A Taxonomy for Design Decisions in Software Architecture Documentation

Jan Keim[0000−0002−8899−7081], Tobias Hey[0000−0003−0381−1020], Bjarne Sauer, and Anne Koziolek[0000−0002−1593−3394]

KASTEL – Institute of Information Security and Dependability,
Karlsruhe Institute of Technology (KIT),
Karlsruhe, Germany
{jan.keim, hey, koziolek}@kit.edu, bjarne.sauer@t-online.de

Abstract. A software system is the result of all design decisions that were made during development and maintenance. Documentation, such as software architecture documentation, captures a variety of different design decisions. Classifying the kinds of design decisions facilitates various downstream tasks by enabling more targeted analyses. In this paper, we propose a taxonomy for design decisions in software architecture documentation to primarily support consistency checking. Existing taxonomies about design decisions have different purposes and do not fit well because they are too coarse. We take an iterative approach, starting with an initial taxonomy based on literature and considerations regarding consistency checking. Then, we mine open-source repositories to extract 17 software architecture documentations that we use to refine the taxonomy. We evaluate the resulting taxonomy with regard to purpose, structure, and application. Additionally, we explore the automatic identification and classification of design decisions in software architecture documentation according to the taxonomy. We apply different machine learning techniques, such as Logistic Regression, Decision Trees, Random Forests, and BERT to the 17 software architecture documentations. The evaluation yields a F_1-score of up to 92.1% for identifying design decisions and a F_1-score of up to 55.2% for the classification of the kind of design decision.

Keywords: Design Decisions · Software Architecture · Documentation · Decision-Making · Software Design · Mining Software Repositories

1 Introduction

During software development, a lot of design decisions about the system are made. The architecture as well as the software system itself are the result of all design decisions. The successful development and maintenance of a software system, therefore, relies on everyone involved in the development understanding the design decisions. With documentation, design decisions can be captured and made easily available to, e.g., help new team members or to recapture past decisions. As a result, the success of a software system also relies on the quality of its documentation (cf. Parnas [22]).

There are different kinds of architectural design decisions (ADDs) in a system and in software architecture documentation (SAD). For example, there are existence decisions about, e.g., components like “The Logic component handles the business logic.” Other examples include decisions about the development process like “The system is fully written in Java” or decisions about design rules like “There should be no dependencies between microservices.”

Taxonomies and ontologies help to structure a body of knowledge, provide a better understanding of interrelationships, and improve decision-making processes [29]. Taxonomies for ADDs, for example, can enable focused discussions about certain ADDs and can raise awareness of the different kinds of ADDs. There are already existing taxonomies and ontologies to classify ADDs. For example, there are Kruchten’s ontology of design decisions [18], the architectural design decision model introduced by Jansen and Bosch [12], and the architectural design decision rationale framework by Falessi et al. [7]. Taxonomies always serve specific purposes and are designed for certain scopes. Existing taxonomies are rather broad. However, some purposes can benefit from fine-grained classes.

One example for such a purpose are consistency analyses. Various researchers already pointed out the importance of consistency checking in software architecture artifacts: Based on two surveys with industrial practitioners, Wohlrab et al. [30] conclude that there are various kinds of inconsistencies in software architecture artifacts that need to be addressed. Moreover, Keim et al. [15] as well as Lytra and Zdun [20] argue in favor of consistency analyses and inconsistency management to avoid missing or losing crucial information about the software system and underlying design decisions. Knowing the kind of ADDs allows analyzing consistency more targeted using different analyses. For example, analyses that check an ADD that prohibits dependencies between microservices need to identify dependencies and check for compliance. In contrast, to examine the existence of some component, other analyses have to investigate existing components and find said component.

Kruchten’s ontology (cf. [18]) distinguishes three kinds of existence decisions: structural and behavioral decisions as well as ban or non-existence. For targeted consistency checking, this level of detail is insufficient. The same problem applies to alternatives like the model by Jansen and Bosch [12] or the framework by Falessi et al. [7]. Therefore, there is the need of a more fine-grained taxonomy to serve the purpose of consistency checking.

In this paper, we present a taxonomy of design decisions in SAD to enable in-depth analysis of ADDs. This taxonomy is intended for analyzing inconsistency, but might as well be helpful for other use cases that can make use of more fine-grained classifications, e.g., traceability link recovery or the generation of specific test cases. To construct the taxonomy, we create an initial taxonomy that is based on literature and theoretical considerations for consistency analyses. We then adapt and refine the taxonomy using SADs that we mined from 17 open-source repositories. The resulting taxonomy is both argumentatively and empirically evaluated. For the empirical evaluation, we perform a small user study. We additionally explore the automatic identification and classification of ADDs in informal textual SADs according to the proposed taxonomy. For this, we employ different machine learning techniques such as Logistic Regression (LR), Decision Trees (DTs), Random Forests (RFs), and the language model Bidirectional Encoder Representations from Transformers (BERT) [6].

The data to our research is published online [14]. This data contains the documentation texts and a classification of contained ADDs. Moreover, we provide the source code of our automated classification and the evaluation results.

The remainder of the paper is structured as follows: We present foundations and related work in Section 2 before we outline our procedure for creating the taxonomy in Section 3. The resulting taxonomy is presented in Section 4, and we evaluate and discuss this taxonomy in Section 5. We explore automated classification of design decisions in SAD and analyze the results on our dataset in Section 6. In Section 7, we argue about threats to validity before we finally conclude this paper in Section 8.

2 Foundations and Related Work

We divide research with particular high relevance to our work into foundations of taxonomy building and classification schemata/taxonomies for design decisionsand automatic analysis of design decisions.

2.1 Foundations for Taxonomy Building

Ralph [23] recommends the following steps for generating a taxonomy: Choosing a strategy, selecting a site, collect data, analyze data, conceptually evaluate the taxonomy, writing-up and peer review. He recommends secondary studies, grounded theory, and interpretive case studies as possible strategies. Data collection and analyses are often conducted iteratively to let preliminary findings drive further insights.

Bedford [3] introduces the following principles that a taxonomy should fulfill: *consistency*, *affinity*, *differentiation*, *exclusiveness*, *ascertainability*, *currency*, and *exhaustiveness*. *Consistency* means that the rules for creating, changing, and retiring categories should be consistent. *Affinity* is the principle that states that each category definition should be based on its parent category and that categories in a lower hierarchy should have an increased intention. Moreover, when creating sub-categories, there should be at least two (*differentiation*). However, there does not need to be a differentiation on all levels. All categories should be *exclusive* so that two or more categories do not overlap, and each category has a clear scope and clear boundaries. Names should be immediately clear and understandable (*ascertainability*) and they should reflect the language of the domain (*currency*). Finally, categories should be *exhaustive* in a way that they cover the whole domain.

2.2 Classification Schemata for Design Decisions

There are several approaches to define a classification schema or taxonomy for design decisions. Kruchten [18] proposes an ontology of architectural design decisions that distinguishes between existence, property, and executive design decisions. Existence design decisions are further divided into structural, behavioral, and ban decisions. The former two are related to the creation or interaction of elements, whereas the latter state that a certain element will not appear in the design or implementation. Kruchten regards positively stated property decisions as design rules or guidelines, and calls negatively stated decisions constraints. In general, property decisions state traits or qualities of the system. According to Kruchten, executive decisions are driven by either the business environment, affect the development process, the people, or the organization, and extend the choices of technologies and tools.

In an expert survey, Miesbauer and Weinreich [21] noticed that developers often face existence decisions, whereas executive decisions are only present in a quarter of decisions. Property decisions even only constitute 10% of the cases. Additionally, they identify four levels of design decisions: implementation, architecture, project, and business

Jansen and Bosch [12] regard software architecture as a composition of design decisions over time. They propose *Archium* that explicitly models the relations between design decisions and software architecture during the whole development process. The software architecture is then described as a set of design decisions, deltas, and design fragments.

Falessi et al. [7] present a framework that associates design decisions with their goals and available alternatives. The goal is to improve the maintenance of systems by improving the comprehensibility of design decisions.

Based on Kruchten's ontology and Falessi et al.'s use cases, Zimmermann et al. [31] structure the decision-making process according to the three steps: decision identification, decision making, and decision enforcement. They group similar design decisions under a shared topic. Then, each topic is assigned to one of the three levels of abstraction: conceptual, technology, and asset level.

In a survey on architectural design decision models and tools, Shahin et al. [26] identified decisions, constraints, solutions, and rationales as key elements of all nine considered models. The models mostly differ in the used terminology.

As our goal is to enable and improve tasks like consistency checking for software architecture documentation, existing work does not fit to our purposes. The approaches are often too coarse, or the abstraction level does not fit. In this work, we will explain why this is the case and present a more fine-grained taxonomy that solves this issue.

2.3 Automatic Analysis of Design Decisions

Approaches that automatically identify or analyze design decisions have the benefit that they reduce or even omit the costly manual effort needed to recover design decisions using a domain expert.

Bhat et al. [4] extract design decisions from issue trackers, using a machine learning-based approach. Based on the classification schema of Kruchten [18], they train a classifier on 1500 issues that is able to detect existence decisions and their subclasses, structural, behavioral, and ban decisions. By using Support Vector Machines (SVMs), they are able to detect existence decisions with an accuracy of 91.29% and classify them into the subclasses with 82.79% accuracy.

Li et al. [19] also use machine learning to extract design decisions, but focus on mailing lists. Their SVM-based classifier is able to detect whether a sentence in an e-mail contains a design decision with a F_1-score of 75.9%. By using an ensemble learning method and feature selection, Fu et al. [8] are able to improve the results achieved in a multi-class classification of decisions into the five classes design, requirement, management, construction, and testing decision. The ensemble of Naïve Bayes (NB), LR, and SVM with 50% features selected achieves the best result with a weighted F_1-score of 72.7%.

3 Research Design

Table 1. The 17 case studies used for developing and evaluating the taxonomy. #Lines shows the number of lines in each documentation file (without empty lines) . Links to the projects and the used documents can be found in our supplementary material [14].

Project	Domain	#Lines	Project	Domain	#Lines
CoronaWarnApp	Healthcare	369	SCons	Software Dev.	79
Teammates	Teaching	252	OnionRouting	Networking	51
Beets	Media	125	Spacewalk	Operating System	38
ROD	Data Mgmt.	119	Calipso	Web Dev.	30
ZenGarden	Media	109	MunkeyIssues	Software Dev.	23
MyTardis	Data Mgmt.	100	BIBINT	Science	22
SpringXD	Data Mgmt.	95	OpenRefine	Data Mgmt.	21
QMiner	Data Analysis	92	tagm8vault	Media	16
IOSched	Event Mgmt.	81			

Our process of creating the taxonomy is oriented towards the guidelines by Ralph [23]. We perform an iterative classification with 17 documentations from open-source projects. We use the insights after applying our taxonomy to identify shortcomings and adapt the taxonomy. This way, we try to also ensure Bedford's principles (cf. [3]).

We first develop an initial taxonomy that is based on literature, majorly on Kruchten's ontology [18]. We adopt the major classes (*Existence*, *Property*, and *Executive* decisions) and add additional insights from related work, e.g., by Jansen and Bosch [12], Bhat et al. [4], and Miesbauer and Weinreich [21]. For example, existence decisions are the most common kind of decisions [21] and, therefore, should be subdivided. Considering the rules for manual classification by Bhat et al. [4], there are two kinds of structural decisions: decisions about the structure within the system and decisions about third-party systems and external dependencies like libraries and plug-ins. We also consult literature and specifications about software architecture and software design (e.g., [5, 9, 24, 27, 28]). Lastly, we make adaptations based on own considerations about needs for stated application areas. For example, we disagree with Kruchten for having *bans* only as a subclass of existence decisions as bans can appear in other categories such as executive decisions as well. As a result, we view bans as negatively formulated decisions and, as such, as a property of each kind of decision.

Next, we iteratively apply the taxonomy on a number of SADs and adapt the taxonomy based on shortcomings and problems during classification. After each iteration, we adapt the taxonomy. We first perform a pre-study (first iteration) with one case study and then use three case studies per iteration. This allows us to identify necessary adaptations early while minimizing the risk of overfitting. We end the whole iterative process when the taxonomy is stable, i.e., the taxonomy does not need to be updated in two consecutive iterations (six SADs).

For this process, we need a number of SADs. Therefore, we mine English SADs from open-source projects. We identify these by querying GitHub for "architecture" and randomly selecting 50 projects. We then manually filter those based on the following criteria: We select English documentation with acceptable text quality, different SAD size, and various domains. Overall, we use 17 case studies with different sizes and domains that are listed in Table 1 and that can be found in our dataset [14].

Finally, we evaluate the taxonomy both argumentatively and empirically and experiment with automatic classification based on the taxonomy.

4 Taxonomy of Design Decisions

Using the process described in Section 3, we gained valuable insights into design decisions in SADs. These insights play a crucial role in the final design of the taxonomy and, as such, also serve as reasoning for some classes. Generally, existing structures were often too general and broad, thus needed further distinction. For example, we further subdivide structural decisions, decisions about functionality, and technological executive design decisions.

After following the process described in Section 3, the final taxonomy is composed of the following classes that are also displayed in Figure 1 and Figure 2. The root of the taxonomy captures the existence of a design decision. Design decisions are then split into three main categories: existence decisions, property decisions, and executive decisions. These three categories originate from Kruchten's ontology [18], and we agree on his categorization. However, there is a major difference in how we see decisions about *Bans*, as we already stated in Section 3. While Kruchten sees bans as a subclass of existence decisions, we see a ban as a negatively formulated design decision and a ban can exist in other sub-categories such as executive decisions as well. For example, there can be a design decision that states that a certain component (e.g., *cache*) should not exist but also that a certain tool or process should not be employed (e.g., for legal reasons). As a consequence, there is no category that indicates bans in our taxonomy. We regard it as an attribute of a design decision that indicates inclusion or exclusion of something. The same applies to delayed decisions, i.e., decisions that are made but postponed to a later point in time.

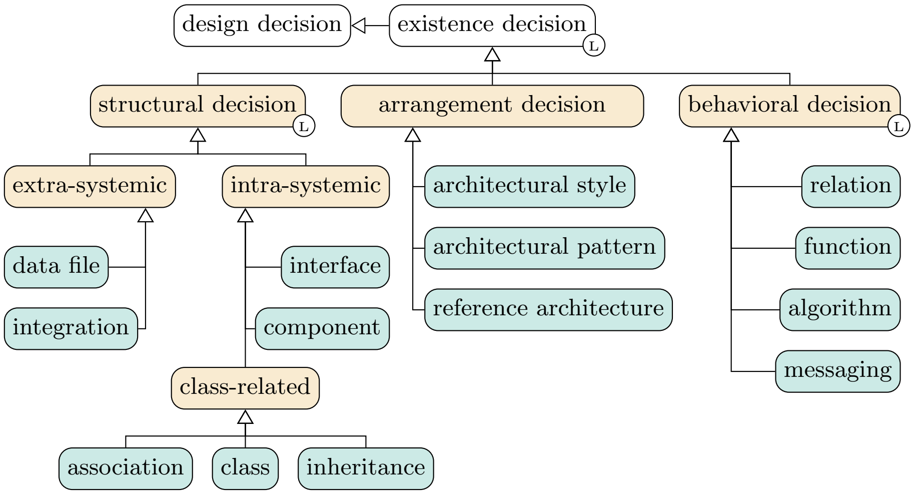

Fig. 1. Sub-categories of existence decisions with leaves highlighted in green and intermediate categories in orange. Categories derived from literature are labeled with Ⓛ.

The taxonomy subdivides existence decisions into structural decisions, arrangement decisions, and behavioral decisions. These three sub-categories are again further refined, as it is visible in Figure 1. An important distinction that we make is between intra- and extra-systemic structural decisions. Coming from our insights, we define *systemic* as parts of the executable system that are developed for this particular system. Therefore, *extra-systemic* means everything that comes from external sources in regard to this system like libraries or external plug-ins. This definition helps us to draw a sharp line between external parts and parts that are developed especially as part of the system, which is especially useful for consistency analyses. External parts are often modelled in a fashion that uses, e.g., external calls. This clearly needs different checks than internal parts that are explicitly modelled.

Aside from existence decisions, there are property decisions and executive decisions. Figure 2 shows the sub-categorization of property decisions and executive decisions. In Table 2 and Table 3, we define the different categories and sub-categories and describe them in more detail.

Table 2. Existence decision and its sub-categories

existence decision: State whether some software element or artifact will be present in the system's design or implementation [18].

structural decision: Break down a system in reusable subsystems and components.

extra-systemic: Decisions that make use of software elements beyond the system's borders to add them to the system under development or to feed in data.

data file: Non-executable files that provide information that can be transferred and shared between (sub-) systems.

integration: In this category fall externally developed software elements like libraries and plug-ins, but also reused components from other projects/systems that are not specifically developed for the current system under development.

intra-systemic: Decisions affecting the division of a system in subsystems and smaller units developed for this particular system.

interface: Provide "a declaration of a set of public features and obligations that together constitute a coherent service" [5]. Interfaces within a software system serve to bundle functionality together.

component: A unit of composition with contractually specified interfaces and explicit context dependencies [28]. We additionally recognize packages as such compositions . This category comprises decisions about components that are developed for the system under development, not imported components (cf. *integration*).

class-related: Deal with the development of software systems along classes of objects. Even if class-related decisions may not be architectural in general, many architecture documentations contain them, so we include them in our taxonomy.

class: A set of objects with consistent properties and functionality [5]. One can make decisions about its existence, structure and naming.

association: A set of links that are tuple[s] of values that refer to typed objects [5].

inheritance: Through *inheritance*, a child class adopts properties of its parent class. Design decisions in this category include the existence and design of inheritance hierarchies and the determination of (abstract) parent and child classes.

behavioral decision: Relate to how software elements are connected and how they interact to provide functionality [18].

relation: Decisions about connectors and dependencies between software elements that can be established, modified or excluded. This determines the accessibility of the functionality of the elements.

function: Decisions about specific functionality of a software system. These *functions* are usually implemented as methods.

algorithm: Refer to a named sequence of operations to realize certain functionality for a specific problem. The procedures must be sufficiently general and state the general goal. Otherwise, it should be categorized as *function* or *messaging*.

messaging: Decisions about functionality concerning communication between software elements through method calls, sending messages, and data packages (cf. [11]). With messaging, the sender usually invokes some behavior at the receiver.

arrangement decision: Decisions for an *arrangement* of software elements in a known manner and under consideration of related principles. Decisions in this category will affect both structure and behavior of the system.

architectural style: Provide solution principles for architectural problems that are independent of the given application and should be used throughout the whole architecture [25]. Examples are Layered Architecture or Client-Server Architecture.

architectural pattern: A proven, reusable solution to a recurring problem. The scope is broader (concerning components) compared to design patterns (concerning code base parts) [24]. In relation to architectural styles, architectural patterns are more specific to a certain problem. Examples for architectural patterns are Model-View-Controller and Domain Model.

reference architecture: Defines a set of architectural design decisions for a specific application domain [24]. The AUTOSAR architecture for the automotive domain can be seen as an example.

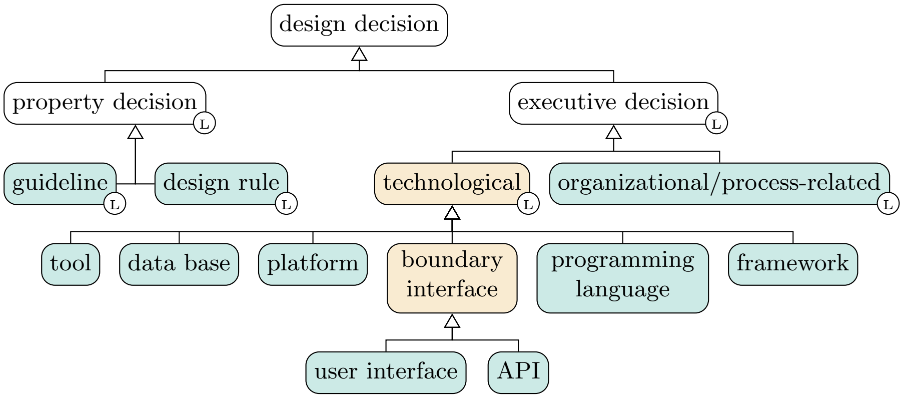

Fig. 2. Sub-categories of property decisions and executive decisions with leaves highlighted in green and intermediate categories in orange. Categories derived from literature are labeled with Ⓛ.

5 Evaluation

In this chapter, we evaluate our taxonomy with respect to its suitability and different properties that we derive from the principles by Bedford [3] and the guidelines by Ralph [23] (cf. Section 2). We argue about the suitability for the intended purpose in Section 5.1. In Section 5.2, we consider the structure of the taxonomy and argue about the consistency, currency, affinity, differentiation, and exhaustiveness. Moreover, we look into the usability, application, and handling of the taxonomy, especially the exclusiveness and ascertainability in Section 5.3.

5.1 Evaluating the Purpose

The main purpose of the ontology is enabling consistency checking using documentation on one side and, e.g., formal architectural models or code on the other side. Consistency analyses need to treat certain kinds of design decisions differently. To enable that, the taxonomy needs clear differentiation and non-overlapping classes (exclusiveness, cf. Sections 5.2 and 5.3). Moreover, the classes need to capture concepts and kinds of design decisions that need different treatment. This is obvious for classes belonging to different upper categories. For instance, structural decisions can be checked for consistency using a structural view on the architecture, while executive decisions cannot be checked this way. This also holds true for classes that fall under a certain sub-category. For example, components need to be treated differently from interfaces or associations, as all these have different attributes and involved elements. Treating named algorithms needs external knowledge to capture all involved parts. For messaging, there are different software elements involved and, therefore, the consistency check explicitly needs to check for these. We argue that similar argumentation can be done for all classes and categories. Overall, we conclude that the purpose

Table 3. Property decision and executive decision and their sub-categories

property decision: State an enduring, overarching trait of quality of a system [18].	
design rule: A rule, positively or negatively stated, expresses some trait or quality the system design must strictly fulfill. This is a combination of Kruchten's classes *design rules* and *constraint* [18]. This does not include observable quality attributes such as performance or reliability.	guideline: Recommended practices that improve the system's quality. They are less strict than design rules and usually not enforced.
executive decision: Relate to environmental aspects of the development process [18].	
technological: State the choice for or against numerous technologies that enable and support software development. tool: Developer *tools* can utilize and automate the development process. data base: Decisions referring to storing data in *data bases* as well as decisions on query languages (e.g., SQL) and data base technologies (e.g., MongoDB). platform: Decisions that refer to an environment of software and hardware components that allow development, deployment, and execution. programming language: An agreed *programming language* for a system or components leads to syntactical and semantic rules for writing code. framework: Abstractions that allow the development of extensive applications. A framework can either be an architecture framework (cf. [1]) or a software framework like a web framework (e.g., Django).	boundary interface: Interfaces that are located on the system's boundary and enable the connection from and to other systems and technologies. API: *Application programming interfaces* provide external functions that can be used in another software system. If a design decision is not about the functionality of communications but about used protocols like HTTP and TCP, we also classify it into this class. user interface: Located between the (human) user and the technical system, and allows the user to enter commands. There are different kinds of user interfaces like graphical user interfaces (GUIs) or command line interfaces (CLIs). organizational/process-related: Sum up all decisions concerned with the development process, the methodological procedure, and the project organization.

for this taxonomy is clear and that the taxonomy fits to this purpose. However, we still need to collect further evidence by creating applications that use the taxonomy in future work.

5.2 Evaluating the Structure

The proposed taxonomy is based on related work and refines the ontology proposed by Kruchten [18]. As Kruchten's proposal is widely used in the community, especially as a foundation for approaches that automatically classify design decisions, our taxonomy can be regarded as consistent with existing work. Our taxonomy only breaks with the structure of Kruchten's ontology at one instance: We regard inclusive/exclusive wording as an attribute of each design decision instead of explicitly modelling ban decisions as a sub-category of existence decisions. With this decision, our taxonomy allows modelling excluding statements on different occasions and at a more fine-grained level. For example, we can model excluding executive decisions. Such excluding executive decisions occurred in our case studies and, therefore, we argue that our taxonomy is more exhaustive. Besides Kruchten's proposed classes, we also regarded Bhat et al.'s rules [4] for manual identification of design decisions while refining the taxonomy.

We reflect the language of the domain by using well-known and widely accepted names and terms within the software architecture domain (currency). We further ensure that there is an affinity of each subclass to its superclasses by iteratively adding new subclasses to existing classes in a top-down fashion. The proposed taxonomy also fulfills the differentiation principle, as each further refined class has at least two subclasses. Concerning the taxonomy's exhaustiveness, we cannot eliminate the possibility that other case studies may include decisions that would require different subclasses. However, as our superclasses stem from Kruchten's ontology and these classes were sufficient to classify each occurring design decision in various related work (cf. [4,21]), our taxonomy is at least as exhaustive as related work.

Additionally, our iterative approach follows the methodology of the National Information Standards Organization [2] and produced no further subclasses even for heterogeneous documentations (length and domain).

5.3 Evaluating the Application

We base our evaluation for the applicability and reliability on the guidelines by Kaplan et al. [13]. We let two software engineers (doctoral researchers) independently classify the design decisions in the case studies Calipso, Spacewalk, and SpringXD with only the information on the taxonomy provided in Section 4 at hand. We calculate Krippendorff's α (Kα) [17] to measure the inter-annotator agreement. The annotators achieved an overall Kα of 0.771 across the three studies. If the results are calculated per project, we achieve an average Kα of 0.841. The discrepancy stems from the documentation of SpringXD seemingly being more difficult. The documentation of SpringXD is far longer than the other two and contains several parts that were no actual architecture documentation. Thus, we state that the difficulty and uniformity of different users applying the taxonomy depends on the quality of the documentation. Still, the Kα values indicate a reasonable agreement that clearly exceed the lower bound of 0.66 and come close or exceed the commonly accepted threshold of 0.8 (cf. Krippendorff [16]). These results are promising regarding that the taxonomy comprises 24 leaf classes. As the annotators were also able to identify a fitting class for each decision, we are confident to state that the taxonomy can be applied in cases where a refined classification of design decisions is beneficial.

Table 4. Distribution of design decisions in our dataset [14]

Taxonomy class					Primary	Secondary	Total
existence	structural	extra		integration	38	7	45
				data file	17	6	23
		intra		component	132	23	155
				interface	19	0	19
			class-r.	class	77	26	103
				association	65	12	77
				inheritance	19	3	22
	arrang.			architectural style	26	14	40
				architectural pattern	21	3	24
				reference architecture	4	2	6
	behavioral			function	271	43	314
				relation	56	14	70
				algorithm	48	6	54
				messaging	37	11	48
property				design rule	73	0	73
				guideline	8	2	10
executive				organizational/process-related	9	1	10
	technological			platform	72	15	87
				programming language	39	19	58
				framework	32	13	45
				data base	40	14	54
				tool	5	2	7
		boundary i.		API	51	12	63
				user interface	31	10	41
identified design decisions (in 1622 lines)					1190	258	1448
lines without a design decision							432

Table 5. Classification results of classifying if a line contains a design decision (binary), the kind of the most prevalent design decision (multi-class, weighted average) and the kinds of a design decision (multi-label, weighted average). Weighted average can produce F_1-scores not in between precision (P) and recall (R).

	$\text{LR}_{\text{trigram}}$			DT_{BoW}			$\text{RF}_{\text{bigram}}$			BERT		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
binary	0.877	0.894	0.885	0.850	0.852	0.851	0.831	0.970	0.895	0.901	0.942	**0.921**
multi-class	0.452	0.451	0.427	0.314	0.322	0.304	0.346	0.353	0.269	0.575	0.559	**0.552**
multi-label	0.578	0.326	0.396	0.453	0.394	0.406	0.482	0.090	0.145	0.679	0.427	**0.500**

6 Exploratory Automated Application

To be able to efficiently utilize design decisions in SAD in other approaches they have to be automatically identified and classified. The ability to automatically identify the design decisions and their subclasses also heavily affects the taxonomy's applicability. To measure this, we employed different established supervised machine learning-based approaches to classify sentences in SADs according to our taxonomy. To train the classifiers, we use the dataset that resulted from our taxonomy building process. The dataset consists of the 1622 lines of the 17 case studies (cf. Table 1). Table 4 gives an overview on the distribution of classes in the dataset. We labeled only the most prevalent primary and also the most obvious secondary decision per line.

To train the classifiers, we experimented with the preprocessing steps stop word removal, lemmatization, transformation to lowercase, and their combinations. For preprocessing, only lemmatization and lowercasing shows positive effects on the results. Likewise, we experimented with bag-of-words (BoW), term frequency-inverse document frequency (tf-idf), and bi- or trigrams as vector representations for the preprocessed lines. Based on the vector representations, we use Logistic Regression (LR), Decision Trees (DTs), and Random Forests (RFs) as ML techniques. This aligns with approaches in literature (e.g., [4, 19]). For the sake of brevity we only report the best combinations here. Additionally, we use BERT [6] as a language model-based approach. For BERT, we use the same setup that performed well for our approach for classifying requirements. For details, see Hey et al. [10]. We fine-tune a classifier based on the *bert-base-uncased* model with 16 epochs, a batch size of 8, and use lowercasing.

To train and test the models on the dataset, we perform a random 5-fold cross validation with three repetitions. For multi-class and multi-label classification we report the weighted F_1-score as it takes the imbalanced dataset into account. Table 5 shows the results for three different classification tasks. For binary classification into design or no design decision, the best performing configuration is the BERT-based classifier achieving 92.1% F_1-score. As the input differs from existing work, we cannot directly compare the results. However, our results are similar to the results by Bhat et al. [4] for identifying design decisions in issue trackers. Our results are also better than the results by Li et al. [19] for identifying design decisions in mailing lists.

Based on these promising results, we apply the multi-class classification to lines containing a design decision. Here, BERT outperforms the best other approaches by over 12.5 percentage points. Given the number of classes and the limited dataset, the result of 55.2% weighted F_1-score is promising. As the abstraction levels of the leaves deviate, the performance of a single multi-class classifier might be limited. In the future, we plan to apply hierarchical approaches.

To be able to identify multiple decisions in one line, a multi-label classification is needed. For BERT, we use a multi-label implementation. For the other approaches, we train One-vs-Rest classifiers for each label. Performance decreases slightly in this more difficult setting, but results are still close to our best results for non-multi-label classification. Again, BERT outperforms the other approaches with a weighted F_1-score of 50%. In this setting, the decision tree classifier performs far better and even outperforms the logistic regression.

7 Threats to Validity

To discuss threats to validity, we follow the guidelines by Runeson and Höst [25].

Construct Validity — We applied commonly used experimental designs and common metrics for classification tasks. However, there might be a certain bias in the selection of the use cases. We ensured construct validity by using projects from different domains and with different characteristics like size or architecture styles and patterns.

Internal Validity — We used the same case studies for the taxonomy creation and automated classification. This way, we create a fitting classification schema but assume representative documentations. Additionally, we only labeled the most prevalent direct decisions and the most obvious implicit decisions. In rare cases, there are more decisions and, thus, there is potential bias in the selection.

External Validity — In our evaluation, we examined 17 publicly available case studies from different domains. We aimed for a representative selection, but we risk that not all facets and aspects of design decisions are covered. All classes are represented, but some classes only have small representation (cf. Table 4). This is caused by the nature of some kind of design decisions to be mentioned only once per documentation (e.g., reference architecture) compared to those occurring more often (e.g., decisions about components).

Reliability — For our experiments on automated classification, we manually created a gold standard. We tried to minimize bias from single researchers by discussing the taxonomy in detail and how to decide on the most prevalent design decision. Moreover, we discussed certain classifications.

8 Conclusion

SAD captures design decisions about the architecture and executive decisions and makes these decisions easily available. In SADs, there are different kinds of design decisions that cover aspects about the structure, execution, or certain properties and guidelines of the system.

In this paper, we argued in favor of a taxonomy about design decisions in SADs with the main purpose to enable and improve consistency analyses. For this, we propose a taxonomy that is based on literature and an iterative process of applying and improving the taxonomy on 17 open-source SADs. This taxonomy consists of 24 leaf-classes, where the new fine-grained classes are designed to support consistency analyses.

We evaluated the taxonomy by arguing for its validity and fitness for its purpose. We argue why our taxonomy and its construction follow the principles by Bedford [3] and the guidelines by Ralph [23]. Moreover, we performed an empirical study with two subjects classifying design decisions achieving a good inter-annotator agreement (K$\alpha = 0.771$).

Lastly, we explored different widely used approaches to automatically identify and classify design decisions in SADs. The results are promising and show that we can identify design decisions with a F_1-score of up to 92.1%. We can also classify into the leaf-classes of our taxonomy with a F_1-score of up to 55.2%. We publish our data and source code online [14].

In the future, we want to obtain feedback from open source architects about the taxonomy and classification. We also plan to further explore the usage of this taxonomy with applications that use classification for inconsistency detection. This way, we can also collect more evidence about the applicability, usability, and suitability. Moreover, we want to identify and fine-tune further approaches for the automatic classification to get reliable results.

References

1. ISO/IEC/IEEE Systems and software engineering – Architecture description. ISO/IEC/IEEE 42010:2011(E) (2011). https://doi.org/10.1109/IEEESTD.2011.6129467
2. ANSI/NISO: Guidelines for the Construction, Format, and Management of Monolingual Controlled Vocabularies. Standard, NISO (2005), https://www.niso.org/publications/ansiniso-z3919-2005-r2010
3. Bedford, D.: Evaluating classification schema and classification decisions. Bulletin of the American Society for Information Science and Technology **39**(2), 13–21 (2013). https://doi.org/10.1002/bult.2013.1720390206
4. Bhat, M., Shumaiev, K., Biesdorf, A., Hohenstein, U., Matthes, F.: Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach. In: Lopes, A., de Lemos, R. (eds.) Software Architecture. pp. 138–154. LNCS, Springer International Publishing (2017)
5. Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., Tolbert, D.: Unified modeling language (UML) version 2.5.1. Standard, Object Management Group (OMG) (Dec 2017), https://www.omg.org/spec/UML/2.5.1
6. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Proceedings of the 2019 NAACL. pp. 4171–4186. ACL (2019). https://doi.org/10.18653/v1/N19-1423
7. Falessi, D., Becker, M., Cantone, G.: Design decision rationale: Experiences and steps ahead towards systematic use. SIGSOFT Softw. Eng. Notes **31**(5), 2–es (Sep 2006). https://doi.org/10.1145/1163514.1178642
8. Fu, L., Liang, P., Li, X., Yang, C.: A machine learning based ensemble method for automatic multiclass classification of decisions. In: Evaluation and Assessment in Software Engineering. p. 40–49. EASE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3463274.3463325
9. Golden, B.: A Unified Formalism for Complex Systems Architecture. Ph.D. thesis, Ecole Polytechnique X (May 2013)
10. Hey, T., Keim, J., Koziolek, A., Tichy, W.F.: NoRBERT: Transfer Learning for Requirements Classification. In: IEEE 28th RE. pp. 169–179 (2020). https://doi.org/10.1007/978-3-319-65831-5_10
11. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (Aug 1978). https://doi.org/10.1145/359576.359585

12. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). pp. 109–120 (Nov 2005). https://doi.org/10.1109/WICSA.2005.61
13. Kaplan, A., Kühn, T., Hahner, S., Benkler, N., Keim, J., Fuchß, D., Corallo, S., Heinrich, R.: Introducing an evaluation method for taxonomies. In: EASE 2022: Evaluation and Assessment in Software Engineering. ACM (2022)
14. Keim, J., Hey, T., Sauer, B., Koziolek, A.: Supplementary Material of "A Taxonomy for Design Decisions in Software Architecture Documentation" (2022), https://doi.org/10.5281/zenodo.6956851
15. Keim, J., Koziolek, A.: Towards Consistency Checking Between Software Architecture and Informal Documentation. In: 2019 ICSA. pp. 250–253 (2019)
16. Krippendorff, K.: Reliability in content analysis: Some common misconceptions and recommendations. Human communication research **30**(3), 411–433 (2004)
17. Krippendorff, K.: Content Analysis: An Introduction to Its Methodology. SAGE Publications (May 2018)
18. Kruchten, P.: An ontology of architectural design decisions in software-intensive systems. 2nd Groningen Workshop on Software Variability pp. 54–61 (2004)
19. Li, X., Liang, P., Li, Z.: Automatic Identification of Decisions from the Hibernate Developer Mailing List. In: Proceedings of EASE '20. pp. 51–60. ACM (2020)
20. Lytra, I., Zdun, U.: Inconsistency Management between Architectural Decisions and Designs Using Constraints and Model Fixes. In: 23rd Australian Software Engineering Conference (Apr 2014). https://doi.org/10.1109/ASWEC.2014.33
21. Miesbauer, C., Weinreich, R.: Classification of Design Decisions – An Expert Survey in Practice. In: Drira, K. (ed.) Software Architecture. pp. 130–145. LNCS, Springer (2013). https://doi.org/10.1007/978-3-642-39031-9_12
22. Parnas, D.L.: Precise Documentation: The Key to Better Software. In: Nanz, S. (ed.) The Future of Software Engineering, pp. 125–148. Springer (2011)
23. Ralph, P.: Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. IEEE Transactions on Software Engineering **45**(7), 712–735 (Jul 2019). https://doi.org/10.1109/TSE.2018.2796554
24. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A.: Modeling and Simulating Software Architectures: The Palladio Approach. MIT Press (Oct 2016)
25. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir Software Eng **14**(2), 131 (Dec 2008)
26. Shahin, M., Liang, P., Khayyambashi, M.R.: Architectural design decision: Existing models and tools. In: 2009 Joint WICSA & ECSA. pp. 293–296 (2009)
27. Sullivan, K.J., Griswold, W.G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. SIGSOFT SE Notes **26**(5), 99–108 (2001)
28. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Pearson Education (2002)
29. Vegas, S., Juristo, N., Basili, V.R.: Maturing software engineering knowledge through classifications: A case study on unit testing techniques. IEEE Transactions on Software Engineering **35**(4) (2009). https://doi.org/10.1109/TSE.2009.13
30. Wohlrab, R., Eliasson, U., Pelliccione, P., Heldal, R.: Improving the Consistency and Usefulness of Architecture Descriptions: Guidelines for Architects. In: 2019 ICSA. pp. 151–160 (2019). https://doi.org/10.1109/ICSA.2019.00024
31. Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., Schuster, N.: Reusable Architectural Decision Models for Enterprise Application Development. In: Software Architectures, Components, and Applications. Lecture Notes in Computer Science, Springer (2007). https://doi.org/10.1007/978-3-540-77619-2_2