Architectural Concepts and Their Evolution Made Explicit by Examples

Mirco Schindler

Institute for Software and Systems Engineering
Technische Universität Clausthal, Germany
Email: mirco.schindler@tu-clausthal.de

Andreas Rausch

Institute for Software and Systems Engineering
Technische Universität Clausthal, Germany
Email: andreas.rausch@tu-clausthal.de

***Abstract*—During the evolution of a software-intensive system, deviations may occur between the implementation and the architecture of the system. One of the main reasons for this is the incomplete knowledge of developers and architects, which is based in the fact that the complexity of today's systems cannot be understood by one person in detail. In addition, there is the language gap between source code representation and architecture description. Following the technique of Programming By Example, the presented approach built up an understanding of architectural concepts with the help of examples, i.e., Architecture By Example. A approach is established to extract architectural concepts from source code and its integration into an evolutionary and incremental development process.**

***Keywords*–Software Architecture; Architecture Erosion; Managed Architectureevolution; Agile Architecturedevelopment; Machine Learning; Architecture By Example;**

I. INTRODUCTION

Typical activities within the scope of software development are development or maintenance of software systems, the implementation of new functionalities, the extension and the reuse of components or software artifacts. What all these activities have in common is a certain understanding of the source code and its underlying architecture required for its successful realization [1].

It is also typical that in smaller projects the architect and developer build a personal union and often an architectural description exists only implicitly, i.e., it is not really comprehensibly documented. If the roles in larger projects are distributed among different people, this usually improves the documentation, but this does not guarantee that the architecture description is conform to the implementation [1]. Furthermore, if the conformance is still given during the development time, knowledge is lost over time, e.g., by a personnel change, which can often lead to an architecture erosion for a long-term evolution, as described in [2] and [3].

Considering implementation and architecture, both are a subject to an evolution that is not always synchronized. New technologies influence the solutions more than in any other field in type, scope and speed. Therefore, the introduction of new technologies is often accompanied by a paradigm or a change of concepts.

On the other hand, the requirements for a system change over time, requirements are added, changed or even disappear. This leads necessarily to the fact that also the architecture is exposed to changes, because the architecture once developed does not need to fit any longer to the future requirements.

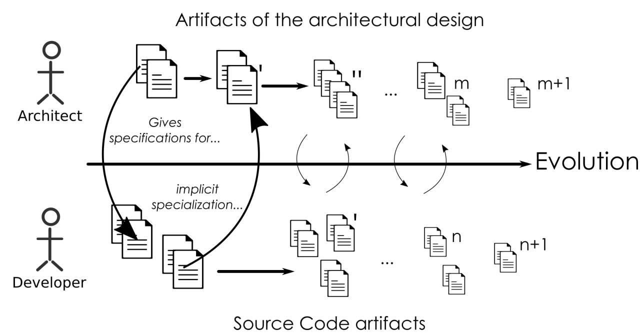

Figure 1. Evolution of Architecture- and Source Code artifacts

Whereas the verification of the specifications can be managed by the architecture, e.g., by rule-based approaches [4], the concepts of the developer are usually not directly played back to the architectural level. One reason is the different language of artifacts the architect and the developer are working with. In addition, it is not that easy to extract best practices directly from the source code artifacts, usually the architect has to exchange information directly with the developer.

The area of tension between abstract architecture description and concrete implementation is illustrated in Figure 1. If the artifacts of the architecture design define the scope for the developer, then the resulting source code artifacts should influence the architecture as well. The compromise that has to be achieved here is between full specification of the software architecture and the creative autonomy of the developer. But this compromise leads to a number of general problems in software engineering, architecture knowledge is incomplete and not up to date, architectural concepts are not well understood especially within different contexts, the same for best practices of implementation. The presented approach address the extraction of architectural concepts of source code artifacts. The occurring interaction between architect and developer will be explained in more detail in the following Section II using an example scenario and introduce to the specific problem space. In Section III, the solution approach is presented in detail and its application in different systems is explained. It ends with a conclusion and outlook.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-706-1

II. PROBLEM DESCRIPTION

In this section, the challenge of making architectural concepts explicit is introduced with the help of a clear application example and a dialog between architect and developer, which can reflect a typical situation in the daily work between architect and developer.

A. A manageable Example System

To illustrate the problem space a software system was chosen, which was designed within the project CoCoME [5]. Both architecture description as well as implementation is existing and can be found in [6].

The system implements the functionality required for a supermarket warehouse, from cash desk systems to report generation and ordering of new goods. The section that is considered here deals with the information system, whose structure is visualized in Figure 2. For clarity reasons, multiplicities and a complete labeling of the interfaces were avoided, for a complete illustration see [5]. The chosen architecture is a classic Three-Tier architecture with Service-Oriented interfaces [7] [8].

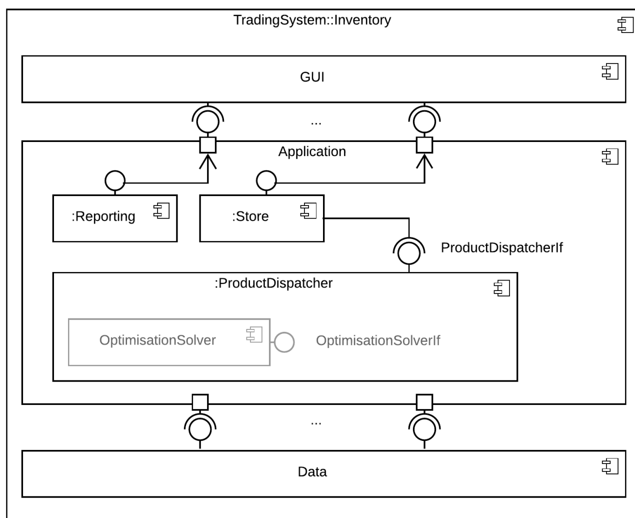

Figure 2. Structural View of the information system of the CoCoME system `TradingSystem::Inventory`

This system has also been studied in the work of Herold [4]. He describes an approach for automatically verifying compliance between a given architecture and implementation using architecture rules. Specifically, this means for the selected section of the application layer with its three internal components (see Figure 2) that the interfaces offered by these components have to be realized as Service-Oriented interfaces. The corresponding architectural rule can be simplified and not formally formulated as follows (a formal description as first-order logic statements see [4] page 139-145):

```
A Service-Oriented interface only
has service methods. A service
method is a method that only has
parameters that are primitive in
type, or the type is a Transfer
Object (TO). A TO also uses
only data types that are either
primitive or TOs.
```

Let us take a look at a typical situation and dialog that can occur during development:

B. Scenario

The SW developer HANNES gets the task to realize the component `OptimisationSolver` in the current iteration, which should be integrated into the component `ProductDispatcher`. He implements the functionality of an optimization algorithm for the distribution of goods to different supermarkets in a functionally correct way.

Afterwards the SW-Architect BO checks the realization concerning the architecture rules presented in the previous Section II-A. However, the result is not positive, the rule is violated! But HANNES does not understand why, the functionality is implemented correctly, the system does what it is supposed to do!

For the developer HANNES, the architecture rules have no relation to the code. BO shows HANNES code examples which represent the rule and thus correctly implement this architectural concept, as well as the lines where it deviates from it (see Figure 3). In this case, the examples belong to the same system, so the developer might know them or the context in which they are used. Also, for the violations, they are linked to concrete lines and contexts within the source files.

```
public interface ReportingIf extends Remote {
  public ReportTO getStockReport(StoreTO storeTO)
     throws RemoteException;
  public ReportTO getStockReport(EnterpriseTO enterpriseTO)
     throws RemoteException;
  public ReportTO getMeanTimeToDeliveryReport(EnterpriseTO enterpriseTO)
     throws RemoteException;}
```

```
public interface CashDeskConnectorIf extends Remote {
  void bookSale(SaleTO saleTO) throws RemoteException;
  ProductWithStockItemTO getProductWithStockItem(long productBarCode)
     throws NoSuchProductException, RemoteException; }
```

```
public interface OptimisationSolverIf {
 public Hashtable<StoreTO, Collection<ProductAmountTO>> solveOptimization(
   Collection<ProductAmountTO> requiredProductAmounts,
    Hashtable<Store, Collection<StockItem>> storeStockItems,
    Hashtable<Store, Integer> storeDistances); }
```

Figure 3. Examples of correct implementation and detected deviations

What does the deviation between the specified rule and the implementation mean? As specified for a Service-Oriented interface only primitive types or Transfer Objects are allowed as parameters. Our developer HANNES now understands his mistake after reviewing the examples and uses the Transfer Objects. BO checks the result again and it fits, it is all good and a new code example that implements the architecture concept of a service-orientated interface also exists.

In the next iteration HANNES gets the task to implement a new database query. Following the realization the architect BO checks the implemented code artifacts, but all rules fail. Form his perspective it seems, that HANNES did something completely wrong. He shows him code examples, which are correct realizations of the failed rules. But HANNES replies that he swapped the database framework and changed the access concept, as the new technology recommends a different concept that increases data security, but this requires a different way of handling the data. The architect is familiar with this

and tells him: "All right, then we need to adjust the rules that will apply to data storage components from now on in this system".

BO select new rules representing the new concept, checks the code with the new rules and all is well!

As the scenario shows, an architectural violation can be solved in two ways, by adapting the architecture or by adapting the implementation. But how can we decide which way makes more sense in which case? The presented approach will support this decision making process by extracting the concepts from source code with the goal to get an understanding of what the developer did to get an indication of the type of violation. It can be summarized in a general research question as follows:"How can developers' best practice be identified and reflected to the architecture level?" This includes the representation and the extraction of concepts, as well as the way the acquired knowledge can be used to support the software engineering process.

III. SOLUTION AND ITS APPLICATION

In this section, the approach, which is making architectural concepts explicit, is explained and each step is illustrated with the help of an application example.

An architectural concept is defined in this work as:

```
"A characterization and
description of a common, abstract
and realized implementation-,
design-, or architecture solution
within a given context represented
by a set of examples and/or rules."
[9]
```

According to this definition, this covers a wide range of concept candidates, a few examples of which are given below:

- *Conventions:* Naming, package- and folder structure, vocabulary, domain model ...; *Desig-Pattern:* Observer, Factory, ...;
- *Architecture-Pattern:* client-server system, tiers, ...;
- *Communication Paradigms:* Service-orientated, message based, ...;
- *Structural Concepts:* Tiers, Pipes, Filter, ... or
- *Security Concepts:* encryption, SSO, ...

A. The Overall Approach

The holistic approach is visualized in Figure 4 and consists essentially of three activities (SELECTION, EXTRACTION and GENERALIZATION), which are explained in detail in the following Section III-B.

A **Concept** c is described as a set of **Properties** P. Furthermore, for a **Element** e there is a so-called **Detector** D is defined. A **detector** is a binary function $d_{p_j} \in D$, which maps a concrete property $p_j \in P$ and a concrete element $e_i \in E$ as follows:

$$d_{p_j}(e_i) = \begin{cases} 1 & \text{, iff the element } e_i \text{ fullfills the property } p_j \\ 0 & \text{, else} \end{cases} \quad (1)$$

An element can be a system artifact such as a class, a method, or a relationship between two elements in a realized system. The considered source code artefacts are transformed in the so-called **System Snapshot** $\mathfrak{S}$. This language independent model representation M, which is an extension of the models of [4] [10] [11], still contains the link to the specific lines within the source code files. This link ensures traceability between the extracted architectural concepts and the source code. The following applies, that the set of elements E is a true subset of the model M, $E \subset M$. A special kind of element are associations A, which are representing dependencies between elements.

The model instance of a given software systems, which is stored in the System Snapshot, is transformed into the **facts base** $\mathfrak{F}$, which describes the fulfillment of concepts for the concrete elements. In addition, it is the repository of the new learned or derived facts. The facts are stored within the fact base in two different ways, a data structures that are organized in a table structure $\mathfrak{F}_E^{|P|\times|E|}$, lists facts that refer to elements or associations, and a graph structure that describes facts about elements, their context and their dependencies between them. In the following the structure of the matrix $\mathfrak{F}_E^{|P|\times|E|}$ is given, combining the System Snapshot with the selected detectors:

P \ E	e_1	$e_2 \cdots e_i\ a_1\ a_2$	$\cdots$	a_n
p_1	$d_{p_1}(e_1)$	$\cdots$	$\cdots$	$d_{p_1}(a_n)$
p_2				
$\vdots$	$\vdots$	$\ddots$		$\vdots$
p_j	$d_{p_j}(e_1)$	$\cdots$	$\cdots$	$d_{p_j}(a_n)$

$, with\ A \subset E$

The concrete graph structure is defined as a common weighted graph $G(V, E', w)$ with a given set of vertexes V and edges E'. Whereby in this approach the vertexes are referred to the set of elements $E \setminus A, wherby A \subset E \subset M$ and the edges are linked to the set of associations A, $\mathfrak{F}_{G(E\setminus A, A, w)}$:

$$e_i \xrightarrow{w_{ij}} e_j \quad , with\ w_{ij}\ number\ of\ assoziations\ of\ the\ same\ type$$

The last of the three data pools is the **Concept Space** Ω. All known concepts are stored in it, whereby a concept is represented as a named element and linked with its detectors and examples, i.e., with concrete source code examples that fulfill this concept.

$$c_i \in \Omega := \{\text{identifier}_i, D_i, R_i\} \quad (2)$$

In this definition D_i is the set of detectors, which are able to check a given element if this fulfills the concept c_i or not. The set R_i includes all known elements, which are fulfill the concept c_i.

The central artifact is the **Configuration** Σ. In each iteration, i.e., with each new execution of the selection step, a new instance $\sigma_i \in \Sigma$ is created. The configuration is used for the information exchange between the three activities and contains all decisions which are made during the execution, both by humans and by the algorithms. The result of the approach is the so-called **Concept Performance Record**. This record informs about the concepts that are in the analyzed system realization.

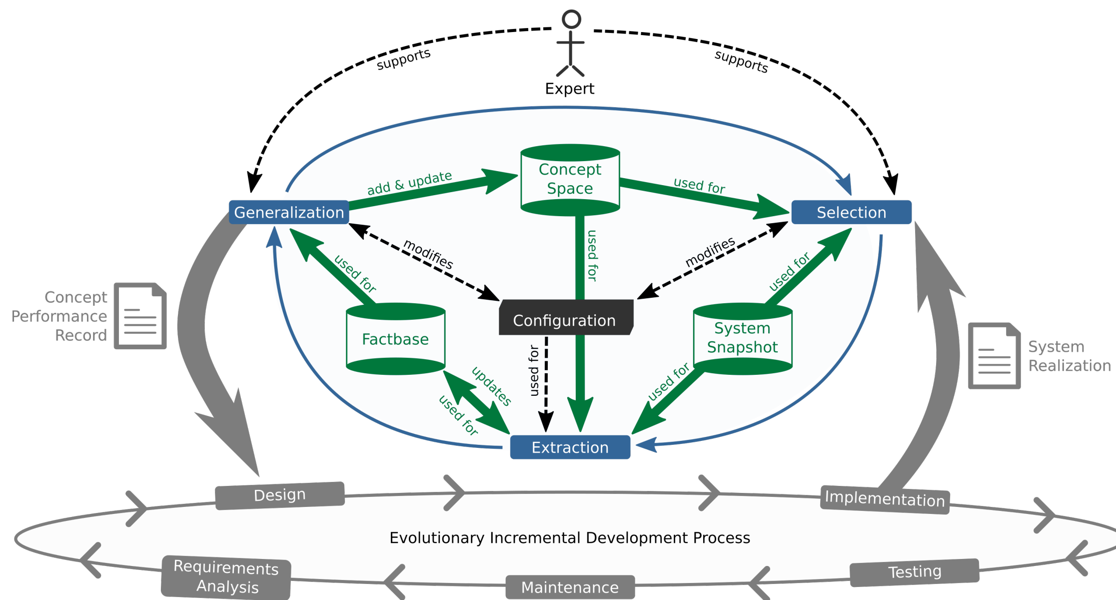

Figure 4. Overview of the solution approach and integration into a development process

In the following, this extraction process is described in detail, as well as how it can be integrated into an evolutionary and incremental development process and can support both architect and developer in their work.

B. The Extraction Phase

Altogether the defined process for the extraction of architectural concepts consists of three activities Selection, Extraction and Generalization (blue boxes in Figure 4), which are carried out iteratively and together are called the extraction phase.

Selection: In this step, an expert decides which parts of the system to analyses and what is the initial set of concepts to use. So it may be possible to reduce the number of initial concepts, since some basic knowledge of the system is usually available, like e.g., whether the system was developed object-oriented or not.

The selected subset of the system or the total system if no selection has been made and the initial selected concept set represented by its detectors are stored in the configuration and serve as input for the next step. In addition, as already described, the source code artifacts are transformed into a language-independent representation and stored as a system snapshot.

Extraction: This step is fully automated. First the fact base is created for all selected elements by executing each selected detector for each element. Each element is therefore assigned to a set of positive and negative properties according to the detector definition.

Subsequently, different algorithms from the field of machine learning are used to extract possible new facts or combinations from them. These so-called **Concept Candidates** $\hat{C}$ are added to the fact base as non-validated concepts. This also includes references to the **representatives** R of these concept candidates, i.e., elements that fulfill this newly extracted concept.

Generalization: After enriching the fact base with new facts respectively potential new concepts, an expert will validate these facts in the generalization step. The expert decides on the basis of the representatives of concept candidates whether it is a relevant concept or not. These decisions are stored in the configuration. Thus, the configuration contains the information about selected detectors and system artifacts as well as the newly extracted and validated concepts on this basis, whereby concept candidates, which were classified as not valid, are stored as so-called anti-pattern for this system snapshot.

Based on this decision process, new detectors are generated. This can be done manually or automatically, manually by storing e. .g. rules, which can be checked and automatically by training a so-called neural network detector with the examples. Finally, this new knowledge has to be integrated into the Concept Space, where it is checked whether the newly extracted concepts are already contained and simply not selected in this iteration. If a concept with this identifier is already contained, the expert has to decide whether it is the same or a different concept or variant, i.e., whether it should be added as a new one or whether the existing detector should be trained with the new representatives.

Implementation of the approach The algorithms listed below describe only one possible selection for the implementation of the individual activities, i.e., the algorithms mentioned fulfill the required characteristics. At this point, however, it cannot be guaranteed that there will be methods that perform better.

During selection, the expert can be supported by static code analysis procedures or clustering techniques, for example, to obtain different views of the system in order to select the source code artifacts and detectors relevant to the given task. Tools like SPAA [12] [13] [14] can be used. Also the visualizations of the software as Software City [15] would be conceivable to assist the expert.

In order to derive new concept candidates from the fact

base, various clustering methods and statistical methods were evaluated. Statistical analysis based on frequency and distribution analyses provide a first clue for concept candidates, but are not suitable for the automation of the extraction step like the clustering methods.

Different clustering methods were chosen to group elements that are similar in terms of their properties in order to derive potential candidates from them. The following were examined: Neural Gas [16], Growing Neural Gas [17], as well as Self-Organizing-Maps (SOM) [18], whereby in particular the work of Matthias Reuter [19], [20] was taken into account.

These algorithms were used to find concepts at element level, such as special data objects like the transfer objects described in the example [5]. In addition, they were used to extract similar properties of dependencies between elements, to extract different types of dependencies such as special communication channels or different types of relationships such as an inheritance relationship typical for an object-oriented realization.

The following algorithms were used to extract new facts from the facts represented by the graph structure: Graph Kernels [21], graph clustering approaches such as SPAA [13] [12] [14] and t-SNE [22] to find similarities and anomalies within the graph. An example in which the coordinator pattern as a candidate results from such an extraction is described in [2].

A SOM is also used for the creation of new detectors within the framework of generalization by training with the representatives. The selection and parameterization of adequate methods in all activities, is the focus of the current, further and ongoing work.

C. The Evolution Phase

As shown in Figure 4, the approach described in the previous section can be embedded into any evolutionary and incremental development process. This means that after each implementation step the source code can be analyzed and these results are available for the next evolution step in the design activity.

This results in a holistic approach that considers the evolution of architectural concepts on two levels. On the one hand an identified concept itself is subject to changes, e.g., it can be refined or degenerated by further examples and on the other hand the application of architectural concepts to a concrete system can change during development by switching to another technology for example.

The generated Concept Performance Record can support the system architect in understanding the realized concepts. This information can be combined with the results of a conformity check, i.e., with a list of architecture violations referring to concrete source code artifacts.

If, for example, the developer was not familiar with the architecture and this is the reason for the violation, it can be fixed by the developer in the next implementation step so that no erosion occurs. On the other hand, it can be decided that the reason for the violation is an unsuitable architecture. In this case, the Concept Performance Record can support the planning of architectural changes by making the aspects the developer has in mind explicit at the architectural abstraction level through examples.

Another aspect is the improvement of the development and maintenance process through monitoring. We can assume that the configuration and all data pools (fact bases and concept spaces) are stored and versioned in a common repository. As already introduced, a concept stored in a concept space consists of a triple, its identifier, at least one detector, and a set of examples that fulfill this concept. As a result of the use of new technologies, frameworks or programming paradigms, it can lead to new concepts that may replace old concepts, so that once extracted concepts disappear over time and are no longer identified. The comparison of two Concept Performance Records from different versions of a product can lead to indications of mutations of concepts. This can also help to detect the erosion of the product or even a product line architecture at an early stage and to react in a managed way.

D. Scenario Mapping

If we have a look to the scenario defined in Section II-B, than the described activities can support it.

We assume that in the first conflict, where no transfer objects were used as parameters, the concept Transfer Object is known, stored in the Concept Space and was also selected. This means that at least one detector and corresponding examples of Transfer Objects are existing. In this case, the execution of the detector will result for the parameters of the `solveOptimization` method (see Figure 3) not positive. With the knowledge that these elements concept a indication for a deviation is given. Furthermore the architect has a direct linkage to the examples for the following discussion with the developer.

In the second case, where the database framework and with it a concept change is implemented, the following can be observed. A concept that has been detected or extracted before is no longer recognized - it has disappeared. On the other hand, previously unknown concepts have been extracted. The fact that an element no longer satisfies a concept fulfilled in the previous iteration and is now referenced in a newly extracted concept is a strong indication of a concept change.

IV. CONCLUSION AND OUTLOOK

The central element of the approach are the directly referenced source code examples through which machine-learned models become explainable, architectural concepts explicit and the adaption of a software architecture takes place in a managed way driven by the extracted knowledge from source files.

The approach presented here contributes to making software architectures explicit. The developer's understanding of architectural concepts is increased by code examples, i.e., a representation he is familiar with. The architect is supported during the decision making process, as he gets additional information about the architecture deviations with whose help he can decide whether the resolution of the violation is brought about by an adaptation in the source code or by a change in the architecture and thus the ideas of the developer are transferred to the architecture level. In this case, it is the examples that provide the architect with the information, with the focus on the properties and concepts that are fulfilled by the source code artifacts concerned.

With regard to the scenario described in Section II-B another question arises here: *Must be the interface* `OptimisationSolverIf` *realized as a service-oriented interface at all?* As visualized in Figure 2, it is only used within the application layer and not provided for external access, i.e., a possible solution for the conflict would also be to allow different types of implementation related to the context of the interface. In this case, the approach supports describing the context by the kind of the connection which exist between interface and its environment, and the differences between the different realization kinds which become understandable by the description of their characteristics.

Also described in the scenario is a technology change. Referring to the prototypical development or testing in the context of a product line as described in [9], the change becomes explicit. For example, it becomes clear which concepts are lost and which are added and which elements are affected. With regard to the rolling out of a concept change to variants of the product line, the detectors can simply determine which elements are affected by the no longer permissible concept.

As an outlook we would like to mention the evaluation with different OpenSource systems (e.g., Java Path Finder[23]. (approx. 178,000 LOC), jEdit[24] (ca. 58,400 LOC), PMD[25] (ca. 110.350 LOC), log4j[26] (approx. 158,600 LOC) and the enrichment of the fact base with semantic information, such as functional information or data describing the runtime behavior.

The statement:*"Satisfying rules can be a hard job for software developer!"* is not surprising considering that software development is a highly creative process [27] and therefore not every architectural violation is basically bad; rather it is necessary to explain the deviation in its context. To achieve this, it is necessary to make the architecture implicitly implemented by the developer explicit. If we also take into account that agile development methods are increasingly used, which are carried out with sprint cycles of e.g., two weeks, that architecture, too, is progressing at this rate of evolution, or it can at least be ensured that it does not erode. A managed architecture evolution in short cycles is only possible if both architect and developer have a basis that both understand.

As well as a common understanding, evolution also takes place on both levels, driven on the one hand by new requirements, which may no longer be met by the current architecture, and on the other hand driven by new technologies or programming paradigm and best practices.

Just like the Programming By Example [28] approach an understanding is here created through examples, too - but at the architectural level, with the goal to bring the architectural and the coding perspective together.

REFERENCES

[1] M. Gharbi, A. Koschel, A. Rausch, and G. Starke, Software Architecture Fundamentals: A Study Guide for the Certified Professional for Software Architecture – Foundation Level – iSAQB compliant. Heidelberg: dpunkt, 2018.

[2] A. e. a. Grewe, "Automotive Software Product Line Architecture Evolution: Extracting, Designing and Managing Architectural Concepts," in International Journal on Advances in Intelligent Systems. IARIA, 2017, pp. 203–222.

[3] C. e. a. Knieke, "A Holistic Approach for Managed Evolution of Automotive Software Product Line Architectures," in ADAPTIVE 2017. Wilmington, DE, USA: IARIA, 2017, pp. 43–52.

[4] S. Herold, Architectural compliance in component-based systems: Foundations, specification, and checking of architectural rules, 1st ed., ser. SSE-Dissertation. München: Verl. Dr. Hut, 2011, vol. 5.

[5] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, Eds., The Common Component Modeling Example: Comparing Software Component Models (Lecture Notes in Computer Science / Programming and Software Engineering), 1st ed. Springer, 2008.

[6] http://www.cocome.org/, accessed: 2018-12-12.

[7] C. Szyperski, Component Software: Beyond Object-Oriented Programming (2nd Edition), 2nd ed. Addison-Wesley Professional, 2002.

[8] R. Reussner, Ed., Handbuch der Software-Architektur, 1st ed. Heidelberg: Dpunkt-Verl., 2006.

[9] A. e. a. Grewe, "Automotive Software Systems Evolution: Planning and Evolving Product Line Architectures," in ADAPTIVE 2017. Wilmington, DE, USA: IARIA, 2017, pp. 53–62.

[10] C. Deiters, Beschreibung und konsistente Komposition von Bausteinen für den Architekturentwurf von Softwaresystemen, 1st ed., ser. SSE-Dissertation. München: Dr. Hut, 2015, vol. 11.

[11] Malte Mues, "Taint Analysis: Language Independent Security Analysis for Injection Attacks," Master's Thesis, Technische Universität Clausthal, Clausthal, 2016.

[12] M. Schindler, C. Deiters, and A. Rausch, "Using Spectral Clustering to Automate Identification and Optimization of Component Structures," in Proceedings of 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013, pp. 14–20.

[13] M. Schindler, "Automatische Identifikation und Optimierung von Komponentenstrukturen in Softwaresystemen," Diploma Thesis, Technische Universität Clausthal, 2010.

[14] M. Schindler, A. Rausch, and O. Fox, "Clustering Source Code Elements by Semantic Similarity Using Wikipedia," in Proceedings of 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2015, pp. 13–18.

[15] R. Wettel, "Software Systems as cities," PhD Thesis, Università della Svizzera Italiana, Switzerland, Lugano, 2010.

[16] M. Cottrell, B. Hammer, A. Hasenfuss, and T. Villmann, "Batch and median neural gas," Neural Networks, vol. 19, no. 6-7, 2006, pp. 762–771.

[17] B. Fritzke, "A growing neural gas network learns topologies," in Advances in neural information processing systems, 1995, pp. 625–632.

[18] T. Kohonen, "The self-organizing map," Neurocomputing, vol. 21, no. 1, 1998, pp. 1–6.

[19] M. Reuter and H. H. Tadijne, "Computing with Activities III: Chunking and Aspect Integration of Complex Situations by a New Kind of Kohonen Map with WHU-Structures (WHU-SOMs)," in Proceedings of IFSA2005, Y. e. Liu, Ed. Springer, 2005, pp. 1410–1413.

[20] M. Reuter, "Computing with Activities V. Experimental Proof of the Stability of Closed Self Organizing Maps (gSOMs) and the Potential Formulation of Neural Nets," in Proceedings World Automation Congress (ISSCI 2008). TSI, 2008.

[21] A. Gisbrecht, W. Lueks, B. Mokbel, and B. Hammer, "Out-of-sample kernel extensions for nonparametric dimensionality reduction," in ESANN, vol. 2012, 2012, pp. 531–536.

[22] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," Journal of Machine Learning Research, vol. 9, no. Nov, 2008, pp. 2579–2605.

[23] http://babelfish.arc.nasa.gov/hg/jpf/jpf-core, accessed: 2018-12-12.

[24] http://www.jedit.org/, accessed: 2018-12-12.

[25] https://pmd.github.io/, accessed: 2018-12-12.

[26] https://logging.apache.org/log4j/2.x/, accessed: 2018-12-12.

[27] M. Gu and X. Tong, "Towards Hypotheses on Creativity in Software Development," in Product focused software process improvement, ser. Lecture Notes in Computer Science, F. Bomarius, Ed. Berlin [u.a.]: Springer, 2004, vol. 3009, pp. 47–61.

[28] H. Lieberman, Your wish is my command: Programming by example, ser. Morgan Kaufmann series in interactive technologies. San Francisco: Morgan Kaufmann Publishers, 2010.