CEUR-WS.org/Vol-2978/saerocon-paper6.pdf

Hard Cases in Source Code to Architecture Mapping using Naive Bayes

Tobias Olsson, Morgan Ericsson and Anna Wingkvist

Department of Computer Science and Media Technology, Linnaeus University, Kalmar/Växjö, Sweden

Abstract
The automatic mapping of source code entities to architectural modules is a challenging problem that is necessary to solve if we want to increase the use of Static Architecture Conformance Checking in the industry. We apply the state-of-the-art automatic mapping technique to eight open-source systems and find that there are systematic problems in the automatically created mappings. All of these eight systems have small modules that are very hard to map correctly since only a few source code entities are mapped to these. All systems seem to use some naming strategy, mapping source code to modules; however, naming is often ambiguous. We also find differences in ground truth mappings performed by experts, which affect mappings based on these, and that architectural refactoring also affects the mapping performance.

Keywords
Orphan Adoption, Software Architecture, Source Code Clustering, Naive Bayes

1. Introduction

Our previous studies [1, 2] of automated techniques to map source code entities to high-level software architectural modules suggest that some entities are much harder to map correctly than others. Even using the best algorithm and different parameters, certain entities always seem to fail to map correctly. We conduct an exploratory study to determine whether our intuition is correct, i.e., that these hard cases exist, and if they do, what their properties are, and what makes them hard to map correctly.

The software architecture of a system captures major design decisions at a high level of abstraction and enables internal and external qualities such as performance, portability, reusability, and maintainability [3]. It serves as a guide for the many decisions that are made during the implementation of a system. As the system evolves, the source code must continue to conform to the architecture or risk accumulating technical debt and no longer possess the desired qualities.

Static Architecture Conformance Checking (SACC) is a collection of methods, such as Reflexion modeling [4], that statically analyze source code to ensure that it does not introduce architectural violations [5, 6]. These methods require an architecture model, with modules and dependencies, and a source code model, with entities and concrete dependencies, e.g., due to inheritance or method invocations. They also require a mapping from the source code model to the architecture model to determine whether the source code dependencies are convergent, absent, or divergent compared to the allowed dependencies specified in the architecture model.

The need for a mapping between the source code and architecture models is a significant reason why SACC has not reached widespread use in the software industry [3, 5, 7, 8]; the tools and methods exist, but the mappings do not or are outdated. Many tools address this by combining manual mapping and regular expressions to filter file, module, and package names. Still, such approaches have proven to be time-consuming and error-prone [5, 7, 8].

If we want to automate the mapping process using, e.g., machine learning, it is vital to understand the hard cases. If there is a class of entities that our approach cannot map automatically or always maps to the wrong modules, we need to ensure that these are part of the initial set that a human expert maps. We perform an exploratory study using eight systems with ground truth mappings to determine whether such a class exists. Once we have established that it exists, we determine its properties to identify its members automatically. We then investigate why these properties make the entities difficult to map to ensure that they will not reduce the effectiveness of the machine learning approach; we do not want it to learn the wrong things from the hard cases.

We hypothesize that at least some hard cases would be difficult for a human to map and that different human experts would disagree on how they should be mapped. This can, for example, be due to poor structuring or the evolution of the system. We rely on different ground-truth mappings of the same system and metrics to identify such cases and study how well these correlate to the hard cases.

ECSA2021 Companion Volume
tobias.olsson@lnu.se (T. Olsson); morgan.ericsson@lnu.se (M. Ericsson); anna.wingkvist@lnu.se (A. Wingkvist)
0000-0003-1154-5308 (T. Olsson); 0000-0003-1173-5187 (M. Ericsson); 0000-0002-0835-823X (A. Wingkvist)
© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

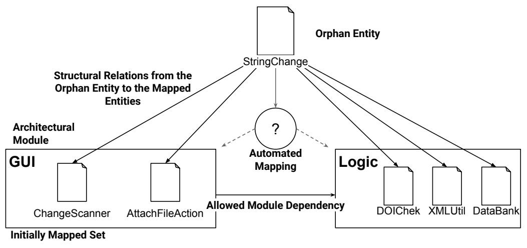

Figure 1: An example mapping that shows the initial sets of the *GUI* and *Logic* modules of JabRef 3.7. A new orphan *StringChange* is about to be mapped.

2. Automated Mapping

To reason about how well an implementation conforms to the intended architecture using, e.g., Reflexion modeling, we need a mapping from the source code to the architecture. In this section, we discuss how such a mapping can be created semi-automatically, starting from an initial set of mapped source code entities.

The source code model consists of Entities (E) and Dependencies (ED). The entities are, e.g., classes defined in a programming language, and the ED are due to, e.g., method calls and inheritance, see *StringChange*, *ChangeScanner*, etc., in Figure 1.

The architecture model consists of Modules (M) and Dependencies (MD) between these. The modules represent the major parts of the architecture; see, e.g., *GUI* and *Logic* in Figure 1. The directed MD indicates how these modules are allowed to interact and depend on each other. If there, for example, is an MD from GUI to Logic, then entities mapped to GUI are expected to call entities mapped to Logic.

An automated mapping algorithm aims to map each entity to the correct module without human assistance. For example, classes in the implementation that deal with the application's business rules should be mapped to the module Logic. Once this mapping exists, we can compare the ED of the implementation to the MD allowed by the architecture and determine whether they are convergent, absent, or divergent [4].

We rely on *orphan adoption* [9] to map entities to modules automatically. An unmapped entity is considered an orphan that should be adopted by one of the modules, e.g., *StringChange* in Figure 1. Tzerpos and Holt identify four criteria that can affect the mapping. *Naming*, naming standards can reveal what module is suitable. *Structure*, dependencies between an orphan and already mapped entities can be used as a mapping criterion. *Style*, modules are often created using different design principles (e.g., high cohesion or not). Classifying the orphan based on style can give hints on how to use, for example, the structure criteria. *Semantics*, the source code itself can be analyzed to determine its purpose and its similarity to the purpose of the modules.

A sub-problem of orphan adoption is orphan kidnapping, where software evolution causes a need for remapping an entity to a new module, or in other words, *corrective clustering*. Tzerpos and Holt identify a fifth criterion related to orphan kidnapping, *Interface minimization*; it is not a good idea to reassign an entity to another module if the removal of the entity will cause the module to get a larger public interface, i.e., the entity is an entry point/facade to the module.

HuGMe [10, 8] relies on orphan adoption to map from the source code to the architecture model. It starts from an initial set of entities that are manually mapped to the correct module. The remaining entities are considered orphans. HuGMe is applied iteratively, and as the set of mapped entities can grow for each iteration, more orphans have the potential to be automatically mapped. In each iteration, there is also the possibility for human intervention using the result of the failed automatic mapping attempts as a guideline. The automatic mapping is done by calculating the attraction between the orphan and the mapped entities for each module. Christl et al. present two attraction functions, *CountAttract* and *MQAttract*, based on dependencies, i.e., the *structure* criterion. Bittencourt et al. evaluate two new attraction functions based on information retrieval techniques. They use the names of modules and entities and the names of identifiers in the entities to form vocabulary documents for modules and entities, i.e., the *naming* and *semantic* criteria. They then use a cosine similarity function, *IRAttract*, and latent semantic indexing, *LSIAttract*, to calculate the attraction values.

Our attraction function, *NBAttract*, combines ideas from the previous two and considers the *structure*, *naming*, and *semantic* criteria [2]. The approach is similar to that of Bittencourt et al., but we instead use a Naive Bayes classifier to determine similarity to other entities. To include the *structure* criterion, *NBAttract* uses a novel approach, Concrete Dependency Abstraction (CDA), to encode dependencies as text [2]. NBAttract has outperformed CountAttract in our previous study [2], and CountAttract was not clearly outperformed in [7]. We, therefore, only use NBAttract in the remainder of this paper.

3. Method

Based on our experiences with different attraction functions, we hypothesize that no matter how well the function performs, there is a specific set of entities that are always misclassified. We seek to investigate this further to determine whether our hypothesis is correct or if the misclassifications happen by chance due to randomness in the composition and size of the initial set.

We have previously implemented a tool to evaluate different mapping approaches, including reporting detailed mapping results [11]. We use this tool to create a new dataset over the mapping results for each source code entity.

We run *NBAttract*, with the following settings. We use an initial set of mapped entities of random size and composition. We extract package names, filenames (these correspond to the outer class names in Java), attribute identifier names, and variable identifier names from the source code entities in the initial set and tokenize these based on Camel-case and the characters - and _. The tokens are then stemmed using a Porter stemmer. Tokens that are shorter than three characters are removed. We use our CDA technique to represent dependencies as text strings. We use a binary token frequency (present or not) and 0.9 as the threshold for automatic classification.

These settings correspond to the settings used in [2] with one exception; we do not require the initial set to contain at least one source code entity from each module in this study. We are interested in how individual files are mapped to find possible flaws in the technique, which is why we allow for a module to be empty initially.

As we run several experiments with random initial sets, we get a dataset that shows the correct mapping of each entity and the number of mappings for each entity and module. Based on this information, we can compute an error rate for each entity according to Equation 1. If the attraction function was completely stochastic, the error rate for each entity would converge to the stochastic error rate, defined in Equation 2.

$$err_{nba} = \frac{|\textbf{erroneous mappings}|}{|\textbf{mappings}|} \quad (1)$$

$$err_{sto} = \frac{|\textbf{modules}| - \mathbf{1}}{|\textbf{modules}|} \quad (2)$$

As NBAttract is not a stochastic function, the err_{nba} for an entity should converge to something less than err_{sto} if there are no systematic problems, i.e., it should systematically produce better mappings than a random mapping. Hence, we can conclude that there are systematic problems if we do not find such a convergence for a source code entity after several iterations. If we find systematic errors in a majority of the systems, we will further analyze all problematic entities to find common, possible causes for the misclassification. An entity is considered problematic if its $err_{nba} \geq 0.5$, i.e., it is misclassified in 50% or more of the mappings. The motivation for this limit is that a non-problematic attraction function should, on average, produce a correct mapping in at least 50% of the cases for each entity. This part of the research is highly exploratory. We investigate the possible reasons for misclassification based on our own experience and the advice from related work, and present exciting findings from the data. The ultimate goal is to construct strategies to detect entities with a high risk of being misclassified so that a human can intervene and classify these manually. More specifically, we will investigate:

Is the set of problematic entities a good candidate for the initial set? This set needs human intervention for automatic mapping to perform well, effectively removing the problem from the automatic mapping. This can be assessed by computing the F1 score of the precision and recall, as we did in [2]. We will compare the F1 scores across the entire range of initial set sizes visually.

Is the set of problematic entities related to small modules? In general, machine learning techniques need good data to perform. In particular, there is a need for a balanced dataset where there is approximately the same amount of data to learn from in each class. If the dataset is imbalanced, there is a high chance that smaller classes will not be properly handled. An architectural module should contain a fair amount of source code entities. Still, there may exist modules that hold source code entities that do not fit well in other modules, or the system may be under evolution, and intended source code has not been created yet, etc. We need to know if such small modules exist and whether they are common or problematic.

Is the set of problematic entities related to entities with poor naming? Tzerpos and Holt [9] define naming as one of the key criteria that influence the mapping. In our experience, it is also a common strategy for developers to create folders, packages, and filenames that reflect the modular architecture to some degree. It would thus be interesting to know if the naming of source code entities includes the module's name it is mapped to. It is also interesting to know if there are ambiguities in the naming, i.e., if several module names match the name of a source code entity.

Is the set of problematic entities related to entities on the border of a module? Bibi et al. [12], Tzerpos and Holt [9], and Bittencourt et al. [7] state that dependencies have an impact on the mappings. We use a textual representation of dependencies in NBAttract, but this may not be good enough. We will investigate the ratio of external dependencies, e.g., an entity with many external dependencies would likely be an entity that lies on the border of a module. If we find a correlation between the external dependency ratio and the error rate, this could suggest that border entities are problematic.

There are several metrics based on dependencies. We use coupling (the count of all dependencies to or from all other entities) and fan (the existence of a dependency to or from all other entities). The coupling may be very high between two entities, but the fan can at most be one between two entities, i.e., fan is a subset of coupling. While coupling captures the absolute number of dependencies fan focuses on the diversity of different entities, i.e., a high fan value captures that an entity has many dependencies to other *different* entities.

Is the set of problematic entities related to problems in the ground truth mapping? We have access to two versions of the JabRef system in which the modules and relations between them are the same (same intended architecture), but the mappings are not the same for all entities. This provides an opportunity to study discrepancies in the ground truth mappings and how these affect the automatic mapping performance. One complicating factor in this analysis is that JabRef underwent an architectural evolution between these two versions. Therefore, we limit our analysis to entities that remain the same (no changes to the source code) but are mapped to different modules.

Is the set of problematic entities related to files that are being refactored due to architectural evolution? The two versions of JabRef provide an opportunity to study entities that have changed packages and mapping (a sign of architectural evolution), have changes to the source code (a sign of refactoring), or were recently added.

We study eight open-source systems implemented in Java. Ant[1] is an API and command-line tool for process automation. ArgoUML[2] is a desktop application for UML modeling. Jabref[3] is a desktop application for managing bibliographical references, and we use the 3.5 and 3.7 versions. Lucene[4] is an indexing and search library. ProM[5] is an extensible framework that supports a variety of process mining techniques. Sweet Home 3D[6] is an interior design application. TeamMates[7] is a web application for handling student peer reviews and feedback.

Table 1 presents the sizes of the systems in lines of code, number of entities, and number of modules. There exist a documented software architecture as well as a mapping from the implementation to this architecture for each system. Jabref 3.7, TeamMates, and ProM have been the subjects of study at the Software Architecture Erosion and Architectural Consistency Workshop (SAERoCon) 2016, 2017, and 2019 respectively, where a system expert has provided both the architecture and the mapping. The architecture documentation and mappings are available in the SAERoCon repository[8]. ArgoUML, Ant, and Lucene were studied by Brunet et al. and Lenhard et al., and the architectures and mappings were extracted from the replication package of Brunet et al. as well as for Sweet Home 3D. JabRef 3.5 was extracted from Lenhard et al..

[1]https://ant.apache.org
[2]http://argouml.tigris.org
[3]https://jabref.org
[4]https://lucene.apache.org
[5]http://www.promtools.org
[6]http://www.sweethome3d.com
[7]https://teammatesv4.appspot.com
[8]https://github.com/sebastianherold/SAERoConRepo

Table 1
Mapping Data Overview.

System	Lines	# Mod	# Ent	err $\geq$ 0.5		err $\geq$ err$_{sto}$	
Ant	36 699	16	468	187	39.96%	72	15.38%
A.UML	62 392	19	767	165	21.51%	74	9.65%
JR 3.7	59 235	6	1 017	107	10.52%	40	3.93%
JR 3.5	51 840	6	733	96	13.1%	51	6.96%
Lucene	35 812	7	514	60	11.67%	17	3.31%
ProM	9 947	4	261	18	6.9%	9	3.45%
S.H 3D	34 964	9	167	39	23.35%	19	11.38%
T.Mates	54 904	12	450	115	25.56%	49	10.89%

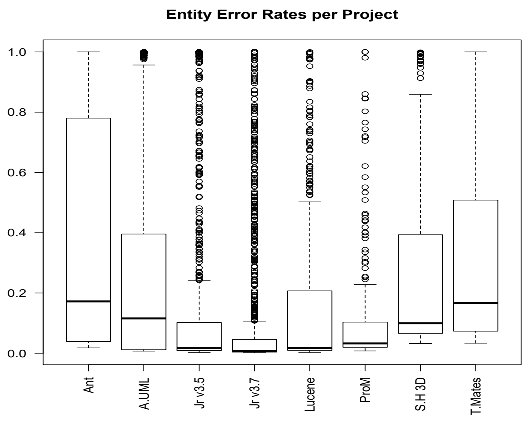

Figure 2: The entity error rates for each project.

4. Results and Analysis

We performed the experiment and collected mapping data per entity for each system. All systems show several entities always being misclassified (an error rate of 1.0) (cf. Figure 2). Table 1 shows an overview of the data collected. Note that each entity has a random chance to be included in the initial set and not be an orphan in that particular run of the experiment. There is also a chance an entity will not be mapped (e.g., due to variations in the initial set). However, each entity has been mapped at least 500 times.

We now construct the initial set using entities with $err_{nba} \geq 0.5$, i.e., only entities with $err_{nba} < 0.5$ are considered orphans, and all the troublesome entities are included in the initial set. We compare this with randomly selecting from all entities in the initial set. We collected 14 849 and 13 754 data points from the respective groups. Figure 3 shows the running median (±100 data points) and limits of the running 75th and 25th percentiles of the F1 scores, respectively, for JabRef 3.7. Since the other systems show similar trends, so we focus on JabRef. We find that our idea is promising overall, especially when the initial set size increase.

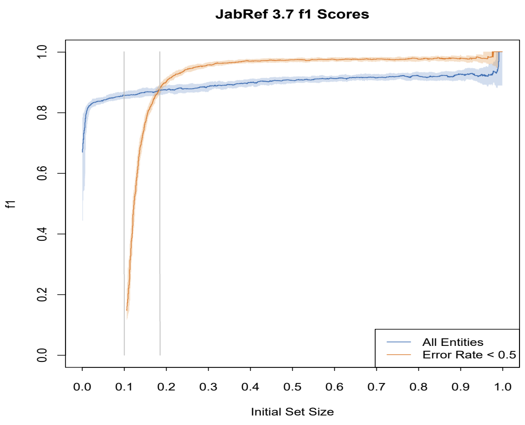

Figure 3: The running median F1 score with limits of the running 75th and 25th percentiles for JabRef 3.7 and initial set type over the whole interval of initial set sizes.

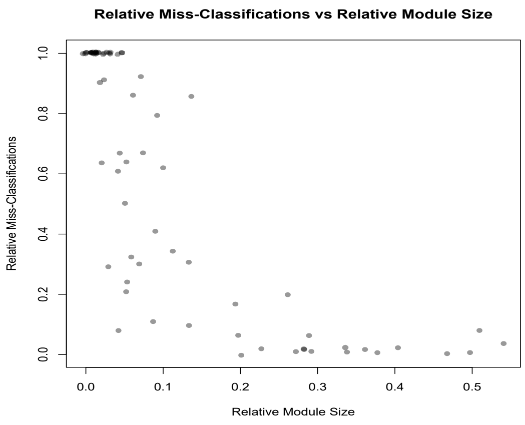

Figure 4: The relative rate of misclassified entities vs. the relative number of entities for each module. Coordinates are slightly jittered to show data points more clearly.

However, there is also an interval between the initial set sizes of 0.1 to 0.18, marked with vertical lines in Figure 3, where the F1 score is considerably lower than using all entities. This indicates that entities with $err_{nba} \geq 0.5$ are not good representatives of modules. Upon further inspection of the actual modules and entities in JabRef 3.7, we find that there is a set of modules with very few entities in the ground truth mapping, and entities mapped to these all have a high error rate.

The high error rate makes sense in general, as machine learning techniques produce better results if there is more data. More specifically, for Naive Bayes, the probability of finding an entity in such a module is very low, so it does not make sense to map entities to it. We, therefore, investigate if all systems have such small modules prone to misclassification. If entities are equally distributed among the modules of a system, there would be $\mathbf{1/|modules|}$% entities in each module. We regard a module as small if it has less than half the number of entities of $\mathbf{1/|modules|}$%. Thus we define the limit for a small module as $\mathbf{0.5/|modules|}$%. It could be argued that the lines of code should be used as a more fine-grained measure of size, i.e., mapping one huge entity in terms of lines of code. However, for example, path and file name information is per entity, and for effectively learning a pattern based on entity names, more entities are needed.

Table 2 shows the limit, the number of small modules, and the rate of misclassification of entities in these modules. A surprising result is that all systems have such small modules, and all systems have small modules where all entities are misclassified. There are 30 (out of a total of 73) modules where all entities are misclassified. Figure 4 shows how the relative number of misclassifications and relative module size are related. Note the cloud of points in the upper left corner. These are the 30 modules where all entities are misclassified. Also, note that there are small modules with a relatively low number of misclassified entities. Another factor to consider is that only 321 entities of 4377 (7.33%) are mapped to these small modules.

To measure the extent of using a naming strategy (NS) when naming concrete entities in each system, we check if the words in the package or class name for an entity contain the module name. Table 2 shows that most systems use a naming strategy (column NS) to a rather high degree. Lower values (e.g., TeamMates) are often due to a module naming discrepancy, e.g., TeamMates defines a module *view* with a corresponding path word named *ui*; however, there are also cases where there is no clear naming strategy for an entity. We consider an entity to have ambiguous naming if its path or filename contains several different module name words. For example, *net.sf.jabref.logic.net.ProxyPreferences*, from JabRef v3.7, contains both the module names *logic* and *preferences*. Ambiguity in entity naming strategy (ANS) seems to be quite common in some systems (Ant, ArgoUML, JabRef 3.5, Sweet Home 3D, and TeamMates) and not at all in others (JabRef v3.7 ProM and Lucene). In some systems, the ambiguity is caused by having a parent-level package that is also a module. For example, Ant uses *ant* as both a high-level package and a module. The misclassification rate in the ambiguously named entities (ANSM) seems to follow the inverse pattern of the ANS; the lower the ANS, the higher the ANSM. This makes sense since a higher ANS means there is more data to learn the pattern of the ambiguous naming from (if there is one).

We now turn our attention to whether entities that lie on the border of a module, i.e., have relatively many dependencies to entities in other modules, are problematic. We use the common coupling and fan metrics. Results for

Table 2
The number of entities for small modules (Limit), the number of small modules (SM), their rate of misclassification (SMM), the rate of entities with a naming strategy (NS), ambiguous naming strategy (ANS), and rate of entities with ambiguous naming that are misclassified (ANSM).

System	Limit	SM	SMM	NS	ANS	ANSM
Ant	3.57	9	92.75	100.00	85.90	29.41
A.UML	3.33	7	70.59	84.62	62.45	53.94
JR v3.5	8.33	4	93.75	71.62	30.29	42.71
JR v3.7	8.33	3	100.00	95.58	7.18	82.24
Lucene	7.14	3	57.45	99.22	2.53	90.00
ProM	12.50	1	100.00	100.00	0.77	88.89
S.H 3D	5.56	5	100.00	89.82	12.57	64.10
T.Mates	4.17	5	72.50	68.22	34.67	93.91

coupling were very similar to the results for fan. However, the fan metric seems less noisy, so we opt only to report these values. We use a scatter plot to check whether there is a correlation between these metrics and the error rate. We do this for entities that are part of large modules since small modules are a confounding factor. Figure 5 shows that there is no clear relation between the external fan ratio and the rate of relative misclassification of an entity. It would, therefore, not make sense to find a correlation between the two variables.

Yet, when we investigate the difference in external coupling for problematic versus non-problematic entities, we find a clear difference in the distribution of the external fan ratio. Problematic entities have a higher external fan ratio in general. This indicates that we need to investigate further how to correctly classify entities that lie on the border of a module. In total, there are 3 502 entities with a low error rate ($err_{nba} < 0.5$) and 497 entities with a high error rate. The number of entities with a low error rate is also higher throughout the distribution of the external fan ratio. This makes the probability of finding a problematic entity using the external fan ratio very small.

To investigate possible cases of disagreement in mappings, we study the entities that have a change in their mapping between versions 3.5 and 3.7 of JabRef. We first specifically look at entities that have only changed their mapping and not moved in the package hierarchy. We consider such nodes as having an ambiguous mapping. We find 17 such entities, 5 of which are mapped to a small module in one or both versions, which will make the error rate unrepresentative. We are left with 12 entities.

We investigate the change of source code for these entities using cloc[9] and find five entities without any code changes and seven with varying degrees of change.

[9]https://github.com/AlDanial/cloc

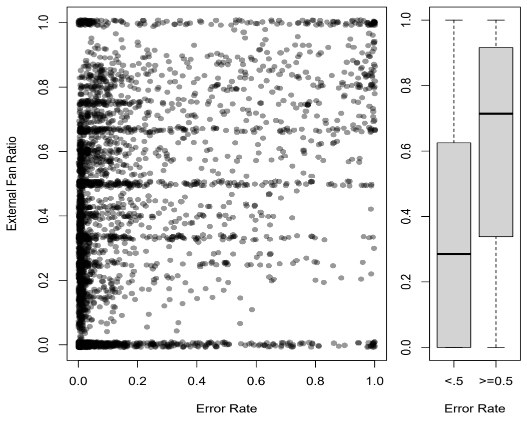

Figure 5: The error rate versus the external fan ratio of each entity in large modules. Coordinates are jittered for clarity. The box plot shows the difference in the external fan ratio of non-problematic (error rate < 0.5) and problematic entities.

Finally, we investigate the difference in error between the versions. All entities move to a lower error rate in JabRef v3.7, and 11 entities have a problematic mapping in JabRef 3.5 (cf. Figure 6). JabRef 3.5 has 36 entities mapped to non-small modules with problematic error rates. Between 4 and 11 of these seem to be due to problems in the ground truth mappings, i.e., 11.1% and 30.6%. Optimally, these are cases where a technique would alert and spark a discussion regarding the ground truth mappings among the developers.

It should be noted that JabRef underwent a refactoring towards a new modular architecture at this point in development. Therefore, we do not think that these relatively high percentages are representative of all software systems. The developers likely have a higher degree of conformance in a more stable architecture.

Lastly, we look at architecturally refactored entities between JabRef 3.5 and JabRef 3.7. We define an architecturally refactored entity as an entity that has changed mapping and package. We view the conscious choice to change the package of an entity as a sign that the change in mapping is not a mistake or disagreement but a part of architectural evolution. We find 61 such entities, 5 of which are mapped to a small module in one or both versions. We find that refactored entities have a significantly higher error rate if we compare the error rate of these entities with both new and normal entities from the majority of modules in JabRef 3.7 (cf. Figure 7).

Such refactored entities could still be in a state of transition, and it seems likely to be a practice to make the change of package and mapping before changing major parts of the implementation. An architectural refactoring can also change the purpose of a module itself, though this will be a slower process for an automatic mapper to

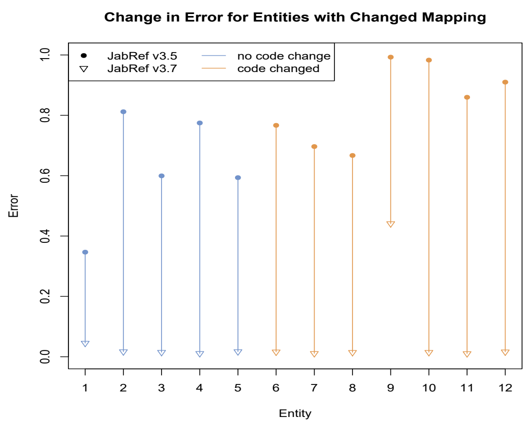

Figure 6: The change in the error of 12 entities from large modules in JabRef that have changed mapping but not changed package. The first five (blue) entities have had no change in source code and the last seven (orange) have changed source code.

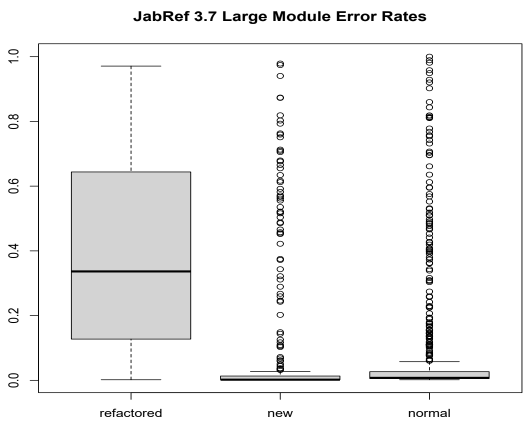

Figure 7: The error rates of entities in large modules that are undergoing refactoring, are new, or normal in JabRef 3.7.

detect. The risk is that a module can be quite chaotic during a transition phase with multiple entities in different stages of the refactoring process.

Another interesting observation is that new files tend to have a lower error rate, indicating that the developers have understood the new architecture and that normal code changes could slowly make an entity harder to classify. This could be due to some form of design erosion, where changes are introduced that make the entity less cohesive over time.

5. Related Work

There is previous work in the area of orphan adoption [9, 10, 8, 7, 12, 15, 16, 17]. The focus is to evaluate and compare the performance of different approaches, but not to specifically analyze problematic cases. We highlight the conclusions of prior work made regarding what may explain the performance.

The orphan adoption criteria naming, structure, style, and interface minimization are used in an algorithm evaluated in three case studies [9]. We find an evolving industrial system where the architecture was created by researchers with the help of developers the most interesting of these. 939 entities were assigned to modules, and in 46 cases (4.9%), the algorithm suggested a different mapping than the developers. In 33 of these cases, the developers agreed with the algorithm's mapping, i.e., the algorithm was able to find developer mistakes. In some of the 13 cases where the suggested module was not accepted, the developers mentioned that (code) changes to the entity were needed for it to conform to the developer mapping.

Bibi et al. compared the structural criteria part of the algorithm proposed by Tzerpos and Holt with supervised machine-learning approaches; Bayesian classification, k-nearest-neighbor, and neural networks. Their study focuses on using dependencies as features (i.e., structural criteria) for incremental clustering. They evaluate the approaches using two versions of six open-source software systems and find that dependencies between entities within the same module are important to avoid misclassifications, especially when there are few dependencies between entities in different modules.

We previously constructed a structure-based heuristic for automatic mapping of source code to Model-View-Controller-based architectures [15]. We evaluated the approach on four products in a product line of games, all using the same game engine. We compared the automatic mapping to the manual mapping, and if they disagreed, then the type was flagged as containing an architectural problem. We compared the mappings of 653 entities and were able to correctly identify 76 out of 101 architectural problems as well as 18 false positives. The heuristic suggested a different mapping in 96 (14.7%) of 653 cases.

Furthermore, two of the projects were refactored to be fully conformant. This refactoring removed 33 true positives and six false positives. The true positives were remedied by refactoring the source code. In the context of evaluating the performance of a method for automatic mapping using the manual mappings as ground truth, these true positives would be regarded as erroneous mappings when they, in fact, are pointing to source code with architectural problems that need to be refactored.

The CountAttract and MQAttract attraction functions of HuGMe have been evaluated in four case studies [10, 8]. The focus is on evaluating the influence of two configuration parameters and comparing the performance of the attraction functions. Both attraction functions assume a modular design based on the high cohesion low coupling style, and mapping would become problematic for modules designed specifically to not use this style. Christl et al. suggest the incorporating a detection step to better handle such modules, which would correspond to handling the style criteria. Furthermore, Chen et al. improves on CountAttract in an evolutionary case, i.e., a pre-existing mapping is used.

Bittencourt et al. present two new attraction functions based on information retrieval techniques. They use the semantic information in the source code and calculate attractions based on cosine similarity (IRAttract) and latent semantic indexing (LSIAttract). They make a quantitative comparison between the performance of their attraction functions with CountAttract and MQAttract in an evolutionary setting (where a few new files are to be assigned a mapping). They find that a combination of attraction functions (e.g., if CountAttract fails, then try IRAttract) performs best. This is explained by their qualitative analysis, where they find that CountAttract usually misplaces entities on module borders, MQAttract performs better when mapping entities with dependencies to many different modules, IRAttract and LSIAttract perform better when mapping entities in libraries or entities on module borders, but perform less well if there are modules that share vocabulary but are not related.

Sinkala and Herold present InMap, which is not an automated approach to mapping per se; instead, InMap suggests mappings to the end-user, who can choose to accept the suggested mapping (or not). It is an iterative approach where a number of mappings are presented, and the accepted mappings are used to improve the suggested mappings further. The suggested mappings are produced with the help of information retrieval information similar to Bittencourt et al. with the addition of a descriptive text for each architectural module. The entities are treated as a database of documents, and InMap uses Lucene to search this database using module information as a query. As InMap is highly interactive, it will also use negative evidence to some degree, i.e., a rejected mapping suggestion will not be suggested again. The data from [16] suggest that using only the module names as a search criterion often results in high precision at the expense of the recall. This is most likely due to the fact that module names often reflect package names to some degree. Adding more and more module information in the query tends to lower precision, but increase the recall, e.g., source code comments increase recall but lower precision in the mapping suggestions.

Garcia et al. discuss the use of package and naming information in software architecture recovery [18]. In general, they found that their ground truth components often spanned or shared several packages. They could not find a correlation between components and single package or directory names. One of their four cases presented a fairly good correlation, and in one system, they could find a repeating pattern of directories. Possibly the ground truth architectures recovered in their study is more low level than the modular architectures that we study. Still, it is likely that there is a variation on what dimension of an architecture that is expressed in the package structure. This is further supported by Buckley et al. where one system of five studied did not have any clear correlation between packages and modules [19], presenting clear difficulties and significant effort when performing the manual mapping.

6. Discussion and Validity

Our results clearly show that there is a set of entities in the systems that are systematically hard for the state-of-the-art automatic mapping techniques to map. One reason for this is the surprising result that all studied systems exhibit some very small modules. An automated technique would have very little data to use for these modules, lowering the chance for successful mapping.

In general, unbalanced data is problematic for machine learning techniques, and in particular, the distribution of probabilities is important in Naive Bayes. 30% (237 out of 784) problematic entities are mapped to such small modules in the ground truth mappings. This is a significant problem that needs to be solved.

In essence, small modules need to be flagged (either automatically or manually) and handled separately. One idea in the context of Naive Bayes would be to manipulate the probability distribution appropriately to not wholly disregard small modules in the mapping. Schemes that could be tested are a uniform distribution or different fixed settings (large, medium, small). These should be reasonably easy for an end-user to assign to a module.

Still, there is a risk that overall performance will drop as potentially more entities will be hard to map. Another approach is to investigate why a few of the small modules do not contain many problematic entities. We suspect that these modules possibly exhibit a unique design, e.g., being very cohesive or having very clear naming, which is perhaps not easy to address directly in a technique as it may simply be a way a module is designed.

Using the naming strategy and possible ambiguity in naming is an attractive approach to create an initial set using a specialized mapper. It should be possible to prompt an end-user with, e.g., keywords from the package or class name asking for a mapping of the keyword. This could significantly reduce the effort of creating an initial set that could then be used as a basis for other mapping techniques. However, a complete approach must be prepared to handle subject systems where the naming information does not reflect the modular architecture.

The data on finding problematic entities among entities that lie on the borders of modules is conflicting. On the one hand, we cannot see any correlation between the external fan ratio and the error rate. On the other hand, we observe a higher median external fan ratio in problematic entities. We observe very high error rates in combination with very low external fan ratios and vice versa. This indicates that the external fan ratio is not a useful metric, and a more refined metric could give better answers. There is possibly a difference between incoming and outgoing dependencies that could be a factor. In [9], these entities were specifically detected and only used when suggesting a new module (orphan kidnapping). Such an approach could also be investigated.

We studied two different mappings in two versions of JabRef and found six cases where only the mapping had changed (no change of source code), of which five mapped to large modules. We found eleven entities where the mapping and source code had changed (though the entity had not changed package), of which seven were in large modules. For these entities, there was a significant difference in error rate between the two mappings. We are relatively confident that the difference in the six entities with no change is due to disagreement among the developers; in the other eleven, it could also be due to the actual change of the entities' source code. This would indicate that between 0.8% and 2.3% of entities are hard to map correctly, even for JabRef experts.

It should also be noted that JabRef is only one case and that it was undergoing architectural refactoring during this time in development. We are reasonably confident that this affects the results. We can argue that there may be more confusion among the developers during refactoring, which should increase the chance of disagreements. There is also the possibility that the process of refactoring has brought the architecture to everyone's attention, possibly lowering the chance of disagreements. The low error rate of new entities suggests the latter as more likely.

The two mappings and versions of JabRef allow us to study entities under refactoring and new entities. We find 61 entities under refactoring and 348 new entities. If we remove entities from small modules (with confounding error rates), we find that entities under refactoring are considerably harder to map correctly. This is likely because architectural refactoring is a process that can take some time to complete. The functional aspects of the entities are likely fixed first, possibly with the removal of unwanted dependencies (especially as JabRef has some tests for this).

There is, however, a risk that the semantic information (e.g., variable names) will not be changed and correctly reflect the vocabulary of the module. It would be interesting to see if this happens to these entities in future versions of JabRef or if the current state is considered good enough. If so, there is a considerable risk that modules will become less semantically cohesive as the vocabulary becomes a mix of words from the previous architecture. The error rate of entities could then be used as a metric to know if an entity is properly aligned to other entities in the module.

Comparing a human-made mapping to the mapping made by an automatic technique seems to be a useful piece of information. The related work [9, 15] shows that this often points to cases where (further) refactoring or discussion is needed and that the automatic technique is not necessarily wrong per se. However, if no human mapping exists, is it important for an automated technique to notify a human user of such issues and not automatically assign the entity a mapping.

Comparing mappings using several different techniques could be a way forward, similar to what is done in [7] but with a different intent. This also points to a problematic situation as we cannot fully trust the ground truth mappings; a perfect mapping technique would thus be flawed. There is also a general lack of ground truth mappings made by human experts and even fewer mappings made by different experts on the same system. Four of the systems (JabRef v3.5, JabRef v3.7, ProM, and TeamMates) have mappings done by experts. The others (ArgoUML, Ant, Lucene, and Sweet Home 3D) have mappings created by researchers studying the systems' documentation and implementations [13]. The architects or developers of these systems would likely not agree to all of these mappings even if it is likely that large parts of the mappings are correct.

Two limiting factors in this study are that all systems are implemented in Java and that we have only studied one set of parameters of the attraction function, i.e., the one from [2] giving the best mapping performance. Another set of parameters would likely give different error rates; however, we think the main points of the paper would still hold.

7. Conclusions and Future Work

We investigate the flaws in the automatic mapping of source code to modules in eight open-source software systems. We show that the state of the art technique has systematic flaws in its suggested mappings that need to be addressed. We find that a major contributing factor is that all investigated systems have modules with very few ground truth mappings. We also find that all systems use a naming strategy, but this strategy is often ambiguous. We found no clear evidence that entities that have many dependencies to or from entities in other modules are systematically problematic. Our data indicate that such dependencies can be a factor, but the metrics used are likely not well suited to clearly show such problems.

We studied differences in expert mappings in one of the systems, where we had two different versions and two different ground truths. We found that disagreements exist and that such entities are likely to have a high error rate in the mappings, although there are not many such entities. We also studied refactored files and new entities. Refactored entities tend to have a significantly higher error rate compared to both new entities and normal entities. There is a risk that refactoring is considered done when the entity is moved and the functional aspects are fixed. Automatic mapping could indicate when the entity is properly aligned to other entities in the module or noticeably different.

Our priority for the future is to address the small modules. We will try different approaches to manipulating the probability distribution of the modules and find the effect on overall mapping performance. Another area of interest is the use of naming information to create an initial set, as this could significantly reduce the mapping effort.

Acknowledgments

The research was supported by the Centre for Data Intensive Sciences and Applications at Linnaeus University.

References

[1] T. Olsson, M. Ericsson, A. Wingkvist, Towards improved initial mapping in semi automatic clustering, in: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA '18, 2018, pp. 51:1–51:7.

[2] T. Olsson, M. Ericsson, A. Wingkvist, Semi-automatic mapping of source code using naive bayes, in: 13th European Conference on Software Architecture - Volume 2, 2019, p. 209–216.

[3] L. De Silva, D. Balasubramaniam, Controlling software architecture erosion: A survey, Journal of Systems and Software 85 (2012) 132–151.

[4] G. C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: Bridging the gap between source and high-level models, ACM SIGSOFT Software Engineering Notes 20 (1995) 18–28.

[5] N. Ali, S. Baker, R. O'Crowley, S. Herold, J. Buckley, Architecture consistency: State of the practice, challenges and requirements, Empirical Software Engineering 23 (2017) 1–35.

[6] J. Knodel, D. Popescu, A comparison of static architecture compliance checking approaches, in: The IEEE/IFIP Working Conference on Software Architecture, 2007, pp. 12–21.

[7] R. A. Bittencourt, G. Jansen de Souza Santos, D. D. S. Guerrero, G. C. Murphy, Improving automated mapping in reflexion models using information retrieval techniques, in: IEEE Working Conference on Reverse Engineering, 2010, pp. 163–172.

[8] A. Christl, R. Koschke, M. A. Storey, Automated clustering to support the reflexion method, Information and Software Technology 49 (2007) 255–274.

[9] V. Tzerpos, R. C. Holt, The orphan adoption problem in architecture maintenance, in: IEEE Working Conference on Reverse Engineering, 1997, pp. 76–82.

[10] A. Christl, R. Koschke, M. A. Storey, Equipping the reflexion method with automated clustering, in: IEEE Working Conference on Reverse Engineering, 2005, pp. 98–108.

[11] T. Olsson, M. Ericsson, A. Wingkvist, s4rdm3x: A tool suite to explore code to architecture mapping techniques, Journal of Open Source Software 6 (2021) 2791. doi:10.21105/joss.02791.

[12] M. Bibi, O. Maqbool, J. Kanwal, Supervised learning for orphan adoption problem in software architecture recovery, Malaysian Journal of Computer Science 29 (2016) 287–313.

[13] J. Brunet, R. A. Bittencourt, D. Serey, J. Figueiredo, On the evolutionary nature of architectural violations, in: IEEE Working Conference on Reverse Engineering, 2012, pp. 257–266.

[14] J. Lenhard, M. Blom, S. Herold, Exploring the suitability of source code metrics for indicating architectural inconsistencies, Software Quality Journal (2018).

[15] T. Olsson, D. Toll, A. Wingkvist, M. Ericsson, Evaluation of a static architectural conformance checking method in a line of computer games, in: 10th international ACM Sigsoft conference on Quality of software architectures, ACM, 2014, pp. 113–118.

[16] Z. T. Sinkala, S. Herold, Inmap: Automated interactive code-to-architecture mapping recommendations, in: IEEE 18th International Conference on Software Architecture (ICSA), 2021, pp. 173–183.

[17] F. Chen, L. Zhang, X. Lian, An improved mapping method for automated consistency check between software architecture and source code, in: IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020, pp. 60–71.

[18] J. Garcia, I. Krka, C. Mattmann, N. Medvidovic, Obtaining ground-truth software architectures, in: 35th International Conference on Software Engineering (ICSE), 2013, pp. 901–910.

[19] J. Buckley, N. Ali, M. English, J. Rosik, S. Herold, Real-time reflexion modelling in architecture reconciliation: A multi case study, Information and Software Technology 61 (2015) 107–123.