This is the accepted version of a paper presented at 36th ACM/SIGAPP Symposium on Applied Computing (SAC 2021).

Citation for the original published paper:

InMap: Automated Interactive Code-to-Architecture Mapping
In: Proceedings of the 36th Annual ACM Symposium on Applied Computing (pp. 1439-1442). Association for Computing Machinery
SAC ’21
https://doi.org/10.1145/3412841.3442124

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-83887
InMap: Automated Interactive Code-to-Architecture Mapping

Zipani Tom Sinkala  
Karlstad University  
Sweden  
tom.sinkala@kau.se

Sebastian Herold  
Karlstad University  
Sweden  
sebastian.herold@kau.se

ABSTRACT
There exist techniques that attempt to automate the mapping step in Reflexion Modelling, a popular software architecture consistency checking method. However, most of these techniques require an initial set of pre-mapped entities in order to execute automated mapping. This study proposes an interactive technique that is able to automate mapping without the need for pre-mapped entities. Using some minimal architecture documentation, the technique utilises information retrieval concepts applied to the software’s codebase to recommend mappings between the software’s source code entities and its architectural modules. The technique achieves higher f1-scores than existing techniques that require initial pre-mapping before automation.

CCS CONCEPTS
• Software and its engineering → Software architectures; • Information systems → Information retrieval

KEYWORDS
Source code mapping, reflexion modelling, software architecture consistency checking, software maintenance, information retrieval

1 INTRODUCTION
Reflexion Modelling is a software architecture consistency checking (SACC) technique used to manage software architecture degradation [10, 11]. It detects and visualizes the divergence between the intended and the as-implemented architecture as violations of the permitted dependencies among the modules of a software system [2]. For this purpose, the software’s source code entities need to be mapped to its architectural modules. The dependencies among the source code entities are then checked for conformance with the permitted module dependencies in its architecture. If the source code contains dependencies that go against the allowed architectural module dependencies, the software is said to have deviated from its intended architecture.

Reflexion Modelling is the most successful technique in terms of industrial adoption and tool support maturity [1]. However, the process of mapping source code to architectural modules is manual. Ali et al. highlight that practitioners cite “the huge effort...required to map the system to the architecture” as one of the main barriers to adoption of architecture consistency checking methods such as Reflexion Modelling [1]. There exist techniques that attempt to automate the mapping process. However, they still require a significant fraction of the source code to be initially pre-mapped manually by a software architect [3, 4]. There also exist industry tools that assist architects with reflexion modelling. Even so mapping is still a manual drag & drop task [5, 13] or via name patterns that are not versatile and do not scale well especially for large systems that have complex mapping configurations. In this paper, we present an interactive automated mapping technique, called InMap, which overcomes the pre-mapping constraint of existing mapping techniques and the limited approaches of existing reflexion modelling industry tools.

2 AUTOMATED MAPPING TECHNIQUES
Human Guided Mapping Generation Method (HuGME) is a dependency-based automated mapping technique that clusters a software system’s source code [3]. HuGME uses an attraction function (CountAttract or MQAttract) that produces a matrix of attraction scores for unmapped entities to modules [6]. The calculation of the score is based on 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. The reported results for HuGME showed on average about 90% recall and 80-90% accuracy. To get these results the technique needed about 20% of the system’s source entities to be pre-mapped before running the algorithm. Since it is using clustering algorithms assuming high cohesion within modules and low coupling between modules, the accuracy of mapping results for systems not following these principles may be low [7]. Axivion Architecture Analysis, a reflexion modelling tool used in industry, uses the HuGME technique for automated mapping [12].

Bittencourt et al. propose an information retrieval based technique similar to HuGME except it replaces the CountAttract and MQAttract attraction functions with IRAAttract and LSAAttract similarity functions [3, 8]. IRAAttract is a function that calculates the similarity of an unmapped source entity to a module by
searching for the module’s name and its classes, methods and fields within the source code of the unmapped class. LSIAttract, uses Latent Semantic Indexing (LSI) which introduces understanding of the meaning of a word or groups of words in natural language by grouping terms into concepts [8]. Olsson et al. use Bayes’ theorem of probability in a technique called Naive Bayes Classification (NBC) [4]. NBC builds 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. To derive the words, the technique retrieves stemmed package, file, attribute and variable names. This is then augmented with syntactical information of the dependencies. Just like HuGME, both Bittencourt et al.’s and Olsson et al.’s proposed techniques require a pre-mapped set.

Today’s commercial state-of-the-art SACC tools provide support for reflexion modelling using Naming Patterns in order to map source code entities to intended architecture. In Sonargraph-Architect source code entities are mapped to the architecture modules using class/package naming patterns defined by the user e.g. **/*gui/** to map source entities to the GUI architecture module. Structure101 Studio specifies a name pattern to map classes and packages to a given module. For example, packages that map to module GUI could be represented by the pattern net.java.gui.*. The drawback of using naming patterns is that it is not always the case that packages and their members directly map to modules in a 1-1 manner making defining of patterns a tedious exercise for software systems that have complex mapping configurations.

We hence believe that the effort in mapping has to be further reduced. From our experience in applying reflexion modelling in industrial contexts, we noticed that current techniques do not exploit one commonly provided and potentially helpful type of information: short descriptions of the purpose of the architectural modules articulated in natural language. In the following, we describe InMap, an interactive mapping algorithm that makes use of such descriptions to recommend mappings without the need for an initial set of pre-mapped entities.

3 INMAP

InMap is an interactive source code-to-architecture mapping recommendation algorithm for SACC methods. It proposes an improvement to the mapping process in reflexion modelling by eliminating the need for an initial set of pre-mapped source code in order to automate mapping. The technique takes as input descriptions of a system’s architectural modules in natural language which can be obtained from existing software system documentation. If documentation is unavailable, the descriptions can be obtained from the software architect/expert who can choose the level of detail to provide. The following is an example of a natural language module description for a “cli” module taken from JabRef, a system used in evaluation of the technique (see Section 4.1), “The cli module is very small and responsible for running JabRef and accessing its functionality without a UI. It parses command line arguments and maps them on certain functions from the logic module.”

Equipped with the module descriptions of a system, InMap works in 7 steps, shown in Fig. 1, as follows:

**STEP 1 – Filtering.** InMap begins by filtering out external libraries or source code entities that an architect does not wish to include in the mapping exercise. For example in the case of JabRef the exclusion list included the javafx.* package.

**STEP 2 – Pre-Processing.** InMap strips the source files of special characters as well as import and private statements, and programming language specific keywords e.g. int and switch in java.

**STEP 3 – Indexing.** In this step InMap indexes all the cleaned source code files. It uses Apache Lucene, an open source indexing and search library, to index the source code.

**STEP 4 – Query Formation.** In this step InMap automatically formulates the string that will be used to search the indexed source code files. The search query is a combination of four parts: (i) the name of the module the class could be mapped to; (ii) the module’s textual description; (iii) names of classes that are mapped to the module; and (iv) names of methods that belong to classes that are mapped to the module. The module name and description words are performed as wildcards searches e.g. “gui*” to search for “guiPane” and fuzzy searches e.g. “affect” to include “effect”. InMap also specifies the relevance of each term in a query using a boost factor. The higher the boost factor, the more relevant the term will be. InMap assigns boost factors to the module name, the Module Name Boost Factor (MNB) as well as to the module description words, the Module Description Boost Factor (MDBF). The module name and descriptions are assigned boost factors as they have the greatest probability of influencing the similarity score of an unmapped source code entity to an architectural module. The general structure of the query appears as follows,

\[
\text{(module name*~)} \text{MNB} + \text{ (module description*~)} \text{MDBF} + \text{ (mapped class names) + (mapped method names)}
\]

InMap retrieves the mapped method names from a call-dependency graph and strips them of common accessor method
prefixes such as get-, set-, is-, and on-. They are then ranked in order of the most frequently occurring within a module and then added to the remaining query space. The following is an example of what an actual query looks like.

```
("cli*~")^35.0 + ("cli*~" + "module*~" + "small *~" + "responsible-running*~" + "JabRef*~")^1.0 + ("Boot" + "PathHackerRunnable" + "AddJarsForPackageRunnable") ^ (+ "bestRepository" ...) 
```

By including mapped class and module names in the search query each time a new class is mapped the query changes allowing InMap’s to return more hits with each new mapping iteration. Each query is automatically created and submitted to Lucene.

**STEP 5 – Similarity Scoring.** InMap uses term frequency-inverse document frequency (tf-idf) as the similarity scoring function because it gives more importance to a term that frequently occurs in a few documents relative to the collection than one that occurs in many documents. Formally, \( \text{tf-idf} \) is defined as follows: given that term frequency \( (f_{t,d}) \) refers to the number of occurrences of term \( t \) in document \( d \); and document frequency \( (df_t) \) refers to the number of documents in the collection that contain a term \( t \); and inverse document frequency \( (idf_t) \) is defined as

\[
\text{idf}_t = \log \frac{N}{df_t}
\]

(1)

with \( N \) denoting the total number of documents in a collection by \( N \); then the \( tf-idf \) is defined as

\[
\text{tf-idf}_{d} = tf_{t,d} \times idf_t
\]

(2)

Lucene returns \( tf-idf \) weightings for each query term, and InMap uses them to produce a class-module similarity matrix.

**STEP 6 – Mapping Recommendations.** For each unmapped class InMap retrieves the highest class-module similarity score. However, InMap does some further filtering before presenting them to the architect. (1) If a class has two or more modules with the same highest score then none are put forth, to avoid arbitrary recommendations. (2) All recommendations below the arithmetic mean of the suggested mappings are discarded. (3) InMap uses a Page Size (PS) to determine how many recommendations at a time should be presented to the architect e.g. 20 per page. InMap ranks the recommendations according to their score and presents, to the architect the first set of recommended class-to-module mappings, \( R_1 \), that are no more than the \( x \) highest scoring pairs, where \( x \) is the page size.

**STEP 7 – Accept/Reject Mappings.** The architect reviews the recommendations agreeing and disagreeing based on his/her knowledge of the software system. Once this is completed, InMap returns to STEP 4 with new information about the module architectures. The names of the newly mapped classes along with names of their methods are used to update the query which is passed to Lucene to search the remaining set of unmapped classes and produce a set of new recommendations. The architect again accepts and rejects the recommendations in \( R_2 \). The algorithm returns to STEP 4 once more with new terms to add to the search query. The algorithm ends when the mapping is complete or when either the architect or

### Table 1: JabRef system properties

<table>
<thead>
<tr>
<th>Version #</th>
<th>Lines of code</th>
<th>Module architecture descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,7</td>
<td>88,603</td>
<td>3-5 sentences each</td>
</tr>
<tr>
<td></td>
<td>Line comments</td>
<td>Number of architectural modules:</td>
</tr>
<tr>
<td></td>
<td>17,170</td>
<td>6</td>
</tr>
<tr>
<td># of source files</td>
<td>843</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: JabRef results for PS=50, MNBF=33-35 & MDBF=1

<table>
<thead>
<tr>
<th>Classes</th>
<th>Recomm.</th>
<th>False Positives</th>
<th># of Iterations</th>
<th>Recall</th>
<th>Precision</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>842</td>
<td>901</td>
<td>61</td>
<td>38</td>
<td>1.00</td>
<td>0.93</td>
<td>0.96</td>
</tr>
</tbody>
</table>

InMap can make no more decisions. A prototypical tool we developed, along with the results can be found here [https://git.cs.kau.se/zipasink/InMap](https://git.cs.kau.se/zipasink/InMap).

## 4 EVALUATION

### 4.1 Data Collection & Analysis

We conducted a case study on JabRef an open source Java-based bibliographic reference manager. Table 1 shows the properties of the system. The architectural descriptions, codebase and oracle mappings used were obtained from the JabRef development team. To evaluate InMap’s recommendations, the oracle mappings provided were used to programmatically accept and reject mapping recommendations given in STEP 7. This allowed us to maintain accuracy and efficiency during our numerous experiments. We selected JabRef for the case study to allow us compare InMap to HuGME and NBC that were evaluated on the same system in a study by Olsson et al [4].

For differing values of PS, MNBF and MDBF, we collected the number of pages, the number of correct and incorrect recommendations per page, as well as the number of unmapped classes per page for each experiment. We computed precision, recall and f1-scores to observe the behaviour of the technique. To be able to compare InMap with HuGME & NBC we used a similar accuracy measure as reported in the study by Olsson et al [4], the f1-score defined as

\[
F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\]

(3)

### 4.3 Results

InMap was able to recommend the correct mapping for almost all 842 classes in JabRef (save for 2) using parameter settings of PS=50, MNBF=33.0-35.0 and MDBF=1.0. This corresponds to a recall value of 1.00. Additionally, InMap had 61 rejected recommendations (or false positives) out of 901 total recommendations given which corresponds to a precision of 0.93. This resulted in an f1-score of 0.96 as shown in Table 2.

## 5 DISCUSSION

From the results, we observe that InMap was able to map JabRef without requiring a pre-mapped set of entities. This was made possible using information retrieval techniques. InMap assumes any combination of the four parts of the query that make up the