Implementation for a particular use case, Selection Criteria such as the cost of the architectural Solution Implementation or the used material can be considered. For example, two Solution Implementations for the pattern mentioned above that deals with room layouts might differ in the historical style they are built. Thus, based on such criteria, the refinement of a pattern's abstract solution can be configured by specifying desired requirements and constraints.

To summarize the concept of Solution Implementations it has to be pointed out that solutions in the domain of patterns are abstract descriptions that are agnostic to concrete implementations and written in ordinary text or sketches that illustrate the essential solution principle to support readers. In contrast to this abstract description, we grasp Solution Implementations as concrete solution artifacts, which provide concrete implementation information for particular use cases of a pattern. Solution Implementations are linked to patterns where Selection Criteria are added to the relation between the pattern and the Solution Implementation to guide pattern users during the selection of Solution Implementations.

C. *Aggregation of Solution Implementations*

The concepts of Solution Implementations and Selection Criteria enable to reuse concrete solutions, which are linked to patterns. But most often problems have to be solved by combining multiple patterns. Therefore, we also need a means to combine Solution Implementations of patterns to solve an overall problem altogether. For this purpose, Solution Implementations connected to patterns can have additional interrelations with other Solution Implementations of other patterns affecting their composability. For example, Solution Implementations in the domain of software development are possibly implemented in different programming languages. Therefore, there may exist various Solution Implementations for one pattern in different programming languages, remembering the above example of the PHP and Java Solution Implementations of the MVC pattern. To be combined, both Solution Implementations often have to be implemented in the same programming language.

This leads to the research question "How to compose Solution Implementations selected from multiple patterns into a composed Solution Implementation?"

Patterns are often stored and organized in digital pattern repositories. These repositories, such as presented by Reiners [3], Fehling [24] and van Heesch [25], support users in searching for relevant patterns and navigating through the whole collection of patterns, respectively a pattern language formed by the relations between patterns. To support navigation through pattern languages, these relations can be formulated at the level of patterns indicating that some patterns can be "combined" into working composite solutions, some patterns are "alternatives", some patterns can only be "applied in the context of" other patterns, etc.

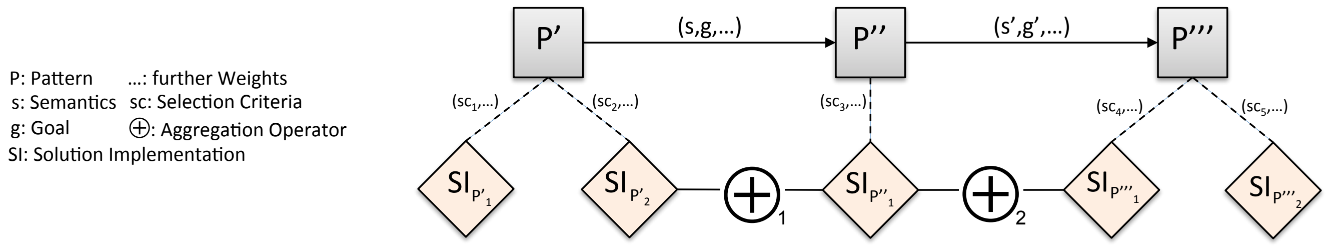


Figure 2. Aggregating Solution Implementations (SI) along the sequence of selected patterns (P).

Zdun [20] has shown that pattern languages can be formalized to enable automated navigation through pattern languages based upon semantic and quality goal constraints reflecting a pattern's effect once it is applied. This also enables combining multiple patterns based on the defined semantics. The approach supports the reader of a pattern language to select proper pattern sequences for solving complex problems that require the application of multiple patterns at once. But, once there are Solution Implementations linked to patterns this leads to the requirement to not only compose patterns but also their concrete Solution Implementations into overall solutions.

We extend the approach of Zdun to solve the problem of selecting appropriate patterns to also select and aggregate appropriate Solution Implementations along the selected sequence of patterns, which is also called solution path.

To assure that Solution Implementations are building blocks composable with each other, we introduce the concept of an *Aggregation Operator,* as depicted in Figure 2. The Aggregation Operator is the connector between several Solution Implementations. It provides the logic to apply two Solution Implementations in combination. Thus, Solution Implementations can just be aggregated if a proper Aggregation Operator implements the necessary adaptations to get two Solution Implementations to work together. Adaptions may be necessary to assure that Solution Implementations match together based on their preconditions and postconditions. Preconditions and postconditions are functional and technical dependencies, which have to be fulfilled for Solution Implementations. In Figure 2, the three patterns P', $\mathrm{P}''$and P''' show a sequence of patterns, which can be selected through the approach of Zdun considering semantics (s) of the relations, goals (g) of the patterns and further weights. Solution Implementations are linked with the patterns and can be selected according to the Selection Criteria introduced in the section above. Furthermore, there are two Solution Implementations associated with pattern P' but only Solution Implementation $\mathrm{SI}_{\mathrm{P}'_2}$ can be aggregated with Solution Implementation $\mathrm{SI}_{\mathrm{P}''_1}$ of the succeeding pattern P'' due to the Aggregation Operator between those two Solution Implementations. There is no Aggregation Operator implemented for $\mathrm{SI}_{\mathrm{P}'_1}$, so that it cannot be aggregated with $\mathrm{SI}_{\mathrm{P}''_1}$, but, nevertheless, it is a working concrete solution of P'. So, in the scenario depicted in Figure 2 an Aggregation Operator has to be available to aggregate $\mathrm{SI}_{\mathrm{P}'_1}$ and $\mathrm{SI}_{\mathrm{P}''_1}$.

In general, Aggregation Operators have to be available to compose Solution Implementations for complex problems requiring the application of multiple patterns. Solution Implementations aggregated with such an operator are concrete implementations of the aggregation of the selected patterns. Aggregated Solution Implementations are, therefore, concrete building blocks solving problems addressed by a pattern language.

Aggregation Operators depend on the connected Solution Implementations, i.e., they are context-dependent due to the context of the Solution Implementations. In contrast to the context section of a pattern, which is used together with the problem section to describe the circumstances when a pattern can be applied, the Solution Implementations' context is more specific in terms of the concrete solution. For example, if an Aggregation Operator shall connect two Solution Implementations consisting of concrete PHP code, the Aggregation Operator itself could also be concrete PHP code wrapping functionality from both Solution Implementations. If the Solution Implementations to aggregate are Java class files, e.g., an Aggregation Operator could resolve their dependencies on other class files or libraries and load all dependencies. Afterwards it could configure the components to properly work together and execute them in a Java runtime. In this case an Aggregation Operator is also a runnable program, which implements the logic to combine Java class files automatically. In other domains like building architecture or costumes in films, where Solution Implementations are not concrete programming code but tangible objects, an Aggregation Operator could provide the logic to combine two Solution Implementations by a description of sequential tasks that have to be performed manually.

Thus, an Aggregation Operator composes and adapts multiple Solution Implementations considering their contexts. However, since Solution Implementations of patterns from varying domains are rather different, they have to be aggregated using specific Aggregation Operators. Because different pattern languages deal with different contexts, they can contain different Aggregation Operators to compose Solution Implementations. The validation section will take a closer look at the Aggregation Operators in different domains.

V. VALIDATION WITH PRACTICAL USE CASES

To validate the concept of Solution Implementations, this section conducts detailed use cases focusing on the application of Solution Implementations in the domains of cloud application architecture, cloud management, costumes in films, user interaction design, and software engineering. These use cases show the practical impact of the presented approach by discussing the application of Solution Implementations, Selection Criteria, and Aggregation Operators in the mentioned domains.

A. Use Case 1: Cloud Application Architecture

General Use Case: Business logic is implemented in a component while instances of the component have to be provisioned and decommissioned based on actual workloads. Provisioning and decommissioning shall be managed by another component.

Concrete Scenario: Solution Implementations provide snippets of Amazon Cloud Formation Templates [35],

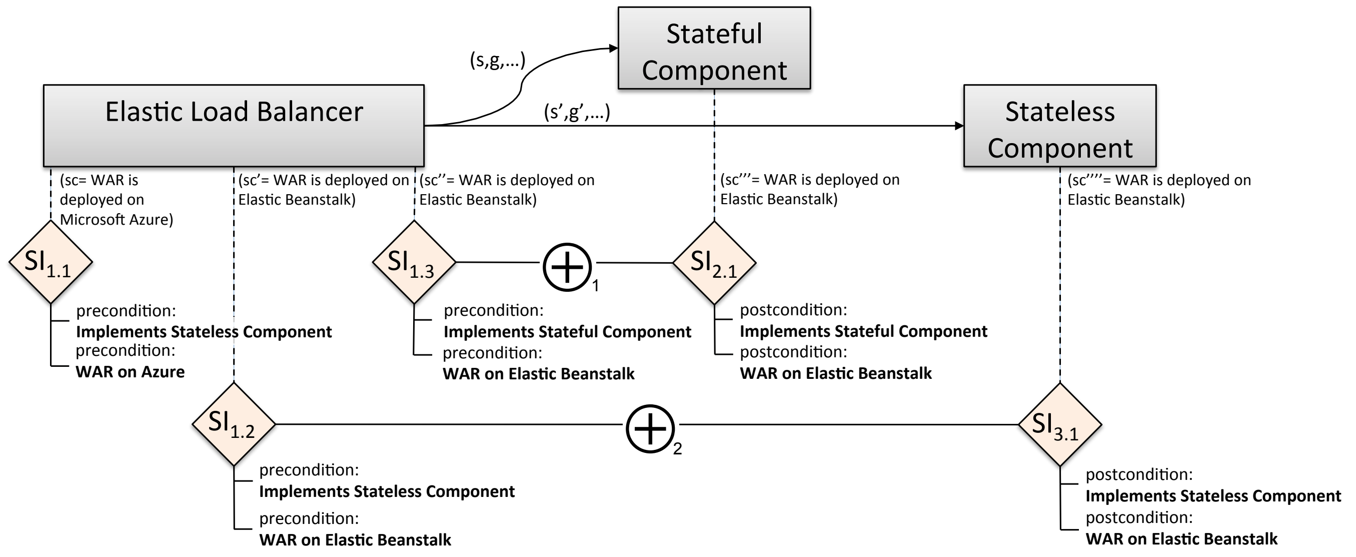

Stateful Component
Elastic Load Balancer
Stateless Component
(s,g,…)
(s',g',…)
(sc= WAR is deployed on Microsoft Azure)
(sc'= WAR is deployed on Elastic Beanstalk)
(sc''= WAR is deployed on Elastic Beanstalk)
(sc'''= WAR is deployed on Elastic Beanstalk)
(sc''''= WAR is deployed on Elastic Beanstalk)
SI1.1
precondition: Implements Stateless Component
precondition: WAR on Azure
SI1.3
precondition: Implements Stateful Component
precondition: WAR on Elastic Beanstalk
1
SI2.1
postcondition: Implements Stateful Component
postcondition: WAR on Elastic Beanstalk
SI1.2
precondition: Implements Stateless Component
precondition: WAR on Elastic Beanstalk
2
SI3.1
postcondition: Implements Stateless Component
postcondition: WAR on Elastic Beanstalk

Figure 3. Solution Implementations in the domain of cloud application architecture linked to patterns and aggregated by Aggregation Operators.

which are manipulated by an Aggregation Operator in order to receive a combined configuration file for Amazon's Cloud.

To explain the concept of Solution Implementations in the domain of cloud computing patterns, the example depicted in Figure 3 shows the three patterns *stateless component*, *stateful component*, and *elastic load balancer* from the pattern language and catalogue of Fehling et al. [17][27]. The stateless component and stateful component patterns describe how an application component can handle state information. They both differentiate between session state – the state with the user interaction within the application and application state – the data handled by the application, for example, customer addresses etc. While the stateful component pattern describes how this state can be handled by the component itself and possibly be replicated among multiple component instances, the stateless component pattern describes how state information is kept externally of the component implementation to be provided with each user request or to be handled in other data storage offerings. The elastic load balancer pattern describes how application components can be scaled out, i.e., how performance is increased or decreased through addition or removal of component instances, respectively. Decisions on how many component instances are required are made by monitoring the amount of requests to the managed components. The elastic load balancer pattern is related to both of the other depicted patterns as it conceptually describes how to scale out stateful components and stateless components: while stateless components can be added and removed rather easily, internal state may have to be extracted from stateful components upon removal or synchronized with new instances upon addition.

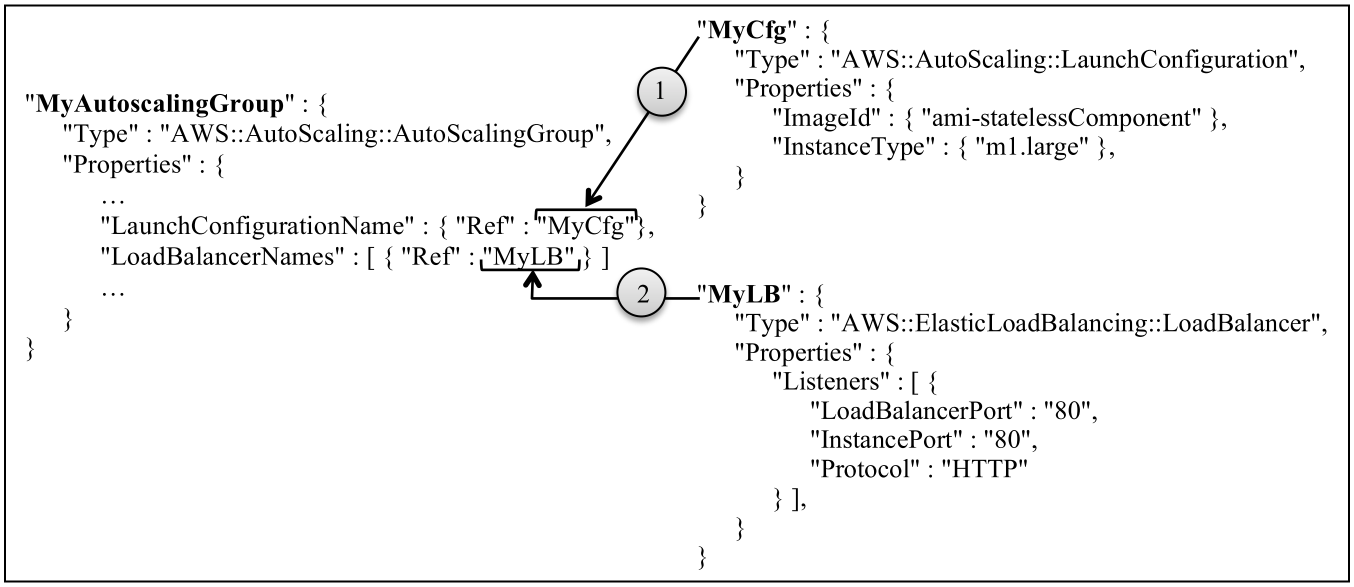
As depicted in Figure 3, the stateless component and stateful component pattern both provide Solution Implementations, which implement these patterns for Java web applications packaged in the web archive (WAR) format that are hosted on Amazon Elastic Beanstalk [36], which is part of Amazon Web Services (AWS) [37]. In this scenario, both Solution Implementations provide a configuration file that describes the provisioning on a certain platform. This configuration file must be adapted by specifying the actual application files to be deployed. The elastic load balancer has three Solution Implementations realizing the described management functionality for stateful components and stateless components for WAR-based applications on Amazon Elastic Beanstalk and Microsoft Azure [38]. The Selection Criteria "WAR is deployed on Microsoft Azure", respectively "WAR is deployed on Elastic Beanstalk" support the user to choose the proper Solution Implementation. For example, if $SI_{1.2}$ is selected, the user knows that this results in a concrete load balancer in the form of a deployed WAR file on Elastic Beanstalk. Since a load balancer scales components, it needs concrete instances of either stateless component or stateful component to work with. Thus, the user can select a proper Solution Implementation for the components based on his concrete requirements considering the Selection Criteria of the relations between the patterns stateless component and stateful component and their Solution Implementations. To ensure that Solution Implementations are composable, i.e., that they properly work together, they refine and enrich the pattern relationships to formulate preconditions, respectively postconditions on the Solution Implementation layer. The preconditions and postconditions of the elastic load balancer Solution Implementations, therefore, capture which related pattern – stateless component or stateful component – they expect to be implemented by managed components. Furthermore, they capture the supported deployment package – WAR in this example – and runtime environment for which they have been developed: $SI_{3.1}$ of stateless component has the postcondition "WAR on Elastic Beanstalk" while $SI_{1.2}$ of elastic load balancer is enriched with the precondition "WAR on Elastic Beanstalk" and $SI_{1.1}$ with "WAR on Azure". The previously introduced Aggregation Operator interprets these dependencies and, for example, composes $SI_{3.1}$ and $SI_{1.2}$. During this task, the configuration parameters of the solutions are adjusted by the operator, i.e., the elastic load balancer is configured with the address of the stateless component to be managed. As some of this information may only become known after the deployment of a component, the configuration may also be handled during the deployment.

In the following, this example is concretely demonstrated by an AWS Cloud Formation template [35] generated by the discussed Aggregation Operator. The template is shown in Listing 1. An AWS Cloud Formation template is a configuration file, readable and processable by the AWS Cloud to automatically provision and configure cloud resources. For the sake of simplicity, the depicted template in Listing 1 shows only the relevant parts, which are adapted by the Aggregation Operator. To run the example scenario on AWS, three parts are needed within the AWS Cloud Formation template to reflect the aggregation of $SI_{3.1}$ and $SI_{1.2}$: (i) an elastic load balancer (MyLB), which is able to scale components, (ii) a launch configuration (MyCfg), which provides configuration parameters about an Amazon Machine Image (AMI) containing the implementation of stateless component as well as a runtime to execute the component in the form of an AWS Elastic Compute Cloud (EC2) [39] instance and, (iii) an autoscaling group (MyAutoscalingGroup) to define scaling parameters used by the elastic load balancer and the wiring of the elastic load balancer and the launch configuration.

MyLB defines an AWS elastic load balancer for scaling Hypertext Transfer Protocol (HTTP) requests on port 80. Further, MyCfg defines the AMI ami-statelessComponent in the property ImageId, which is used for provisioning new instances by an elastic load balancer. The autoscaling group MyAutoscalingGroup wires the stateless component instances and the elastic load balancer at the depicted adaption points one and two by means of referencing the property LaunchConfigurationName to MyCfg and LoadBalancerNames to MyLB, respectively. Since all the mentioned properties are in charge of enabling an elastic load balancer instance to automatically scale and load balance instances of components contained in an AMI, an Aggregation Operator can dynamically adapt those properties based on the selected Solution Implementations to be aggregated. So, presuming that ami-statelessComponent contains an implementation of $SI_{3.1}$, an Aggregation Operator can aggregate $SI_{3.1}$ and $SI_{1.2}$ by adapting the mentioned properties at the depicted adaption points and, therefore, provides an executable configuration template for AWS Cloud Formation.

The same principles can be applied to aggregate $SI_{1.3}$ and $SI_{2.1}$ because of their matching preconditions and postconditions. By adapting the ImageId of the LaunchConfiguration to an AMI, which runs an AWS EC2 instance with a deployed stateful component, the Aggregation Operator can aggregate $SI_{1.3}$ and $SI_{2.1}$.

Further, $SI_{1.1}$ has precondition "WAR on Azure" and is, therefore, incompatible with $SI_{2.1}$ and $SI_{3.1}$, i.e., $SI_{1.1}$ cannot be combined with these Solution Implementations due to their preconditions and postconditions. The selection of a Solution Implementation, therefore, may restrict the number


```
"MyAutoscalingGroup" : {
    "Type" : "AWS::AutoScaling::AutoScalingGroup",
    "Properties" : {
        …
        "LaunchConfigurationName" : { "Ref" : "MyCfg"},
        "LoadBalancerNames" : [ { "Ref" : "MyLB"} ]
        …
    }
}
```
```
"MyCfg" : {
    "Type" : "AWS::AutoScaling::LaunchConfiguration",
    "Properties" : {
        "ImageId" : { "ami-statelessComponent" },
        "InstanceType" : { "m1.large" },
    }
}
```
```
"MyLB" : {
    "Type" : "AWS::ElasticLoadBalancing::LoadBalancer",
    "Properties" : {
        "Listeners" : [ {
            "LoadBalancerPort" : "80",
            "InstancePort" : "80",
            "Protocol" : "HTTP"
        } ],
    }
}
```

Listing 1. Adaption Points configured by an Aggregation Operator in an extract from an AWS Cloud Formation template to aggregate configuration snippets of elastic load balancer and stateless component.

of matching Solution Implementations of the succeeding pattern since postconditions of the first Solution Implementation have to match with preconditions of the second. This way, the space of concrete solutions is reduced based on the resulting constraints of a selected Solution Implementation. To elaborate a solution to an overall problem described by a sequence of patterns exactly one Solution Implementation has to be selected for each pattern in the sequence considering its selection criteria to match non-functional requirements, as well as postconditions of the former Solution Implementation.

B. *Use Case 2: Cloud Application Management*

General Use Case: An application component has to be migrated to a cloud environment and downtime is acceptable during the migration. In the cloud environment, the number of component instances shall be automatically increased and decreased considering workloads.

Concrete Scenario: Solution Implementations provide concrete solutions by means of executable workflow snippets, which are combined by an Aggregation Operator. This aggregated solution in the form of a combined workflow snippet automatically deploys the application on Amazon's Cloud offering Elastic Beanstalk and configures the automated scaling.

In this use case, we show how the presented approach can be applied in the domain of cloud application management. Therefore, we describe how applying *management patterns* introduced in [17][40] to cloud applications can be supported by reusing and aggregating predefined Solution Implementations in the form of executable management workflows.

In the domain of cloud application management, applying the concept of patterns is quite difficult as the refinement of a pattern's abstract solution to an executable management workflow for a certain use case is a complex challenge: (i) mapping abstract conceptual solutions to concrete technologies, (ii) handling the technical complexity of integrating different heterogeneous management APIs of different providers and technologies, (iii) ensuring non-functional cloud properties, (iv) and the mainly remote execution of management tasks lead to immense technical complexity and effort when refining a pattern in this domain. The presented approach of Solution Implementations enables to provide completely refined solutions in the form of executable *management workflows* that already consider all these aspects. Thus, if they are linked with the corresponding pattern, they can be selected and executed directly without further adaptations. This reduces the (i) required management knowledge and (ii) manual effort to apply a management pattern significantly. To apply the concept of Solution Implementations to this domain, two issues must be considered: (i) selection and (ii) aggregation of Solution Implementations in the form of management workflows.

To tackle these issues, we employ the concept of *Management Planlets*, which was introduced in our former research on cloud application management automation [41][42]. Management planlets are *generic management building blocks* in the form of workflows that implement management tasks such as installing a web server, updating an operating system, or creating a database backup.

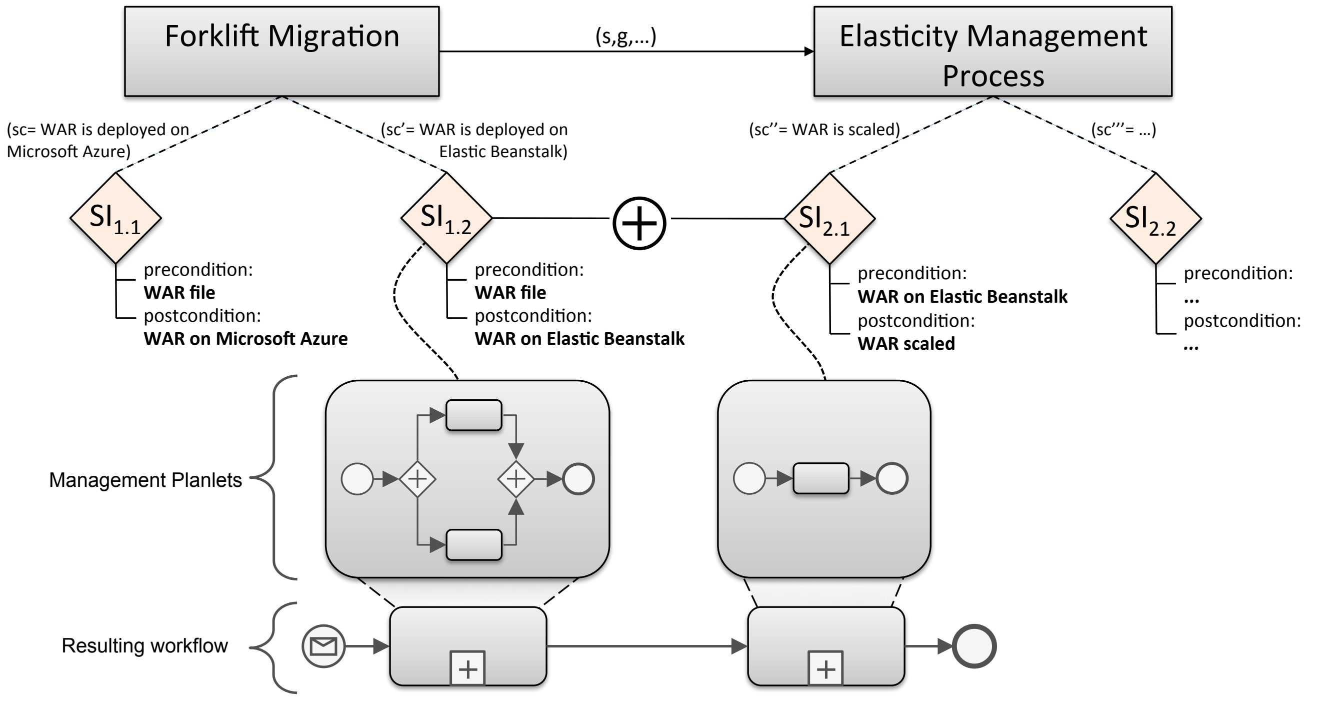

Forklift Migration
(s,g,...)
Elasticity Management Process
(sc= WAR is deployed on Microsoft Azure)
(sc'= WAR is deployed on Elastic Beanstalk)
(sc''= WAR is scaled)
(sc'''= ...)
SI1.1
SI1.2
SI2.1
SI2.2
precondition: WAR file
postcondition: WAR on Microsoft Azure
precondition: WAR file
postcondition: WAR on Elastic Beanstalk
precondition: WAR on Elastic Beanstalk
postcondition: WAR scaled
precondition: ...
postcondition: ...
Management Planlets
Resulting workflow

Figure 4. Management Planlets are Solution Implementations in the domain of cloud management linked to patterns and aggregated by an Aggregation Operator.

Each planlet exposes its functionality through a formal specification of its *effects* on components, i.e., its *postconditions*, and defines optional *preconditions* that must be fulfilled to execute the planlet. Therefore, each specific precondition of a planlet must be fulfilled by postconditions of other planlets. Thus, planlets can be combined to implement a more sophisticated management task, such as migrating an application or its components. If two or more planlets are combined, the result is a *Composite Management Planlet (CMP)*, which can be recursively combined with other planlets again: the CMP inherits all postconditions of the orchestrated planlets and exposes all their preconditions, which are not fulfilled already by the composed planlets. Thus, management planlets provide a recursive aggregation model to implement management workflows. Based on these characteristics, Planlets are ideally suited to implement management patterns in the form of concrete Solution Implementations. We create Solution Implementations that implement a pattern's refinement for a certain use case by orchestrating several Planlets to an overall Composite Management Planlet. This CMP implements the required functionality in a modular fashion as depicted in Figure 4.

As stated above, selection and aggregation of Solution Implementations must be considered, the latter if multiple patterns are applied together. For example, Figure 4 shows two management patterns: (i) *forklift migration* [40] – application functionality is migrated with allowing downtime and (ii) *elasticity management process* [17] – application functionality is scaled based on experienced workload. Both patterns are linked to two Solution Implementations, each in the form of Composite Management Planlets that implement the corresponding management logic as executable workflows. The forklift migration pattern provides two Solution Implementations: one migrates a Java-based web application (packaged as WAR file) to Microsoft Azure [38], another to Amazon Elastic Beanstalk [35]. Thus, if the user selects this pattern and chooses the Selection Criteria defining that a WAR application shall be migrated to Elastic Beanstalk, $SI_{1.2}$ is selected. Whether this Solution Implementation is applicable at all depends on the context: if the application to be migrated is a WAR application, then the Solution Implementation is appropriate and the associated Planlet migrates the WAR application to Beanstalk. Equally to this pattern, the elasticity management process pattern shown in Figure 4 provides two Solution Implementations: one provides executable workflow logic for scaling a WAR application on Elastic Beanstalk ($SI_{2.1}$). In this scenario, the workflow simply configures the automated scaling feature, which is natively supported by Amazon Beanstalk. Thus, if these two patterns are applied together, the selection of $SI_{1.2}$ restricts the possible Solution Implementations of the second pattern, as only $SI_{2.1}$ is applicable (its preconditions match the postconditions of $SI_{1.2}$). As a result, the selection of appropriate Solution Implementations can be reduced to the problem of (i) matching Selection Criteria to postconditions of Solution Implementations and (ii) matching preconditions and postconditions of different Solution Implementations to be combined.

After Solution Implementations of different patterns have been selected, the second issue of aggregation has to be tackled to combine multiple Solution Implementations in the form of workflows into an overall management workflow that incorporates all functionalities. Therefore, we implemented a single Aggregation Operator for this pattern language as described in the following: to combine multiple Solution Implementations, the operator integrates the corresponding workflows as subworkflows [43]. The control flow, which defines the order of the Solution Implementations, i.e., the subworkflows, is determined based on the patterns' solution path depicted in Figure 2. So in general, if a pattern is applied before another pattern, also their corresponding Solution Implementations are applied in this order.

C. Use Case 3: Costumes in Films

General Use Case: An actor or an actress playing the role of a superhero that hides his strength by means of boring clothes in his daily live has to be dressed with several costumes. The superhero needs the ability to easily exchange his every day clothes with the superhero costume.

Concrete Scenario: Solution Implementations are provided by means of concrete costumes, which are manually aggregated into one costume.

In the domain of costumes in films, costume patterns can be defined as a proven solution to the design problem for communicating a certain character such as a sheriff or an outlaw by their clothes [12]. A costume transports a lot of information about a character like character traits, moods and social standing, as well as information on the setting of the film. Costume patterns capture the convention of this communication. Like in the other domains, when working with the costume patterns the costume designer needs to spent significant effort to implement the abstract solution description provided by the pattern for a concrete context. When starting to search for the right costumes needed for a certain film, the patterns are of great help by providing the essence of the convention on how to dress characters like the typical superhero or a shy guy in means of being understood and recognized easily by the spectators. For example, the superhero costume probably contains items of clothes like a cape, tight-fitting pants, and a shirt that emphasize the muscles and allow free movements together with a unique logo of this hero. The shy guy, on the other hand, is mostly communicated by a costume of rather pale colors and is dressed in a slightly too big modest suit hiding his face behind big glasses. As this solution is rather abstract, it needs refinement when being applied.

Therefore, in our approach, we suggest the concept of Solution Implementations for connecting the patterns with concrete solutions, meaning descriptions of concrete costumes occurring in films. Since the real tangible costumes are hardly ever kept and stored after the production of a film and since the communicative effect of a costume is retained in films the Solution Implementations are the costumes seen on screen. The descriptions to capture the Solution

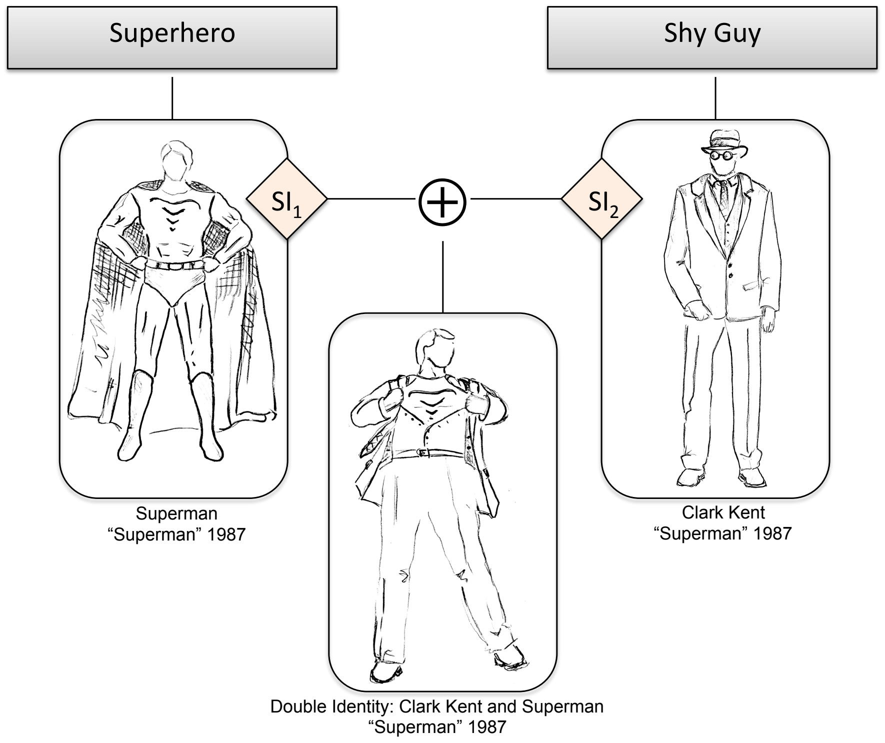


Figure 5. Concrete costumes occurring in the film "Superman" (1987) as Solution Implementations (SI_1, SI_2) of the costume patterns Superhero and Shy Guy are aggregated by an Aggregation Operator.

Implementations contains detailed information on the items of clothes, their material and color, a collection of pictures of the costume as well as contextual information like character traits of the role or its stereotype [32]. Such Solution Implementations can be stored in a Solution Implementation repository [33].

Figure 5 illustrates how the *superhero* pattern, for example, can be connected to the concrete Solution Implementation that the character "Superman" wears in the movie "Superman" (Director: Richard Donner, 1978) or how the *shy guy* pattern can point to the costumes of the character "Clark Kent" in the same movie. But next to the Solution Implementation of the Superman costume, various other Solution Implementations could be connected to the pattern "Superhero" like the Batman or Spiderman costume. Since every pattern can be connected to various Solution Implementations, it is necessary to select the suitable Solution Implementation for the right context. To support finding the right Solution Implementation, the introduced concepts of Selection Criteria as well as defining the pre- and postcondition of the Solution Implementation is also adaptive in the domain of costumes. To find suitable Solution Implementations, i.e., concrete costumes for a concrete film, the Selection Criteria as well as the defined pre- and postcondition of the Solution Implementation can ensure that the costume makes sense in a certain scene. For example, if the Shy Guy pattern shall be applied for Clark Kent in a cold winter scene, other costumes must be taken than if the pattern has to be applied for a scene in summer.

While the concepts of the Solution Implementations, the Selection Criteria, and defined pre- and postconditions are very promising in the domain of costumes, the concept of Aggregation Operators is not always needed: when using a costume pattern to find the right costume, the application of this pattern usually needs just one Solution Implementation and in difference to fragments of code, they are mostly connected together by the storyline and only seldom in a physical way. Nonetheless, there are some situations were *physical* Aggregation Operators are needed. For example, when multiple costume patterns are applied together to one character at once, the corresponding Solution Implementations also need an aggregation and, therefore, need a physical Aggregation Operator. Figure 5 depicts how in the film "Superman" the Solution Implementations of the superhero pattern (SI_1: Superman) and a Solution Implementation of the shy guy pattern (SI_2: Clark Kent) are aggregated together using the Aggregation Operator to build a costume that contains both characters and allows the transformations from one to the other (we omitted Selection Criteria for the sake of simplicity). The necessary adaptations to get those two Solution Implementations to work together would contain actions like making sure that the costume on top needs to be a bit bigger to hide the other, where to store the cape so it is not seen, and how to modify the suite so it does not get torn when being ripped off, for example. We also point out, that in this case, the concept of the Aggregation Operator cannot be automated because the adaption of the costumes in order to fit together has to be done manually by a costume designer.

D. Use Case 4: User Interaction Design

General Use Case: Users need the ability to sign up for accounts of a website. Thus, the users need to provide a password and the sign up process shall only start if the strength of the entered password is validated as strong enough. If a user enters a weak password, he has to be notified that the password needs to be improved.

Concrete Scenario: Solution Implementations provide concrete HyperText Markup Language (HTML) and JavaScript snippets used for designing user interfaces. The final user interface is constructed by aggregating a sequence of Solution Implementations by manipulating associated HTML code.

Patterns are a well-known concept in the domain of user interaction design. A broad number of publications exist that introduce patterns for good user interface designs and user interaction concepts [5][44 - 47]. This use case shows how the approach of Solution Implementations is applied in this domain and, especially, how Solution Implementations from a series of four patterns are aggregated into one combined concrete solution.

As designing user dialogs on websites is a very common issue, many patterns are published that deal with the problem how to design and arrange control elements on a website. Nevertheless, it is still a time consuming effort for a web designer to implement the solution concepts provided by patterns – especially if the concrete website needs to combine several patterns in order to design a complex web interface for users. This is due to the manifold of possible concrete solutions because of the vast number of available technologies to implement websites and control structures. To mention some common technologies today, there are PHP [13], HTML [48], JavaScript [49], Java Servlets [50], JavaServer Pages [51], JavaServer Faces [52], Angular.js [53], jQuery [54], Spring [55], Ruby on Rails [56], Google Web Toolkit [57] and many more. Although websites are rendered using HTML, the different technologies often employ specific concepts to implement a user interface. Unfortunately, this is mostly not plain HTML but a complex combination of server side logic and JavaScript libraries on the browser. In addition, some technologies employ technology-specific constructs and domain-specific languages on server side to specify the control elements of a user interface, which is then transformed into HTML code and the corresponding JavaScript libraries. This means that a developer has to be familiar with language-specific constructs and concepts, complex libraries, and how to combine them in order to refine a pattern's conceptual solution to a concrete implementation. As a result, implementations have to be redeveloped for every technology and use case leading to huge manual efforts.

In the following, we investigate this statement in more detail and assume that a web designer has to implement a website where users can sign up an account by entering a user name and a password. The sign up process shall only start if a safe password is entered (for the sake of simplicity we omit the second password field, which is usually provided for reentering the password to ensure that a user keys in the right password). Therefore, the website has to indicate the strength of the currently entered password. Further, if the user tries to sign up with a weak password, the website should notice him or her about the necessity of a stronger password. This is a very common use case since almost every web shop in the World Wide Web provides such functionality in order to store user specific configurations of the site or the user data for delivery, payment, and invoicing.

In order to realize a website to create accounts, a web designer can use user interaction patterns from [5]. Patterns that are appropriate for the mentioned use case are depicted in Figure 6: *registration, password field, password strength meter* and, finally, *input error message*. The registration pattern describes that a registration form needs control elements to input a user name and a password as well as a button to submit the sign up request. The password field pattern describes that input fields for passwords should not show the password in plaintext. Nevertheless, they should

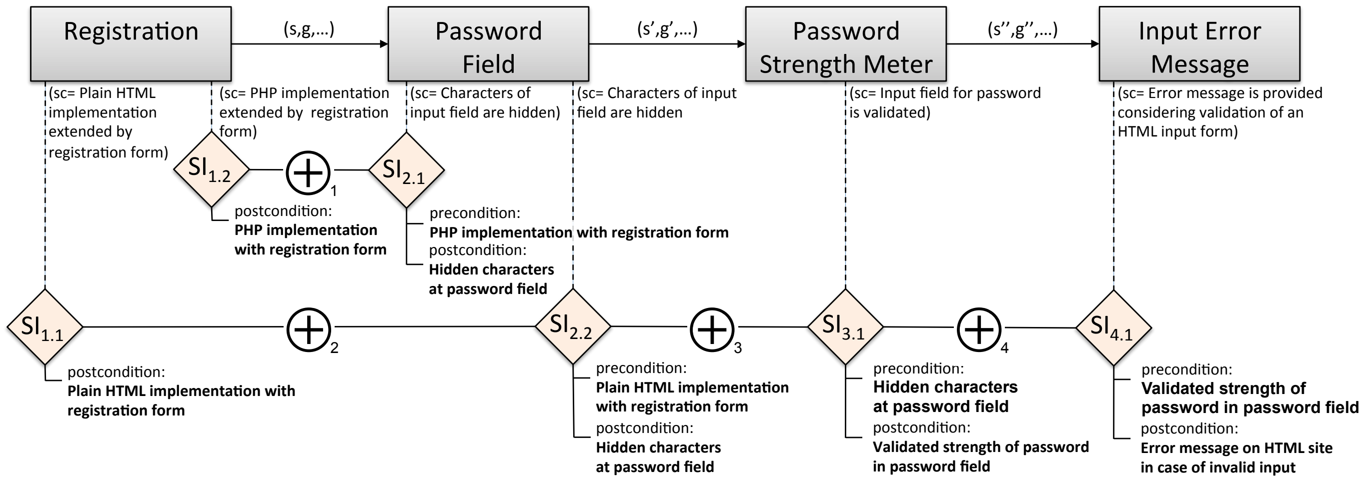

Registration
(s,g,...)
Password Field
(s',g',...)
Password Strength Meter
(s'',g'',...)
Input Error Message
(sc= Plain HTML implementation extended by registration form)
(sc= PHP implementation extended by registration form)
(sc= Characters of input field are hidden)
(sc= Characters of input field are hidden
(sc= Input field for password is validated)
(sc= Error message is provided considering validation of an HTML input form)
SI1.2
1
SI2.1
postcondition: PHP implementation with registration form
precondition: PHP implementation with registration form
postcondition: Hidden characters at password field
SI1.1
2
SI2.2
3
SI3.1
4
SI4.1
postcondition: Plain HTML implementation with registration form
precondition: Plain HTML implementation with registration form
postcondition: Hidden characters at password field
precondition: Hidden characters at password field
postcondition: Validated strength of password in password field
precondition: Validated strength of password in password field
postcondition: Error message on HTML site in case of invalid input

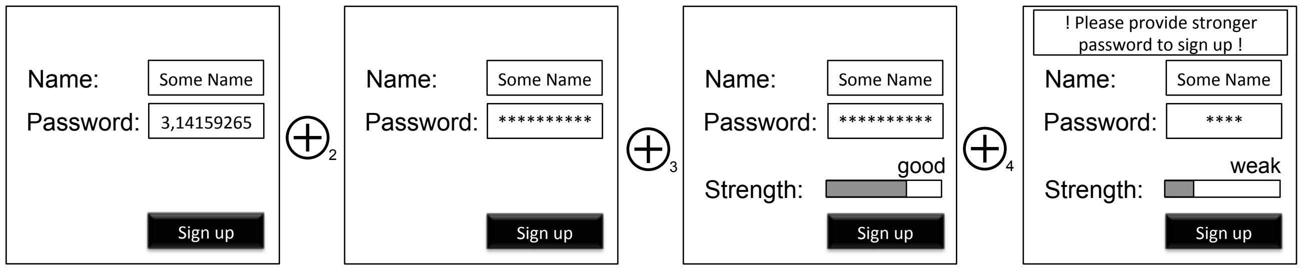
Figure 6. Solution Path of four User Interaction Patterns with related Solution Implementations, which are aggregated by Aggregation Operators.

indicate to the user how many characters have been entered. Further, the password strength meter pattern describes how the strength of a password, i.e., if it is secure or not, can be validated and how a user can be notified about the strength. Finally, the input error message pattern provides a solution how to notify a user about invalid inputs in input fields. It also defines that the website should inform which input field contains the invalid data.

In order to create a concrete solution based on the selected patterns, Solution Implementations have to be selected from all patterns of the solution path. The registration pattern and the password field pattern provide Solution Implementations that extend a plain HTML website (SI$_{1.1}$) or a website coded in PHP (SI$_{1.2}$) as indicated by the Selection Criteria "Plain HTML implementation extended by registration form" and "PHP implementation extended by registration form", respectively. Since the password field shall be protected to avoid unintended discoveries of entered passwords by viewers, either SI$_{2.1}$ or SI$_{2.2}$ have to be combined with SI$_{1.1}$ or SI$_{1.2}$. This is possible since pre- and postconditions of both pairs of Solution Implementations match and Aggregation Operator 1 exists to combine SI$_{1.1}$ with SI$_{2.2}$ as well as Aggregation Operator 2 for SI$_{1.2}$ and SI$_{2.1}$. Since for the following patterns of the solution path – password strength meter and input error message – no Solution Implementations are available in the example depicted in Figure 6, which can be combined with the PHP alternative of password field, we assume that SI$_{1.1}$ and SI$_{2.2}$ are selected. Therefore, SI$_{3.1}$ and SI$_{4.1}$ are also selected because also Aggregation Operators exist to combine them with the previous Solution Implementations along the solution path.

In order to investigate how the Aggregation Operators manipulate the plain HTML file, all aggregations along the solution path are depicted in Figure 7 from left to right. On the left side of this figure, the user interface is illustrated as provided by SI$_{1.1}$. The user interface contains two input fields with their labels "Name" and "Password" as well as a button to submit the sign up request. The password in the second input field still shows the entered characters in plain text. Beneath the sketched user interface, an excerpt of the HTML code provided by the Solution Implementation is shown. The bold letters indicate the registration form with its control elements. After Aggregation Operator 2 has combined SI$_{1.1}$ and SI$_{2.2}$, the input field for the password is manipulated to hide entered characters and only show how many characters are keyed in by means of stars. In plain HTML, this can be achieved by changing the type of the input field from text to password as depicted in the second code snippet in bold letters. Thus, the Aggregation Operator configures the type of the existing input field.

The password strength meter provided by SI$_{3.1}$ extends the HTML file by validation logic implemented in an additional JavaScript file. Besides the logic to determine if an entered password is secure or not, the JavaScript file also contains code to display the strength meter by means of a bar and a label. The more the bar is filled, the more secure the entered password is. To integrate this functionality, Aggregation Operator 3 manipulates the HTML file so that the JavaScript file is loaded, as depicted with the top bold letters in the third code snippet from left. The bottom bold letters in this code snippet shows that the password strength meter is placed between the password field and the submit button as illustrated in the sketch upon the code snippet. To wire the password strength meter with the password input field, the Aggregation Operator has to be configured in order to parameterize the password strength meter with the id of the password input field. The resulting HTML file can be modified by the web designer manually, if the position of the password strength meter does not suit the needs of the



```
<html>
...
<form action="reg.html">
<input id="name" type="text">
<input id="pw" type="text">
<input type="submit"
        value="Sign up">
</form>
...
</html>
```

```
<html>
...
<form action="reg.html">
<input id="name" type="text">
<input id="pw" type="password">
<input type="submit"
        value="Sign up">
</form>
...
</html>
```

```
<html>
...
<script src="strengthMtr.js">
...
<form action="reg.html">
<input id="name" type="text">
<input id="pw" type="password">
<strengthMeter validate="pw">
<input type="submit"
        value="Sign up">
</form>
...
</html>
```

```
<html>
...
<script src="strengthMtr.js">
<script src="inputErrMsg.js">
...
<form action="reg.html">
<inputErrMessage>
<input id="name" type="text">
<input id="pw" type="password">
<strengthMeter validate="pw">
<input type="submit"
        value="Sign up">
</form>
...
</html>
```

Figure 7. Aggregation Operators combine Solution Implementations by adapting HTML code.

website structure etc.

Finally, also $SI_{4.1}$, which provides logic to show input error messages, is combined into the HTML file by means of Aggregation Operator 4. As with Aggregation Operator 3, the HTML code is adapted to load an additional JavaScript file, which contains the code of the input error message field as depicted with the top bold letters in the HTML snippet far right in Figure 7. Further, the visualization of the input error message field is put into the form so that it can show the validation results of the input fields. Of course, also this must be configured manually as only the web designer knows which error messages shall be displayed.

This use case shows that the concept of Solution Implementations can help to implement concrete solutions of several patterns together. Since user interfaces mostly incorporate many control elements, Solution Implementations can lead to immense reduction of effort in contrast to combine them manually. Especially if a developer has to deal with a vast of different technologies as mentioned above and, therefore, many specific implementation concepts for each of these technologies, Solution Implementations can provide a means to easily reuse available solutions for new use cases. Nevertheless, user interface design is often an act of creativity so that standardized implementations, as provided by Solution Implementations and Aggregation Operators, need to be adapted. But also in such cases, the presented concept can provide starting points with runnable code that then can be adapted creatively to meet the challenges of a non-standard user interface.

E. Use Case 5: Object-Oriented Software Engineering

General Use Case: A software engineer needs to combine an implementation of the Model View Controller Pattern with user interface patterns.

Concrete Scenario: An Aggregation Operator combines Solution Implementations of the Model View Controller pattern and the Pulldown Button Pattern in the form of Java classes. So, Solution Implementations from different pattern domains, i.e., different pattern languages are aggregated by means of an Aggregation Operators by adapting Java code.

When developing software systems, it is a common practice to first design the architecture of the software. In the architecture phase, design decisions are made, which are on an abstracter level in contrast to the concrete implementation problems, because they deal with general questions about the structure of software. In the domain of software architecture, patterns are a pervasive means to discuss design decisions and to describe the architecture of software systems [7]. They often affect later implementations, since the abstract structure of the software has to be implemented by concepts of the used technology. If Solution Implementations are provided for such patterns, the application of these patterns can be eased in order to save efforts to work them out manually for new use cases.

As already mentioned in the former use cases, patterns are also very common in the domain of user interaction design. Especially patterns describing control elements of user interfaces are often used. Thus, such patterns deal with problems that are very close to concrete implementations, since they often provide sketches that show how control elements should look like and how they should be arranged on a user interface [5].

This last use case shows how Solution Implementations of patterns from the two different domains of object-oriented design and user interaction design can be combined using our concept of Solution Implementations. Therefore, we show how an Aggregation Operator composes Solution Implementations of the pattern *Model View Controller (MVC)* [16], which is from the domain of object-oriented software architecture, and the *Pulldown Button* pattern [5], which is from the domain of user interaction design. The MVC pattern describes how the user interface of a program can be separated from its domain logic in order to prevent that changes of the user interface affect the implementation of the domain logic. Therefore, the user interface is encapsulated into a view entity, while the domain logic is provided by a model entity. The controller receives user interactions and triggers processing of domain logic based on the user's inputs. The pulldown button pattern provides a means to select exactly one value from a list of values. This list is only shown when a user clicks on the control element. If he or she selects a value from the list, the list is hidden again and only the selected value is visible.

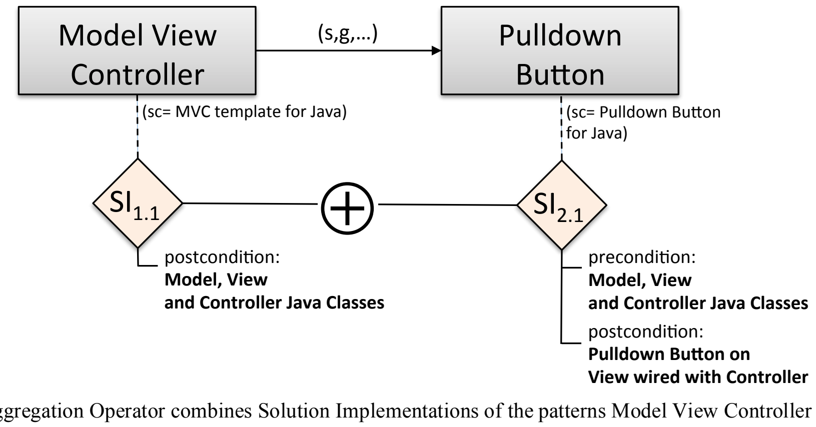
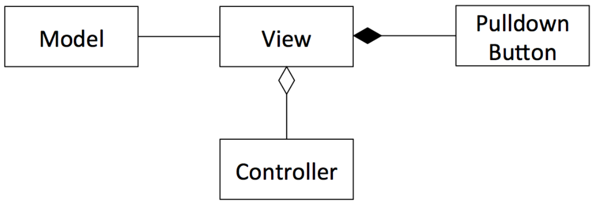

Model View Controller
(s,g,…)
Pulldown Button
(sc= MVC template for Java)
(sc= Pulldown Button for Java)
SI1.1
SI2.1
postcondition: Model, View and Controller Java Classes
precondition: Model, View and Controller Java Classes
postcondition: Pulldown Button on View wired with Controller

Figure 8. An Aggregation Operator combines Solution Implementations of the patterns Model View Controller and Pulldown Button.



Adaptions in View Class

```
public class myView extends JFrame{
     ...
     //define pulldown button
     JComboBox b;
     ...
     public myView(){
          ...
          //add pulldown button to view
          b = new JComboBox(valueList);
          this.add(b);
          ...
     }
     ...
}
```

Figure 9. Aggregated Solution Implementations of MVC and Pulldown Button in UML as well as adaptions of Java code by the Aggregation Operator.

Both patterns are depicted in Figure 8. For the sake of simplicity, there is just one Solution Implementation provided for each pattern – $SI_{1.1}$ and $SI_{2.1}$. Both Solution Implementations provide concrete solutions in the form of Java code as illustrated by the corresponding Selection Criteria. The postcondition of $SI_{1.1}$ "Model, View and Controler Java Classes" shows that this Solution Implementation consists of Java classes that implement the MVC paradigm. Further, the precondition of $SI_{2.1}$ matches the mentioned postconditions of $SI_{1.1}$, so both can be aggregated to form a combined solution.

The aggregation of both Solution Implementations is depicted as a Unified Modeling Language (UML) class diagram in Figure 9 [58]. The figure shows on the left that the pulldown button class is associated with the view class of the MVC Solution Implementation $SI_{1.1}$. To achieve this aggregation, the Aggregation Operator manipulates the java code of the view class so that an instance of the pulldown button is created and shown when the view is launched. Bold letters on the right in Figure 9 highlight the adaptions of the java code. So, this use case shows that the concept of Solution Implementations also allows combining solution knowledge from different pattern domains, since MVC is categorized as an architectural pattern, while pulldown button is a pattern from user interaction design. As they appear in different pattern languages, this use case demonstrates that Solution Implementations of patterns originally provided by different pattern languages can be applied together based on the presented approach. Of course, the aggregation must be adapted manually to place the pulldown button at the desired position and to select the appropriate view and so on. However, the actual aggregation, i.e., copying the corresponding java code, defining the required Java libraries, and linking the affected classes can be done by an Aggregation Operator automatically – and this already eases applying those patterns together in reality.

VI. Prototypes

To prove the approach's technical feasibility, we implemented a pattern repository prototype that aims to capture patterns and their cross-references in a domain-independent way to support working with patterns [33][59]. Based on semantic wiki-technology, it enables capturing, management, and search of patterns. To adapt to different pattern domains, the pattern format is freely configurable. The pattern repository already contains various patterns from different domains such as cloud computing patterns [17], cloud data patterns [60], and costume patterns [12] to demonstrate the generic flexibility of our approach. The cross-references between the patterns enable an easy navigation through the pattern languages. Links like "apply after" or "combined with" connect the patterns, which results in a pattern language. The pattern repository does not only contain the patterns and their cross-references, but can be connected to a second repository containing Solution Implementations. We realized a Solution Implementation repository [33][61] for the domain of costume patterns to prove the interoperability of these two kinds of repositories. Here, for example, the concrete costumes of a sheriff occurring in a film are represented as the Solution Implementation of a sheriff costume pattern. By connecting the pattern to a Solution Implementation as a concrete solution of the abstracted solution of the pattern, the application of the pattern in a certain context is facilitated. Although the implemented solution repository for costumes in films is specifically tailored to store Solution Implementations from this domain, the concept of combining pattern repositories and solution repositories as described in [33] can easily be reused to create repositories for the other use cases to store code, HTML files, Cloud Formation Templates, or workflows.

To test the concept of Aggregation Operators, we prototyped the combination of several concrete Solution Implementations in the domain of cloud management patterns (use case 2). This domain is very appropriate, as the aggregation can be automated completely: we employed our workflow generator [41] to automatically combine different Management Planlets to an overall workflow implementing a solution to a problem that requires the use of multiple patterns. The input for this generator is a partial order of (composite) management planlets, i.e., Solution Implementations that have to be orchestrated into an executable workflow. This partial order is determined by the relations of combined patterns: if one pattern is applied after another pattern, also their Solution Implementations, i.e., Management Planlets, have to be executed in this order. The workflow generator creates BPEL-workflows while Management Planlets are also implemented using BPEL. As BPEL is a standardized workflow language, the resulting management plans are portable across different engines and cloud environments supporting BPEL as workflow language, which is in line with the TOSCA standard [62][63][64]. Thus, this prototype shows that in certain domains, Aggregation Operators can be realized in an automated fashion. However, as seen in costumes, this is not always the case and in many other domains manual effort has to be spent for the aggregation.

VII. Conclusion and Future Work

In this paper, we introduced the concept of Solution Implementations as concrete instances of a pattern's solution. We showed how Solution Implementations can enrich patterns and pattern languages and how this approach can be integrated into a pattern repository. To derive concrete solutions for problems that require the application of several patterns we proposed a mechanism to compose these solutions from concrete solutions of the required patterns by means of Aggregation Operators. We concretized the general concept of Solution Implementations by five detailed use cases in the domains of cloud application architecture, cloud management, costumes in films, user interaction design and software engineering. We partially verified the approach by means of a prototype of an integrated pattern repository.

Currently, we extend the implemented repository for solution knowledge in the domain of costume design to capture Solution Implementations more efficiently. This repository integrates patterns and linked Solution Implementations in this domain and we enlarge the amount of costume Solution Implementations. We are also going to extend the presented approach to not only work on Solution Implementation sequences but also on aggregations of concrete solution instances not ordered temporally due to pattern sequences of a solution path. Since Solution Implementations are composed by Aggregation Operators, we are going to enhance our pattern repositories to also store and manage the Aggregation Operators. Finally, we will investigate Aggregation Operators in domains besides the above mentioned to formulate a general theory of Solution Implementations and Aggregation Operators.

Acknowledgment

This work was partially funded by the Co.M.B. project of the DFG under the promotional reference LE 2275/5-1.

References

[1] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "From pattern languages to solution implementations," Proceedings of the Sixth International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 12–21, May 2014.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, "A pattern language: towns, buildings, constructions," Oxford University Press, 1977.

[3] T. Iba and T. Miyake, "Learning patterns: a pattern language for creative learners II," Proceedings of the 1st Asian Conference on Pattern Languages of Programs (AsianPLoP 2010), pp. I-41 – I-58, March 2010.

[4] F. Salustri, "Using pattern languages in design engineering," Proceedings of the International Conference on Engineering Design, pp. 248–362, August 2005.

[5] M. van Welie, A pattern library for interaction design, http://www.welie.com, last accessed on 2014.11.28.

[6] R. Reiners, Bridge Pattern Library, http://bridge-pattern-library.fit.fraunhofer.de/pattern-library/, last accessed on 2014.11.28.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-oriented software architecture, volume 1: a system of patterns," Wiley, 1996.

[8] M. Fowler, "Patterns of enterprise application architecture," Addison-Wesley, 2003.

[9] T. Brunner and A. Zimmermann, "Pattern-oriented enterprise architecture management," Proceedings of the Fourth International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 51–56, July 2012.

[10] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, "An architectural pattern language of cloud-based applications," Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP), pp. A-20–A-30, October 2011.

[11] J. Yoder and J. Barcalow, "Architectural Patterns for Enabling Application Security," Pattern Languages of Program Design 4, pp. 301–336, 2000.

[12] D. Schumm, J. Barzen, F. Leymann, and L. Ellrich, "A pattern language for costumes in films," Proceedings of the 17th European Conference on Pattern Languages of Programs (EuroPLoP), pp. C4-1–C4-30, July 2012.

[13] PHP, PHP: Hypertext Preprocessor, http://php.net, last accessed on 2014.11.28.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Addison-Wesley, 1995.

[15] G. Hohpe and B. Wolf, "Enterprise integration patterns: designing, building, and deploying," Addison-Wesley, 2004.

[16] T. Reenskaug, "The original MVC reports," https://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf, last accessed on 2014.11.28.

[17] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, "Cloud computing patterns," Springer, 2014.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-oriented software architecture volume 1: a system of patterns," Wiley, 1996.

[19] R. Reiners, R. Halvorsrud, A. Wegner Eide, and D. Pohl, "An approach to evolutionary design pattern engineering," Proceedings of the 19th international Conference on Pattern Languages of Programs, October 2012, scheduled for 2014.

[20] U. Zdun, "Systematic pattern selection using pattern language grammars and design space analysis," Software: Practice and Experience, vol. 37, pp. 983–1016, 2007.

[21] R. Reiners, "A pattern evolution process – from ideas to patterns," Lecture Notes in Informatics – Informatiktage 2012, pp. 115–118, March 2012.

[22] M. Falkenthal, D. Jugel, A. Zimmermann, R. Reiners, W. Reimann, and M. Pretz, "Maturity assessments of service-oriented enterprise architectures with iterative pattern refinement," Lecture Notes in Informatics - Informatik 2012, pp. 1095–1101, September 2012.

[23] R. Porter, J. O. Coplien, and T. Winn, "Sequences as a basis for pattern language composition," in Science of Computer Programming, Special issue on new software composition concepts, vol. 56, pp. 231–249, April 2005.

[24] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck, "A collection of patterns for cloud types, cloud service models, and cloud-based application architectures," http://www.cloudcomputingpatterns.org, last accessed on 2014.11.28, University of Stuttgart, Report 2011/05, Mai 2011.

[25] U. van Heesch, Open Pattern Repository, https://code.google.com/p/openpatternrepository/, last accessed on 2014.11.28.

[26] M. Demirköprü, “A new cloud data pattern language to support the migration of the data layer to the cloud,” in German “Eine neue Cloud-Data-Pattern-Sprache zur Unterstützung der Migration der Datenschicht in die Cloud,” University of Stuttgart, diploma thesis no. 3474, 2013.

[27] C. Fehling, F. Leymann, J. Rütschlin, and D. Schumm, “Pattern-based development and management of cloud applications,” Future Internet, vol. 4, pp. 110–141, 2012.

[28] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, “An architectural pattern language of cloud-based applications,” Proceesings of the 18th Conference on Pattern Languages of Programs (PLoP), pp. A-20 – A-21, Oct. 2011.

[29] A. G. Mirnig and M. Tscheligi, “Building a general pattern framework via set theory: towards a universal pattern approach,” Proceedings of the Sixth International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 8–11, May 2014.

[30] D. Krleža and K. Fertalj, “A method for situational and guided information system design,” Proceedings of the Sixth International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 70–78, May 2014.

[31] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Automating cloud application management using management idioms,” Proceedings of the Sixth International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 60–69, May 2014.

[32] J. Barzen and F. Leymann, “Costume languages as pattern languages,“ accepted at Pursuit of Pattern Languages for Societal Change, unpublished.

[33] C. Fehling, J. Barzen, M. Falkenthal, and F. Leymann, “PatternPedia - collaborative pattern identification and authoring,“ accepted at Pursuit of Pattern Languages for Societal Change, unpublished.

[34] C. Alexander, “The timeless way of building,” Oxford University Press, 1979.

[35] Amazon, AWS Cloud Formation, http://aws.amazon.com/cloudformation/, last accessed on 2014.11.28.

[36] Amazon, Elastic Beanstalk, http://aws.amazon.com/elasticbeanstalk/, last accessed on 2014.11.28.

[37] Amazon, Amazon Web Services, http://aws.amazon.com, last accessed on 2014.11.28.

[38] Microsoft, Microsoft Azure, http://azure.microsoft.com, last accessed on 2014.11.28.

[39] Amazon, AWS EC2, http://aws.amazon.com/ec2/, last accessed on 2014.11.28.

[40] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, and S. Verclas “Service migration patterns – decision support and best practices for the migration of existing service-based applications to cloud environments,” Proceedings of the IEEE International Conference on Service Oriented Computing and Applications (SOCA), in press, December 2013.

[41] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Pattern-based runtime management of composite cloud applications,” Proceedings of the 3rd International Conference on Cloud Computing and Service Science (CLOSER), pp. 475–482, May 2013.

[42] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and M. Wieland, “Policy-Aware Provisioning of Cloud Applications,” in SECURWARE. Xpert Publishing Services, August 2013, pp. 86–95.

[43] O. Kopp, H. Eberle, and F. Leymann, “The subprocess spectrum,” Proceedings of the 3rd Business Process and Services Computing Conference (BPSC), pp. 267–279, September 2010.

[44] J. Tidwell, “Designing interfaces – patterns for effective interaction design,” O’Reilly, 2011.

[45] D. K. van Duyne, J. A. Landay, and J. Hong, “The design of sites: patterns for creating winning websites,” Prentice Hall, 2007.

[46] J. Borchers, “A pattern approach to interaction design,” John Wiley & Sons, 2001.

[47] Yahoo Developer Network, Yahoo design pattern library, https://developer.yahoo.com/ypatterns/, last accessed on 2014.11.28.

[48] World Wide Web Consortium, HTML 4.01 Specification, http://www.w3.org/TR/html401/, last accessed on 2014.11.28.

[49] Ecma International, ECMAScript Language Specification, http://www.ecma-international.org/ecma-262/5.1/, last accessed on 2014.11.28.

[50] Oracle, Java Servlet Technology, http://www.oracle.com/technetwork/java/index-jsp-135475.html, last accessed on 2014.11.28.

[51] Oracle, JavaServer Pages Technology, http://www.oracle.com/technetwork/java/javaee/jsp/index.html, last accessed on 2014.11.28.

[52] Oracle, JavaServer Faces Technology, http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html, last accessed on 2014.11.28.

[53] Google, Angular.js, https://angularjs.org, last accessed on 2014.11.28.

[54] jQuery, jQuery, http://jquery.com, last accessed on 2014.11.28.

[55] Spring, Spring Framework, http://projects.spring.io/spring-framework/, last accessed on 2014.11.28.

[56] Rails, Ruby on Rails, http://rubyonrails.org, last accessed on 2014.11.28.

[57] Google, Google Web Toolkit, http://www.gwtproject.org, last accessed on 2014.11.28.

[58] Object Management Group, Unified Modeling Language, http://www.uml.org, last accessed on 2014.11.28.

[59] N. Fürst, “Semantic wiki for capturing design patterns,” in German “Semantisches Wiki zur Erfassung von Design-Patterns,” University of Stuttgart, diploma thesis no. 3527, 2013.

[60] S. Strauch, V. Andrikopoulos, U. Breitenbücher, S. Gómez Sáez, O. Kopp, and F. Leymann, “Using patterns to move the application data layer to the cloud,” Proceedings of the 5th International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 26–33, May 2013.

[61] D. Kaupp, “Application of semantic wikis for solution documentation and pattern identification,” in German “Verwendung von semantischen Wikis zur Lösungsdokumentation und Musteridentifikation,” University of Stuttgart, diploma thesis no. 3406, 2013.

[62] OASIS, Topology and Orchestration Specification for Cloud Applications Version 1.0, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, last accessed on 2014.11.28.

[63] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, “TOSCA: portable automated deployment and management of cloud applications,” in Advanced Webservices, A. Bouguettaya, Q. Z. Sheng, F. Daniel, Eds., Springer, pp. 527–549, 2014.

[64] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, “Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA,” in IC2E. IEEE, March 2014, pp. 87–96.