Towards Hierarchical Code-to-Architecture Mapping Using Information Retrieval

Zipani Tom Sinkala [1], Sebastian Herold [1]

[1] *Department of Mathematics and Computer Science, Karlstad University, Karlstad, Sweden*

Abstract

Automating the mapping of a system's code to its architecture helps improve the adoption of successful *Software Architecture Consistency Checking (SACC)* methods like *Reflexion Modelling*. InMap is an interactive code-to-architecture mapping recommendation technique that has been shown to do this task with good recall and precision using natural language software architecture descriptions of the architectural modules. However, InMap like most other automated recommendations techniques maps low level source code units like source code files or classes to architectural modules. For large complex systems this can still be a barrier to adoption due to the effort required by a software architect when accepting or rejecting the recommendations. In this study we propose an extension to InMap that provides recommendations for higher-level source code units, that is, packages. It utilizes InMap's information retrieval capabilities, using minimal architecture documentation, applied to a software's codebase, to recommend mappings between the software's high-level source code entities and its architectural modules. We show that using our proposed hierarchical mapping technique we are able to reduce the effort required by the architect, as high as 6-fold in some cases, and still achieve good precision and fairly good recall.

Keywords

Automated Mapping, Software Architecture Consistency Checking, Information Retrieval.

1. Introduction

Mapping code to architecture is a task that is common in Software Architecture Consistency Checking (SACC) [1, 11, 14, 16]. Popular SACC methods like Reflexion Modelling [9, 12] require a mapping step in order to be able to identify conformance or divergence of a system's code to its intended software architectural modules [8, 9, 12, 13]. The mapping step is a manual and labour-intensive task for the most part that becomes a barrier to industry adoption of effective SACC techniques like Reflexion Modelling especially for large complex software systems [1, 7].

There have been a number of techniques that have been created that attempt to decrease the burden of mapping on software architects by automating the mapping step [4–6, 11, 15, 16]. Most of these however, are class- or file-based [11, 15, 16]. This implies, in the case of systems developed using an object oriented programming language, where classes are considered as the underlying unit of source code, they automate mapping at a class level – attempting to predict which architectural module, a class (or class-file) maps to. This has been done quite well with techniques like InMap [15, 16] and NBC [11]. In our paper "InMap: Automated Interactive Code-to-Architecture Mapping Recommendations" we show that InMap achieved a recall of 0.87-1.00 and precision of 0.70-0.96 for the systems tested.

However, in a large system of say a 1000+ classes, in spite of achieving recall and precision of 1, it is still burdensome for an architect to inspect over a thousand recommendations before accepting them as correct. In an attempt to reduce the effort needed, we investigate making mapping recommendations for higher-level source code units – that is, we make mapping recommend-

ECSA2021 Companion Volume
EMAIL: tom.sinkala@kau.se (Z.T. Sinkala); sebastian.herold@kau.se (S. Herold)

dations for larger units of code at a time (packages rather than classes) thereby reducing the amount of work required by an architect. In this paper, we present an automated hierarchical package mapping technique. It garners from the successful information retrieval-based InMap approach [15, 16] that computes similarity of an unmapped class to an architectural module. We exploit class-to-module similarity scores produced by InMap to generate package-to-module similarity scores. These are filtered using a defined set of hueristics from which recommendations, that are detemined by a system's package hierachy, are made. We show that using our proposed hierarchical mapping technique we are able to reduce the effort required by the architect, as high as 6-fold in some cases, and still achieve good precision.

Section 2 briefly discusses automated mapping techniques along with their hierarchical mapping capabilities. In Section 3, we detail the approach, describing how package scores are computed and how package-to-module mapping recommenddations are constructed. Section 4 describes the experiment setup to evaluate the technique and presents the results obtained. In Section 5, we interpret and discuss the results and in Section 6 we draw our conclusions on our findings and present opportunities for further research.

2. Related Work

Christl et al. conceived, HuGME, a dependency analysis (DA) based automated mapping recommendation technique. It clusters a software system's source code using an architect's knowledge about its intended architecture [4, 5]. HuGME applies an attraction function, which minimizes coupling and maximizes cohesion, to produce a matrix of attraction scores for unmapped entities to modules [17]. The calculation of the score uses the dependency values between unmapped entities and mapped entities. The higher the score, the higher the likelihood that an unmapped entity belongs to a given module. All unmapped entities that result in only one candidate having a similarity score higher than the arithmetic mean of all scores produce a single recommendation. All unmapped entities for which two or more candidates exist are presented to the user in ranked order, from highest to lowest, as recommenddations. HuGME presents recommendations to the user to allow cluster decisions to be made exclusively by the architect. This process is incremental, in that HuGME does not attempt to map all source code entities in one complete step; rather it maps a subset at a time until no more mapping is possible. The approach is *non-hierarchical* as it views the mapping task from a clustering perspective in which source code entities that are mapped to the same hypothesized entity form a cluster [4].

In their study, the results for HuGME had on average about 90% recall and 80-90% accuracy [5]. To get these results the technique needed about 20% of the system's source entities to be pre-mapped before running the algorithm. Of interest is that because this mapping technique is dependency-based, for it to give meaningful results, the 20% pre-mapped source entities need to be spread across various modules. In addition, they must have dependencies to unmapped entities. This presents a problem in that in order to benefit from this technique one needs to not only dedicate some time for pre-mapping but must also ensure that the mapping is evenly spread across the modules. Additionally, one must also ensure that the selected pre-mapped source code entities have dependencies to the unmapped entities otherwise entity relationship discovery is poor. This all becomes a highly labour-intensive exercise. Furthermore, because it uses clustering algorithms based on high cohesion and low coupling, if developers do not follow this principle in the software's implementation then the mapping of the algorithm will be affected [2].

Bittencourt et al. propose an information retrieval (IR) based technique that uses the same automated mapping recommendations approach as HuGME except it replaces dependency-based attraction functions with IR based similarity functions [3]. It calculates the similarity of an unmapped source entity to a module by searching for specific terms (a module's name and mapped classes, methods and fields) within the source code of the unmapped class. Similar to HuGME, Bittencourt et al.'s technique needs some manual pre-mapping before it can automate mapping.

Olsson et al. combine IR & DA methods in their automated mapping technique called Naive Bayes Classification (NBC) [11]. NBC uses Bayes' theorem to build a probabilistic model of classifications using words taken from the source code entities. The model gives the probability of words belonging to a source file entity. This is augmented with syntactical information of the dependencies, a method called Concrete Dependency Abstraction [11]. Just like HuGME, Olsson et al.'s proposed technique requires a pre-mapped set in order to perform well. Both Bittencourt et al.'s and Olsson et al.'s results showed that when there was a smaller pre-mapped set there was a decreasing trend in the f_1-score of their techniques [3, 11]. Additionally, they both do not address package-level based mapping.

Naim et al. present a technique called Coordinated Clustering of Heterogeneous Datasets (CCHD), that combines both DA and IR methods to compute a similarity score for source code files [10]. CCHD uses an architect's feedback on the recovered architecture to iteratively adjust the results until there are no suggestions for change. These adjusted results train a classifier that automatically places new code added to a codebase in the "right" architectural module. However, the technique is not necessarily meant for automated mapping in SACC but rather for software architecture recovery tasks. Moreover, it too does not directly address package-level based mapping.

Common among industry tools is the use of naming patterns (or regular expressions). For example, the expressions ***/gui/*** or **.gui.* or net.java.gui.** can be used to map source code units (whether classes or packages) to an architecture module named *GUI*. This is the technique used by both **Sonargraph Architect** and **Structure101 Studio** in addition to their drag & drop capabilities. However, the drawback of using naming patterns and/or drag & drop functionality is that they are both manual tasks which makes mapping a tedious exercise – especially for large software systems that have complex mapping configurations.

In summary, despite advances made, available techniques that are designed to automate mapping have short comings. Some require an initial set of the source code to be pre-mapped manually [3–5, 11], while the industry tools that do not require pre-mapping offer manual methods. Additionally, the automated mapping techniques that require pre-mapping in order to "jump-start" mapping, as it were, require about 15-20% of the source code to be pre-mapped in order to give worthwhile results [4, 5, 6, 15].

InMap [15, 16] addresses the limitations of these techniques in that it is able to automate mapping without requiring pre-mapping. Using simple and concise natural language descriptions of the architecture modules it is able to automate mapping of a completely unmapped system with rather good results. Its limitation though is that the mapping recommendations provided are for low-level source code units, namely, classes. This results in considerable work for an architect in the case of large software systems. We therefore explore the following research question:

How can we exploit InMap's good class-to-module mappings to produce package-to-module mappings, thereby reducing the effort needed by an architect in accepting and/or rejecting mapping recommendations produced by InMap?

In the following section, we describe our approach to answering this question.

3. Approach

We begin by describing the InMap technique briefly. We then describe a technique for hierarchical package-to-module mapping that builds on top of InMap.

3.1. InMap

InMap is an interactive code-to-architecture automated mapping technique for SACC methods that uses information retrieval concepts to produce class-to-module mapping recommendations. It does not require manual pre-mapping in order to produce recommendations, rather it uses natural language architectural descriptions of the architectural modules as input to predict mappings. It presents its best mapping recommendations a page/set at a time (the most optimal being 30 per page) from which the architect can accept and reject. As recommendations from each page/set are accepted or rejected, InMap learns from this and adapts its next page/set of recommendations from the obtained knowledge. This method works quite well giving an average recall of 97% and a precision of 82% for the systems evaluated [16].

3.1.1. Class-to-Module Similarity

InMap's algorithm is made up of *seven steps* [16]. However, for our hierarchical package-to-module mapping technique the following steps in InMap are used to generate what are called class-to-module mapping scores.

Firstly, the source code files are filtered to exclude any external or third-party package libraries or classes of system that the architect does not want to include in the mapping exercise.

Secondly, the filtered sourced files are stripped of any special characters and programming language keywords. *Third*, the pre-processed source code files are indexed as an inverted index. In the *fourth and fifth steps*, InMap formulates a query using four items namely, (1) the names of the modules and (2) the module's architectural descriptions (stripped of any special characters and stop words) to search the indexed source code files for similarity to each module. In the first iteration, InMap uses this information only to build a query. However, once the first set of classes are mapped, InMap then adds to the query (3) the names of classes mapped to a module and (4) the names of methods contained within classes mapped to a module. This 'enriches' the query used to search for the similarity of an unmapped class to a module. Therefore, after each set of newly mapped classes the query for the next set of recommendations looks different. The search returns a set of scores for every class-module pair based on the similarity information retrieval function, *tf-idf*. The *tf-idf* scores are called *class-to-module similarity scores* (SS_{cm}), where, *c* and *m* are a class-module pair in the system. Specifics of how *tf-idf* is calculated can be found in [16].

3.1.2. Class-to-Module Mapping Recommendations

In the *sixth and seventh steps*, InMap gives as a *class-to-module mapping recommendation* the highest scoring class-to-module pair. The architect can either accept or reject it. However, InMap presents as recommendations either: only those above the arithmetic mean of all highest scoring class-module pairs; or the best 30 recommendations (if those above the mean is greater than 30). After the architect gives feedback, it returns to step 4 and repeats steps 4 to 7 until no more recommendations can be given.

Our proposed hierarchical package mapping technique picks up right after the fifth step, that is, once InMap produces the matrix of *class-to-module similarity scores* (SS_{cm}).

3.2. Hierarchical Package Mapping

In as much as InMap is able to achieve good results with the approach described in Section 3.1 because it based on class module mappings, the effort required by architects could still be significant for large and complex systems. However, if we could map entire packages then we could reduce the effort needed. For example, a package that has 50 classes that all map to the same module could be (or should be) given as a single package-to-module mapping recommend-dation. Additionally, because packages are hierarchal in nature, they present even more opportunity to reduce the number of "necessary" mapping recommendations to present to an architect. For example, say we have two packages A and B that are both sub-packages of C. If A and B have 50 classes each and say all the classes in A and B map to the same module. Then mapping C to the module would suffice and saves the architect from reviewing 99 other mapping recommendations. Figure 1 illustrates a package hierarchy, that our technique (and certainly others) can benefit from to reduce the number of recommendations needed.

3.2.1. Package-to-Module Similarity

Our package-to-module mapping technique picks up from step 5 of the InMap algorithm after it produces similarity scores for all class-to-module pairs. We group the *class-to-module similarity scores* SS_{cm}, according to the packages they belong to. This means for each package we have a set of classes with scores to each identified module. From this set of *class-to-module similarity scores* that have a given package as their parent we then calculate the *interquartile mean* (IQM_{pm}), where, p and m are a package-module pair in the system. That is, the range of values between the first quartile and third quartile (the *interquartile range*, *IQR*) are used to calculate the arithmetic mean. Module *IQRs* for a package taken from Jittac are demonstrated in *Figure 2.* The lowest 25% and the highest 25% of the scores are ignored. Important to note is that the *IQR* and hence the *IQM* of a non-terminal package is calculated from not only the classes that belong to the package but also the classes of its child packages. For example, *se.kau.cs.jittac.eclipse.b-uilders.jdt* shown in the package tree in *Figure 1* has its *IQR* calculated using the 8 classes that belong to it but also the 3 classes in *s-e.kau.cs.jittac.eclipse.builders.jdt.commands* and the single class in *se.kau.cs.jittac.eclipse.buil-ders.jdt.util.* Formally, we define IQM_{pm} as,

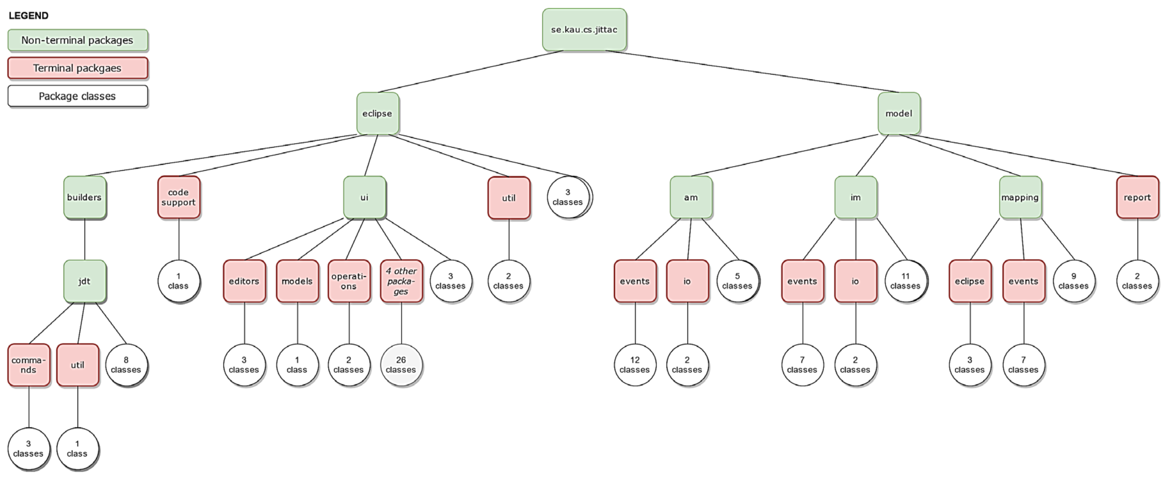


Figure 2: Package hierarchy for Jittac - one of the systems we evaluate our technique on.

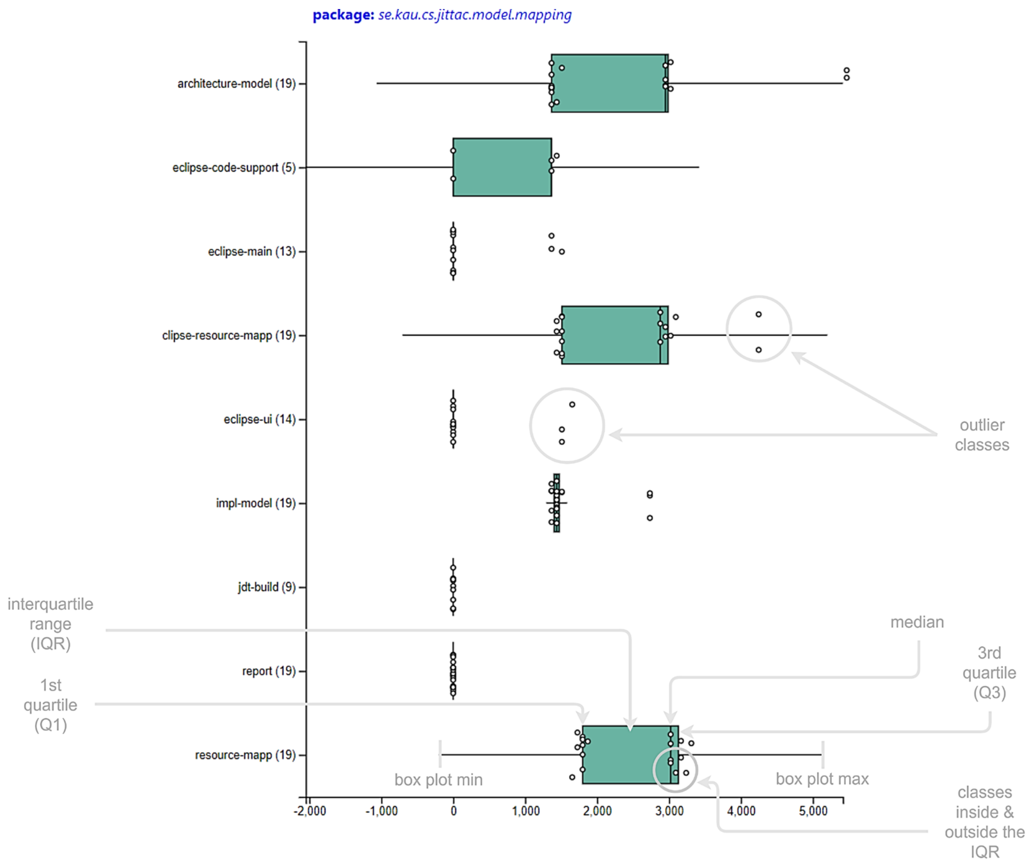


Figure 1: Box plots for a package taken from Jittac showing the *IQRs* for Jittac's modules as well as the class distribution inside and outside the *IQRs*. The x-axis shows the class SS_{cm} scores and the y-axis shows the architectural modules of the system. The number in brackets beside a module indicates the total number of classes for the given package that have an SS_{cm} score to the module.

$$IQM_{pm} = \frac{2}{n} \sum_{i=\frac{n}{4}+1}^{\frac{3}{4}n} SS_{cm_i} \qquad (1)$$

where, p and m are a package-module pair in the system, c has p as its parent package, n is the number of classes that make up the package p and i is the position of SS_{cm} in the ordered set of *class-to-module similarity scores* for the package p.

Using the scores within the *IQR* as opposed to the full set of scores makes a package-to-module similarity, more resilient to the presence of outlier classes in the *class-to-module similarity scores* that it is derived from. *Figure 2* shows outlier

Table 1

Extract of $\boldsymbol{SS_{pm}}$ scores taken from Jittac. A value >= 0.6 (highlighted blue) implies it is a *good package-to-module similarity score*; a score >= 1.5 (highlighted red) implies it is an *outstanding package-to-module similarity score*.

Packages	Modules		
	architecture-model	eclipse-ui	impl-model
se.kau.cs.jittac.model	2.3	-0.6	1.0
se.kau.cs.jittac.model.am	2.6	-0.4	0.4
se.kau.cs.jittac.model.am.events	2.4	-0.4	0.3
se.kau.cs.jittac.model.am.io	2.3	-0.5	0.6
se.kau.cs.jittac.model.im	1.0	-0.5	2.3
se.kau.cs.jittac.model.im.events	0.9	-0.6	1.6
se.kau.cs.jittac.model.im.io	0.5	-	1.6

classes which we define as classes with SS_{cm} scores that are higher than the *box plot max*, classes with SS_{cm} scores that are lower than the *box plot min* but also classes with SS_{cm} scores that are within the *box plot min-max* but outside the *IQR*. The result of this step is a matrix of *IQMs* for each package-module combination.

We then apply feature scaling to normalize the *IQM* module scores for each package. We use *standardization* (also known as *z-score normalization*) which makes the scores for each package-module pair have a zero-mean. In our hierarchical package mapping technique we call the resulting z-scores of the standardization normalization *package-to-module similarity scores* (SS_{pm}). Formally we define SS_{pm} as follows,

$$SS_{pm} = \frac{IQM_{pm} - average(IQM_{pm})}{\sigma} \quad (2)$$

where, p and m are a package-module pair in the system, IQM_{pm} is the original *package-to-module similarity score*, $average(IQM_{pm})$ is the mean of the IQM_{pm} scores for a specific package to the range of given modules, and σ is the standard deviation of IQM_{pm}. Using this method on all package module pairs we obtain a matrix of *package-to-module similarity scores* for the entire system. *Table 1* shows an extract of these scores.

3.2.2. Package Mapping Filtering

Using the matrix of *package-to-module similarity scores* we then traverse the package-tree bottom-up starting with the terminal packages and working our way up to the root package. At each tree-depth level we retain the *package-to-module similarity scores* into two sets for each package, namely a *set of outstanding package-to-module similarity scores* and a *set of good package-to-module similarity scores*. Outstanding mappings are those in which a package has a score above the outstanding threshold and its child packages have a score above the good threshold. Good mappings are those in which a package and its children have a score above the good threshold. We formally define this notion with the following two rules,

Given:
Package **p**
Module **m**
Package-to-module score $\mathbf{SS_{pm}}$
Good score threshold **GSt**
Outstanding score threshold **OSt**

Rule 1: A mapping ($\mathbf{p_i m}$) is called good iff

$$\mathbf{SS_{pm} >= GSt}$$

and for all sub-packages $\mathbf{p_i}$ of **p**, $\mathbf{p_i m}$ is a good mapping.

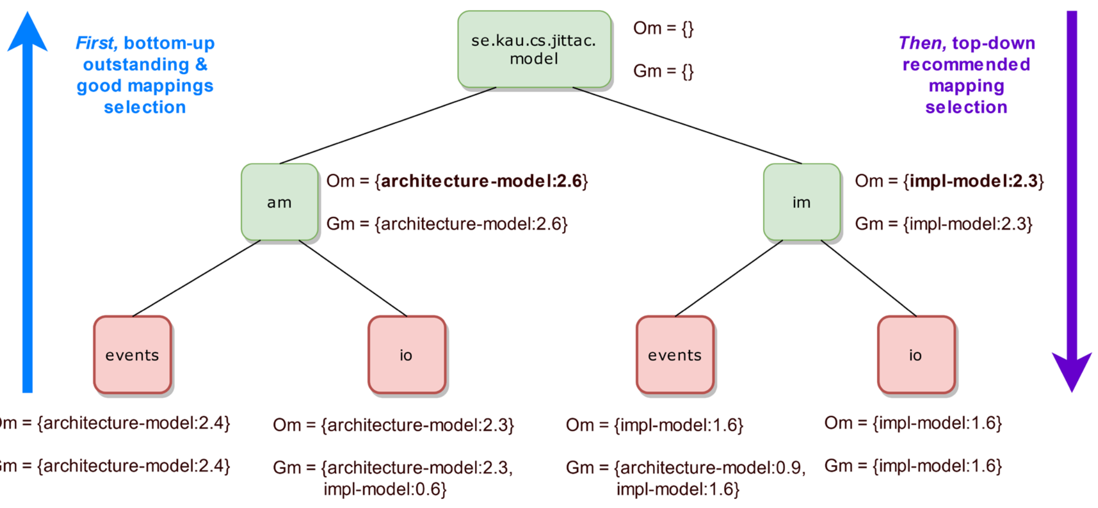
Rule 2: A mapping ($\mathbf{p_i m}$) is called outstanding iff

$$\mathbf{SS_{pm} >= OSt}$$

and for all sub-packages $\mathbf{p_i}$ of **p**, $\mathbf{p_i m}$ is a good mapping.

Figure 3 illustrates the rules with an example using SS_{pm} scores shown in *Table 1*. You will notice that despite the package *se.kau.cs.jittac.model* having good and outstanding scores for the modules *impl-model* and *architecture-model* respectively in *Table 1, Figure 3* indicates that the package has no good or outstanding mappings. This is because it fails to satisfy the second part of *Rule 2*, that is, that all its sub-packages must have good mappings to the same module. However, one of *se.kau.cs.jittac.model*'s sub-packages has a good mapping to the same module but the other does not hence no good or outstanding mappings for the *se.kau.cs.jittac.model* package.

These rules are applied from the bottom of the package tree starting with the deepest terminal packages then their parent packages, then their grandparent packages and so on and so forth until we reach the root package at the top of the tree.



First, bottom-up outstanding & good mappings selection
Then, top-down recommended mapping selection
se.kau.cs.jittac.model
Om = {}
Gm = {}
am
Om = {architecture-model:2.6}
Gm = {architecture-model:2.6}
im
Om = {impl-model:2.3}
Gm = {impl-model:2.3}
events
Om = {architecture-model:2.4}
Gm = {architecture-model:2.4}
io
Om = {architecture-model:2.3}
Gm = {architecture-model:2.3, impl-model:0.6}
events
Om = {impl-model:1.6}
Gm = {architecture-model:0.9, impl-model:1.6}
io
Om = {impl-model:1.6}
Gm = {impl-model:1.6}

Figure 3: Package tree traversal in order to produce *package-to-module mappings recommendations*.

This is necessary as packages higher up in the package tree depend on the results of packages lower in the package tree.

3.2.3. Package-to-Module Mapping Recommendation Selection

Once both sets of good and outstanding mappings for each package are obtained, we then traverse the package tree top-down. At each tree-level we check if a package has outstanding mappings and pick the highest that fulfils the above defined criteria for outstanding and recommend it as the most likely mapping. If a package is recommended then we terminate following that tree path downwards and do not recommend any of its sub-packages, we instead proceed to check its siblings. If a package returns an empty set, then we go one-step lower in the package tree. *Figure 3* illustrates this; it shows two *package-to-module mapping recommend-dations* (in bold). Observe that *architecture-model* is recommended as the module to which se.kau.cs.jittac.model.am should map to and *impl-model* as the module to which *se.kau.cs.jittac. model.im* should map to. Their sub-packages are skipped since they are already considered as a result of *Rule 2* and *se.kau.cs.jittac.model* has no mapping recommendation since it retained no mappings after the package mapping score filtering step.

4. Evaluation

Test Cases: We evaluated our hierarchical package mapping approach on six Java-based systems that were used in the evaluation of InMap’s class-to-module mapping technique. These are *Ant*, a command line and API-based tool for process automation; *ArgoUML*, a desktop-based application for UML modelling; *JabRef* a desktop-based bibliographic reference manager; *Jittac* an eclipse plugin for reflexion modelling tasks; *ProM* a desktop-based processes mining tool; and *TeamMates* a web-based application for handling peer reviews and feedback. Table 2 shows the attributes of these systems. The natural language architectural module descriptions used as input to InMap to generate *class-module similarity scores* were obtained from the previous study of InMap. The prior study of InMap obtained the oracle mappings, that is the correct list of code-to-module mappings, from experts involved in developing each respective open-source project. The oracle package-to-module mappings used in this study were extracted from these. We retained in the oracle only packages that had direct 1-1 mappings with a module, and excluded packages that had child entities that map to more than one module.

From the oracle mappings we only extracted package-to-module mappings, leaving out the class-to-module mappings to allow us to evaluate the performance of proposed technique strictly at a package-level. Table 2 also shows the number of packages in the oracle mapping of a system. This is the number of actual packages our proposed technique should predict mappings for, in other words, the packages that are of concern. For example, if *se.kau.cs.jittac.eclipse* is part of the oracle mapping and our technique puts up *se.kau.cs.jittac.eclipse.builders* as a possible map-

Table 2

System Case Studies

System Attributes	Ant	Argo UML	JabRef	Jittac	ProM	Team Mates
Version #	r584500	r13713	3.7	0.1 (…)	6.9	5.11
# of source files	778	1,429	843	124	700	467
# of source files after filtering (# of classes)	724	763	840	110	699	293
# of packages	64	60	118	27	162	18
# of packages in oracle mapping	14	21	11	9	30	11
# of source files in oracle package mapping	558	692	812	98	675	293
# of modules	15	17	6	9	11	11

Table 3

Results showing the optimal thresholds for each system tested.

Test System	Good Thresh.	Oustand. Thresh.	# of Recomm.	Package Recall	Package Precision	Class Coverage
Ant	1.9	2.4	9	0.64	1.00	276/558 (50%)
ArgoU	0.1-1.6	3.2-3.3	13	0.43	0.69	93/692 (13%)
JabRef	1.2-1.4	1.6-2.0	6	0.55	1.00	794/812 (98%)
Jittac	1.0-1.4	1.7	7	0.67	0.86	88/98 (90%)
ProM	0.4-0.6	1.6	37	0.40	0.32	58/675 (9%)
TeamM	1.3-1.4	0.1-1.2	10	1.00	1.00	293/293 (100%)

Table 4

Effort comparison of class vs package mappings for systems with class coverage >= 50%.

Test System	Class Mapping		Package Mapping		Effort saved (*Effort reduced*)
	Class Coverage after …	# of Recomm.	Class Coverage after 1 pass	# of Recomm.	
Ant	*13 passes*, 50%	390	50%	9	381 (*−97.7%*)
JabRef	*32 passes*, 98%	853	98%	6	847 (*−99.3%*)
Jittac	*7 passes*, 90%	123	90%	7	116 (*−94.3%*)
TeamM	*14 passes*, 97%	275	100%	10	265 (*−96.4%*)

ping we count this as a false positive even though the latter is a child package of the former. The reason is the technique must reduce the effort needed by an architect and therefore must be penalized for recommending child packages of a package that is already mapped (or should be).

Experimentation & Data Collection: To experiment on the test cases with various good and outstanding threshold combinations we extended the evaluator tool we developed in our previous studies of InMap to accommodate the evaluation of package-based mappings. Using the oracle architecture package-to-module mappings of each system the tool automatically simulates a "human architect" accepting and rejecting the recommendations produced.

For all possible single decimal combinations within the range -5.0 to 5.0 for the good and outstanding threshold we collected the recall of the package mappings as well the technique's precision. The min-max of the test range was based on the highest and lowest SS_{pm} scores obtained by all 6 systems. We also collected the number of recommendations it took to achieve the given recall & precision. Finally, we also collected the class coverage (or code reach), that is, the number of classes that were mapped as a result of their parent packages being mapped by our hierarchical mapping technique.

Results: Table 3 shows the results obtained for the optimal thresholds for each system, i.e. they gave the best results for the range of values tested. We got for three systems, Ant, JabRef and TeamMates, perfect precision with TeamMates getting the same for its recall and class coverage. We found 6 out of Jabref's 11 package-to-module mappings (as package recall) and 9 of Ant's 14, which resulted in class coverage of 98% and 50% respectively. For Jittac, 90% of its classes were mapped by finding 6 of it's 9 package-to-module mappings with a precision of 0.86. ArgoUML had fairly good precision but low recall resulting in low class coverage as well. ProM appeared to be an outlier obtaining poor precision and the lowest recall from the six systems tested. All results presented are for a single iteration (or pass) of the technique.

In Table 4 we compare the effort required by an architect of our hierarchical mapping technique vs InMap in its original form. We do this by looking at the class coverage of each technique and the number of recommendations an architect has to sift through to achieve the given class coverage. Table 4 shows this for the systems that achieved more than 50% class coverage after a single iteration. In simple terms we define the effort saved (ES) and the effort reduced (ER) as follows,

$$ES = |\, R^p - R^c \,| \tag{3}$$

$$ER = -100 \times \frac{ES}{R^c} \tag{4}$$

where R^c is the number of class-to-module recommendations needed by the InMap class-based technique and R^p is the number of package-to-module recommendations needed by our hierarchical package mapping technique. As an example, Table 4 shows that in the case of Ant it would take 390 recommendations to map 50% of Ant's classes using the InMap class-to-module technique, whereas it would take 9 recommendations to map 50% of Ant's classes using our heirarchical package-to-module mapping technique. You will also notice the effort saved is more than 800 recommendations for JabRef and the effort reduced is more than 90% for all 4 systems.

5. Discussion

Table 3 shows that the technique has almost perfect precision, 0.91 excluding ProM. This is likely due to the fact that our hierarchal package mapping technique is an extension of InMap's class-to-module similarity function. Using simple natural language descriptions of architecture modules the InMap algorithm, which has the *class-to-module similarity score* SS_{cm} function at its core, was shown to obtain rather good precision. Our hierarchical package mapping technique borrows from InMap's success by using the information retrieval based SS_{cm} to generate is own *package-to-module similarity score* SS_{pm}.

The package recall of our technique is fairly good considering that these results are obtained only after 1 iteration (or pass). As outlined in Section 3.1, InMap is an interactive-iterative technique that presents a set of recommendations at a time and progresses by learning from the feedback of the architect to formulate the next set of recommendations. However, the number of iterations (or passes) is proportional to the size of the system under review. Compare Tables 2, 3 and 4, observe that systems with a high number of source files require a high number of passes (or iterations) compared to the "smaller" systems. Table 3 shows that with our hierarchical mapping technique we are able to obtain a package recall of more than 50% in the first pass for 4 out of the 6 systems. Of these 4, from the first iteration we get 50% class coverage for Ant with the other 3 getting more than 90% class coverage. Despite this, two systems get low package recall and class coverage. We do not see this as a problem because it is resolved simply by having more package-mapping recommendation iterations which would still be far less compared to class-based mapping recommendation algorithms.

Table 3 shows the threshold values that give the optimal results for each system. However, we observed some similarities across the systems in our threshold values experiments. The optimal *outstanding score threshold* is very close to or the same as the arithmetic mean of the max package similarity scores for each module of the system. And the optimal *good score threshold* was usually 0.5 less than the optimal *outstanding score threshold.* This establishes a basis for developing an automated approach for deriving threshold values that will give good results across different systems.

Threats to Validity: Since our package-based technique is derived from InMap the external validity of its results is affected by similar things, that is, factors such as number of modules and classes, code commenting style and quality, and architecture description quality. Therefore, more cases studies with varying attributes would add to the validity of the results. However, the results of the six test systems used with varying attributes shown in Table 2 provide a compelling case for an automated hierarchical package mapping technique.

With regard to construct validity, the effort required by an architect using our technique needs to be evaluated against other package-based mapping methods provided by industry tools like drag & drop, naming patterns or regular expressions. For example, how does our hierarchical package-based technique compare with manually mapping packages? Evaluations such as these would require enhanced user studies with software architects in appropriately planned and controlled experiments.

6. Conclusion & Future Work

We have presented a proposed solution to hierarchical package-based mapping. It extends or builds on InMap, an information retrieval class-based mapping technique that uses concise natural language architectural descriptions of modules. Our hierarchical package-based mapping technique provides almost perfect precision and fairly good recall and great code coverage. But most importantly our techniques helps reduce the effort or workload required by an architect in accepting and rejecting mapping recommendations in interactive techniques like InMap. The technique is an improvement over the manual package mapping methods used in today's state-of-the-art reflexion modelling tools.

Despite reducing effort required, the drawback of using a purely package-based approach is that due to their 1-1 package-to-module mapping style these methods do not work well for systems that have more complex mapping configurations. It is not always the case that packages, and their members directly map to modules in a 1-1 manner. It is more likely the case that a software system's code-to-architecture mapping has a combination of both package and class mappings. Cases where package members are spread across multiple modules requires a class-based technique. Therefore, we plan as future work to derive an approach to combine InMap's good class-based approach with the good package hierarchy-based approach presented in this paper. The aim is to combine class and package mapping recommendations in a way that benefits from the advantages, and negates the disadvantages, of both mapping styles. Nevertheless, the hierarchical packaged/based mapping technique presented in this paper remains useful and is useful in cases where it is appropriate to map entire packages.

7. References

[1] Ali, N. et al. 2018. Architecture Consistency: State of the Practice, Challenges and Requirements. *Empirical Software Engineering*. 23, 1 (2018), 224–258. DOI:https://doi.org/10.1007/s10664-017-9515-3.

[2] Bauer, M. and Trifu, M. 2004. Architecture-aware adaptive clustering of OO systems. *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* (2004), 3–14.

[3] Bittencourt, R.A. et al. 2010. Improving automated mapping in reflexion models using information retrieval techniques. *Proceedings - Working Conference on Reverse Engineering, WCRE*. (2010), 163–172. DOI:https://doi.org/10.1109/WCRE.2010.26.

[4] Christl, A. et al. 2007. Automated Clustering to Support the Reflexion Method. *Information and Software Technology*. 49, 3 (2007), 255–274. DOI:https://doi.org/https://doi.org/10.1016/j.infsof.2006.10.015.

[5] Christl, A. et al. 2005. Equipping the reflexion method with automated clustering. *12th Working Conference on Reverse Engineering (WCRE'05)* (2005), 10 pp. – 98.

[6] Fontana, F.A. et al. 2016. Tool Support for Evaluating Architectural Debt of an Existing System: An Experience Report. *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2016), 1347–1349.

[7] Knodel, J. 2010. *Sustainable Structures in Software Implementations by Live Compliance Checking*.

[8] Knodel, J. and Popescu, D. 2007. A Comparison of Static Architecture Compliance Checking Approaches. *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture* (USA, 2007), 12.

[9] Murphy, G.C. et al. 2001. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. *IEEE Transactions on Software Engineering*. 27, 4 (Apr. 2001), 364–380. DOI:https://doi.org/10.1109/32.917525.

[10] Naim, S.M. et al. 2017. Reconstructing and Evolving Software Architectures Using a Coordinated Clustering Framework. *Automated Software Engineering*. 24, 3 (2017), 543–572. DOI:https://doi.org/10.1007/s10515-017-0211-8.

[11] Olsson, T. et al. 2019. Semi-Automatic Mapping of Source Code using Naive Bayes. *ACM International Conference Proceeding Series*. 2, (2019), 209–216. DOI:https://doi.org/10.1145/3344948.3344984.

[12] Passos, L. et al. 2010. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software*. 27, 5 (2010), 82–89. DOI:https://doi.org/10.1109/MS.2009.117.

[13] Rosik, J. et al. 2011. Assessing Architectural Drift in Commercial Software Development: A Case Study. *Softw., Pract. Exper.* 41, (2011), 63–86.

[14] de Silva, L. and Balasubramaniam, D. 2012. Controlling software architecture erosion: A survey. *Journal of Systems and Software*. 85, 1 (2012), 132–151. DOI:https://doi.org/https://doi.org/10.1016/j.jss.2011.07.036.

[15] Sinkala, Z.T. and Herold, S. 2021. InMap: Automated interactive code-to-architecture mapping. *Proceedings of the ACM Symposium on Applied Computing* (Mar. 2021), 1439–1442.

[16] Sinkala, Z.T. and Herold, S. 2021. InMap: Automated Interactive Code-to-Architecture Mapping Recommendations. *Proceedings - IEEE 18th International Conference on Software Architecture, ICSA 2021* (Mar. 2021), 173–183.

[17] Wiggerts, T.A. 1997. Using Clustering Algorithms in Legacy Systems Remodularization. *Proceedings of the Fourth Working Conference on Reverse Engineering* (1997), 33–43.