Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

An incremental software architecture recovery technique driven by code changes*

Li WANG[1,2], Xianglong KONG[1], Jiahui WANG[3], Bixin LI‡[1]

[1]*School of Computer Science and Engineering, Southeast University, Nanjing 210096, China*

[2]*Jiangsu Automation Research Institute, Lianyungang 222061, China*

[3]*Huawei Digital Technology Lab, Suzhou 215125, China*

E-mail: wangli1218@seu.edu.cn; xlkong@seu.edu.cn; 18262609320@163.com; bx.li@seu.edu.cn

Received Sept. 29, 2021; Revision accepted Dec. 29, 2021; Crosschecked Mar. 1, 2022; Published online Apr. 6, 2022

Abstract: It is difficult to keep software architecture up to date with code changes during software evolution. Inconsistency is caused by the limitations of standard development specifications and human power resources, which may impact software maintenance. To solve this problem, we propose an incremental software architecture recovery (ISAR) technique. Our technique obtains dependency information from changed code blocks and identifies different strength-level dependencies. Then, we use double classifiers to recover the architecture based on the method of mapping code-level changes to architecture-level updates. ISAR is evaluated on 10 open-source projects, and the results show that it performs more effectively and efficiently than the compared techniques. We also find that the impact of low-quality architectural documentation on effectiveness remains stable during software evolution.

Key words: Architecture recovery; Software evolution; Code change
https://doi.org/10.1631/FITEE.2100461 **CLC number:** TP311

1 Introduction

Software architecture encompasses the principles and decisions of design and plays an important role in software lifecycle, especially in software evolution. It is difficult to maintain up-to-date architectural documentation because it should contain knowledge from all software stakeholders; therefore, significant literature exists concerning software architecture recovery (Kong et al., 2018; Cho et al., 2019; Schmitt Laser et al., 2020; Pourasghar et al., 2021). Software architecture recovery refers to the process of identifying and extracting architectural information from lower-level representations of a software system, such as source code (Mendonça and Kramer, 1998). Software recovery is a costly task in both academia and industry. For example, Garcia et al. (2013b) took two years of effort to recover the architecture of Google Chromium with the assistance of related developers.

Because of high cost of software architecture recovery, efforts have been dedicated to automated recovery, which applies automatic methods to extract architectural information, such as code dependency and module functionality (Lima et al., 2019; Link et al., 2019; Silva et al., 2019; Sözer, 2019; Lee and Lee, 2020). Most of the current techniques rely on the information extracted from source code (Andritsos and Tzerpos, 2005; Tamburri and Kazman, 2018). Code-based recovery techniques extract dataflow- or controlflow-based dependencies between code entities, and then identify components by applying clustering or prediction methods. The complexity and changeability of software code result in the hidden

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61872078)

ORCID: Li WANG, https://orcid.org/0000-0002-4093-7303; Bixin LI, https://orcid.org/0000-0001-9916-4790

drawback of architecture recovery techniques. On one hand, researchers usually extract architectural information by analyzing and clustering code entities based on the dependencies between them. Information loss is inevitable during aggregation and extraction, which may impact the effectiveness of architecture recovery. On the other hand, software evolution is constant, and it is costly to apply architecture recovery techniques to constantly update architecture. Software evolution refers to the dynamic behaviors of software maintenance and continuous updating throughout its lifecycle (Lehman, 1996; Ali and Maqbool, 2009). Code change is the most important form of software evolution, and it includes addition, deletion, and modification (Mens and Tourwe, 2004). Effectiveness and efficiency of automated architecture recovery are directly impacted by the efforts of code analysis and clustering methods.

There are also some empirical studies on current code-based recovery techniques (Anquetil and Lethbridge, 2003; Maqbool and Babri, 2004; Kobayashi et al., 2012; Garcia et al., 2013a); however, the conclusions from these studies are usually different from each other and there are no "silver bullets" for architecture recovery. There is no technique that always performs better than others in the recovery of massive projects. In practice, most large-scale projects have one or more high-quality architecture documents which are generated at the beginning of development or are revised through maintenance. Well-documented architecture clearly presents the structure of specific versions, and developers usually pay much attention to the design documents at the beginning of a project's lifecycle. Current software architecture recovery techniques still have a lot of room for improvement in effectiveness due to information loss during dependency processing (Garcia et al., 2013b). Therefore, our goal is to recover architecture based on some existing high-quality design documents. We can track code changes during software evolution, and we aim to build a mapping mechanism between code-level changes and architecture-level updates. In this way, we can recover architecture based on a partial dependency graph that is related to the changed code entities.

In this study, we propose an incremental software architecture recovery (ISAR) technique which consists of three steps. We first extract information from the changed code (Tufano et al., 2019). Then, we apply file-level code preprocessing on a dependency graph to map code-level changes and architecture-level updates. Finally, we recover the architecture including the update caused by code evolution. Our technique requires documentation of the previous version to update the architecture, and its quality directly impacts the performance of ISAR. The final step applies double classifiers to adjust the recovered architecture.

To evaluate the effectiveness and efficiency of ISAR, we conduct experiments on 10 open-source projects with two other architecture recovery techniques, i.e., Bunch (https://www.cs.drexel.edu/~spiros/bunch) (Mancoridis et al., 1999) and directory-based dependency processing (DBDP) (Kong et al., 2018). The results of our experiments show that ISAR performs the best in terms of effectiveness and efficiency. We also find that ISAR's effectiveness decreases obviously during evolution, but it stabilizes after several released versions. This means that ISAR can work well with low-quality architectural documentation. In summary, our paper makes the following contributions:

1. We propose an ISAR technique which recovers architecture based on existing architectures and related code changes.

2. We build a mapping between code-level changes and architecture-level updates, which can help researchers improve software recovery techniques.

3. We evaluate ISAR on 10 projects with two other recovery techniques, and find that our approach can generally improve the effectiveness and efficiency.

2 Approach

In this section, we present details of the ISAR technique. The technique comprises three main steps: information extraction from code, file-level code preprocessing, and incremental software architecture update.

The ISAR framework is presented in Fig. 1. First, the changed code files and file-level dependency graph are obtained by analyzing code before and after software evolution. Second, the changed code files and file-level dependency graph are preprocessed to determine the changed elements. These

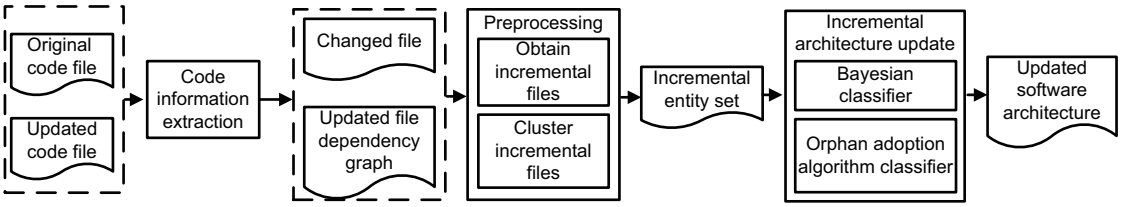

Fig. 1 Incremental software architecture recovery (ISAR) framework

changed elements are preliminarily clustered to form an incremental entity set. Finally, these incremental entities are processed using double classifiers to achieve the top-down incremental update of the software architecture.

2.1 Information extraction from code

The purpose of code information extraction is to obtain the changed code files and to build the file-level dependency graph after code changes. Code changes can be divided into five levels: directory level, file level, class level, method level, and statement level. The elements and types of changes are different in different levels. Table 1 describes the elements and operations that cause changes at different levels.

In this study, we focus on file-level code changes. The changed code elements need to be clustered from bottom to top into three kinds of change files, as shown in Table 2. We use the multi-level change detection tool (Wu et al., 2005) to obtain changed code files. The tool uses the abstract syntax tree (AST) built by Java development tools (JDT) to parse two different versions of code and to obtain multi-level changed files between target versions.

The file-level dependency graph is an abstraction of dependency information between files. As shown in Fig. 2, the node represents the code file and the directed edge represents the dependency between two files. For example, $A \rightarrow B$ means that file A depends on file B. In binary $<A, B>$ on the dependency edge, element A denotes the dependency type and element B denotes the number of dependencies. The numbers on the dependent edges indicate the file dependency strength. The dependency types and the number of dependencies between files are obtained by analyzing the AST, which can provide the basis for generating file-level dependency graphs.

In this study, we focus on 10 types of dependencies: generalization, implementation, combination,

Table 1 Elements and operations at different levels

Level of change	Element	Operation
Directory level	Directory	Addition Deletion Renaming
File level	File	Addition Deletion Modification
Class level	Common class Internal class Enum class Anonymous class	Addition Deletion Modification Movement
Method level	Method	Addition Deletion Modification Movement
Statement level	Statement	Addition Deletion Modification Movement

Table 2 Three types of file-level changes

Changed file	Types of change
Added file	Add statement, add method, add class, add file, add directory
Deleted file	Delete statement, delete method, delete class, delete file, delete directory
Modified file	Modify statement, move statement, modify method, move method, modify class, move class, move file, rename file, rename directory

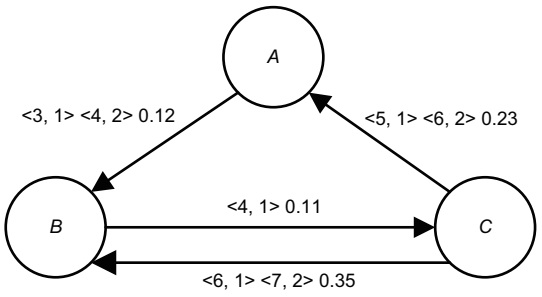

Fig. 2 An example of the file-level dependency graph

association, call, instantiation, parameter, return, declaration type, and import. Among them, generalization, implementation, and combination are the top three strongest dependencies (Glukhikh et al., 2012; Lutellier et al., 2018), and may have stronger relationships with architecture updates. The strength of code dependency can reflect the degree of dependence on the architecture level. The code entities that have stronger dependencies are more likely to be clustered into one module during architecture recovery. If A and B represent two files, the strength of dependency between them is defined as follows:

$$\mathrm{DependFile}_{AB} = \frac{1}{\ln(\mathrm{LOC}_A)} \sum_{i=1}^{i=n} \delta_i \mathrm{DependType}_i, \tag{1}$$

where LOC_A represents the number of valid lines in file A, $\mathrm{DependType}_i$ $(i = 1, 2, \cdots, n)$ represents the number of dependencies of type i, and δ_i represents the weight of dependency type i.

2.2 File-level code preprocessing

The purpose of file-level code preprocessing is to obtain the incremental entity set. First, when a file is added, deleted, or modified, the dependency between the file and other files may change. However, when a file does not change, its dependency may change due to the changes in other files. There are four types of dependency changes in code files: adding a new type of dependency, deleting a type of dependency, increasing the number of dependencies, and reducing the number of dependencies. In this study, the code file whose dependency changes is defined as an incremental file (Table 3).

According to Table 3, the incremental files can be divided into six sets: Set_Incre_Add, Set_Incre_Del, Set_Incre_TypeAdd, Set_Incre_TypeDel, Set_Incre_DepAdd, and Set_Incre_DepDel. There is also a set of non-incremental files, i.e., Set_NoIncreFile.

Second, we cluster incremental files with strong dependencies, namely, generalization, implementation, and combination. The incremental files with strong dependencies are clustered to form code modules. This approach can effectively reduce the number of objects to be updated and reduce the time needed for updating. Because the code module concept is added at this step, the concept of "entity" needs to be introduced. The incremental file set is upgraded to the incremental entity set. The elements in the incremental entity set include incremental files and incremental modules. The files in Set_Incre_Del are not to be clustered because they will be removed directly during the updating process. The clustering conditions are as follows:

Rule 1: When incremental file A is strongly dependent on non-incremental file B, A and B are clustered. A is classified into the module containing B.

Rule 2: When incremental file A is strongly dependent on incremental file B, A and B are merged into module C. The dependency between C and other files is the union set of the dependency of A and B. The dependency strength is calculated by summation.

Table 3 Information of incremental file types

File type	Type of dependency change	Type of incremental file
Added file	No change	Incre_Add
	Increasing the number of dependencies	Incre_DepAdd
Deleted file	No change	Incre_Del
	Reducing the number of dependencies	Incre_DepDel
Modified file	No change	NoIncreFile
	Increasing the number of dependencies	Incre_DepAdd
	Reducing the number of dependencies	Incre_DepDel
	Adding a new type of dependency	Incre_TypeAdd
	Deleting a type of dependency	Incre_TypeDel
Unchanged file	No change	NoIncreFile
	Increasing the number of dependencies	Incre_DepAdd
	Reducing the number of dependencies	Incre_DepDel
	Adding a new type of dependency	Incre_TypeAdd
	Deleting a type of dependency	Incre_TypeDel

Rule 3: When module C strongly depends on unchanged file D, we cluster all incremental files in C to D and delete C from the incremental entity set.

In Fig. 3, A, B, D, and F are code files and C is a code module. Suppose that A and B have strong dependency with type 1 and that the dependency strength is 0.3. According to the above rules, A and B are clustered into C. The type of dependency between C and D is type 2, the dependency strength is 0.1+0.2=0.3, the type of dependency between C and F is type 3, and the dependency strength is 0.2.

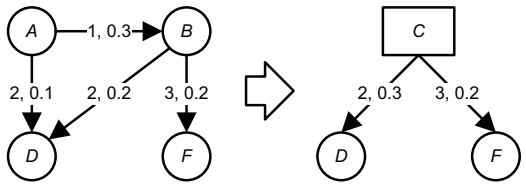

Fig. 3 An example of file clustering

2.3 Incremental software architecture update

Incremental software architecture update uses double classifiers to classify entities in the six kinds of incremental sets obtained by preprocessing to realize top-down incremental update. It can ensure the consistency of code and architecture. Incremental software architecture update includes the following two steps:

Step 1: We delete the incremental entities and their dependencies. The components of incremental entities may change after the classification by double classifiers. These incremental entities and their dependencies need to be removed from the file dependency graph. Algorithm 1 presents the details. First, we add the file dependency graph, module-file dependency graph, component-module hierarchy graph, and component dependency graph to the software architecture graph. Second, all the nodes and edges of entities in Set_Incre_Del, Set_Incre_TypeAdd, Set_Incre_TypeDel, Set_Incre_DepAdd, and Set_Incre_DepDel are removed from the module-file dependency graph. Then, we delete the modules and entities that do not exist in the new version of architecture graph. Finally, all elements in the six incremental entity sets except Set_NoIncreFile are merged and added into UPDATE for subsequent entity classification.

Step 2: We apply the entity classification using double classifiers, which can update software architecture incrementally. The two classifiers used in this study are a Bayesian classifier and an Orphan adoption algorithm classifier.

2.3.1 Bayesian classifier

The Bayesian classifier follows the Bayes theorem to check whether an incremental entity can be

Algorithm 1 Deletion of incremental entities and dependencies

Input: previous architecture graph preArcG and the sets of incremental entities
Output: architecture graph ArcG and the incremental entity set UPDATE

```
create Module-FileTable()
create Component-ModuleTable()
create Component-DependTable()
put preModule-FileGraph into Module-FileTable
put preComponent-ModuleGraph into Module-FileTable
put preComponent-DependGraph into Component-DependTable
for entity in sets of incremental entities do
    if entity.type == file then
        delete filenode in the module-file graph (MFG)
        delete edges in the MFG
    end if
    if entity.type == module then
        delete all filenode in the MFG
        delete all edges in the MFG
    end if
end for /* Sets of incremental entities consist of Set_Incre_Del, Set_Incre_TypeAdd, Set_Incre_TypeDel, Set_Incre_DepAdd, and Set_Incre_DepDel */
for module in the MFG do
    if module.filenum == 0 then
        delete module
    end if
end for
for component in MFG do
    if component.filenum == 0 then
        delete component
    end if
end for
ArcG←MFG
for entity in sets of incremental entities do
    if entity.type == file then
        add entity into UPDATE
    end if
    if entity.type == module then
        create _entity
        _entity.type←module
        _entity.fileSet←entity.fileSet
        _entity.edges←entity.edges
        add _entity to UPDATE
    end if
end for
return ArcG and UPDATE
```

classified into a component according to the clustering probability between them. The basic formula is

$$P(C_j|F_i) = \frac{P(F_i|C_j)P(C_j)}{P(F_i)}, \tag{2}$$

where j ranges from 1 to n (n represents the number of components) and i ranges from 1 to m (m represents the number of entities to be classified). $P(C_j)$ and $P(F_i)$ are the prior probabilities of components C_j and F_i, respectively. $P(F_i|C_j)$ is the posterior probability of F_i given C_j. $P(C_j|F_i)$ is the posterior probability of C_j given F_i.

In Fig. 4, nodes represent files, rectangles represent components, arrows represent dependencies, and the numbers of the dependence instances are presented in the arrows. According to the Bayes formula, $P(C_1) = \frac{4}{9}$, $P(C_2) = \frac{2}{9}$, $P(C_3) = \frac{3}{9}$, $P(F_\mathrm{a}|C_1) = \frac{3}{4}$, $P(F_\mathrm{a}|C_2) = \frac{1}{2}$, $P(F_\mathrm{a}|C_3) = \frac{1}{3}$, $P(C_1|F_\mathrm{a}) = \frac{1}{3}$, $P(C_2|F_\mathrm{a}) = \frac{1}{9}$, and $P(C_3|F_\mathrm{a}) = \frac{1}{9}$. In this case, F_a has the highest probability of being divided into component C_1.

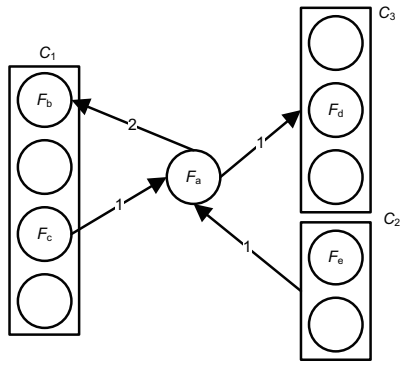

Fig. 4 Component-file dependency graph

2.3.2 Orphan adoption algorithm classifier

The Orphan adoption algorithm classifier (Tzerpos and Holt, 2000) uses naming rules and structural rules to find components with the highest strength of dependence for Orphan resources (i.e., incremental entities). When the dependence strength is the same, we can use tie-breakers to carry out the final classification. The rules are as follows:

Rule 1: The incremental entity is included in the candidate component, which reduces the penetration.

Rule 2: The incremental entity is included in the candidate components, and the output of the candidate components increases the least.

2.3.3 Combined use of these two classifiers

First, the Bayesian classifier is used to calculate the probability of the incremental entity F being allocated to each component, and the probability is sorted non-incrementally according to the value. The probabilities that incremental entity F is assigned to components C_i and C_j are $P(C_i|F)$ and $P(C_j|F)$, respectively. $P(C_i|F)$ and $P(C_j|F)$ are the maximum probability and the secondary maximum probability, respectively. The usage rules of these two combined classifiers are as follows:

Rule 1: If $P(C_i|F) \geq 50$ and $P(C_i|F) \pm P(C_j|F) \geq \alpha$, then incremental entity F is allocated to component C_i, where $0 \leq \alpha \leq 1$. The larger the value of α, the greater the probability that incremental entity F is allocated to component C_i. Through our experiments in Section 3, α is taken as 0.3.

Rule 2: If $P(C_i|F) \pm P(C_j|F) < \alpha$ and $P(C_i|F)$ is not 0, we select the top-k components that have the highest probabilities of incremental entity F being allocated to them. The Orphan adoption algorithm classifier is used to select the components to which incremental entity F should finally be allocated. The larger the value of k, the higher the classification accuracy, and the higher the time cost. k is taken as 10 in this work.

In rule 1, only the Bayesian classifier is used to classify incremental entity F. This is because the probability that file F is allocated to component A is far greater than those of other components. If the Orphan adoption algorithm classifier is used for secondary classification, the results could be the same with a significant probability, which will increase the time cost. In rule 2, the Bayesian classifier and the Orphan adoption algorithm classifier are used to classify the incremental entities. This is because only the Bayesian classifier cannot complete the classification accurately. The Orphan adoption algorithm classifier is used to analyze the dependency between the incremental entities.

According to the double-classifier usage rules, we classify the files and modules in the incremental entity set UPDATE from the preprocessing step. First, entities in UPDATE are classified using these double classifiers. When the incremental entities are classified to a new component, all the non-incremental files associated with the incremental entities are put into UPDATE for re-classification.

Second, we classify the entities that have been classified at step 1 at the module level. Finally, we adjust the component clustering, including adding, deleting, and splitting components. Details of the incremental software architecture update algorithm are shown in Algorithm 2. The rules used in Algorithm 2 are as follows:

Algorithm 2 Incremental software architecture update

Input: architecture graph ArcG and UPDATE
Output: newArcG /* All entities in UPDATE are classified by the Bayesian classifier */

```
for entity in UPDATE do
    if Bayesian_Classifier(entity) == true then
        update newCFG() /* When an incremental entity is classified from the original component to another component, all files associated with the incremental entity are re-classified in UPDATE */
    end if
    if entity.OwningComponents != entity.PreComponent then
        put entity.fileSet into UPDATE
    else if OA(entity) == true then
        update newCFG()
    else
        put entity into Failset
    end if
end for
for entity in UPDATE do
    classify the entity to various modules
end for /* Iterative module classification */
AddComp(Failset) /* Adjusting components */
DeleteComp(ArcG) /* When no entity is detected in a component, delete the component */
newArcG=SpiltComp(ArcG)
return newArcG
```

Rule 1: To add components, there are two situations for entities in Failset that cannot be classified. One is that some entities (including files and modules) do not have any dependency relationship with other entities and they belong to isolated entities. We will add components to which these isolated entities can be classified. The other one is that there are dependency relationships among these entities. We will cluster these entities from bottom to top to form new components.

Rule 2: When no entity is detected in a component, we delete the component.

Rule 3: When the component scale is larger than 30% of the system scale, the component is considered to be too large to split.

3 Experiments and results

The incremental software architecture update technique can generate accurate relationships between code elements based on historical versions of architecture, which can help developers understand and maintain software projects. In this section, we aim to answer the following research questions:

RQ1: How effective is the software architecture update technique?

RQ2: How does the software architecture update technique perform with the evolution of projects?

RQ3: How efficient is the software architecture update technique?

3.1 Experimental setup

3.1.1 Subject projects

To answer the above research questions, we selected 10 Java projects from GitHub according to their popularity, i.e., Okhttp, Mabatis, Mockito, Junit, Retrofit, Jadx, Terrier, Clone, Freecol, and Fastjson, which have been widely used in existing studies (Sievi-Korte et al., 2019; Kong et al., 2020; Jia et al., 2021; Liu et al., 2021; Pourasghar et al., 2021; Zhang et al., 2021). For each project, we selected the first five released versions on GitHub. All of these projects have high-quality documentation, which makes the manual recovery of software architecture possible for our team.

The initial architectures of the studied projects were recovered manually according to the existing works (Garcia et al., 2013b; Kong et al., 2018). The manual recovery included four steps. First, we extracted the file-level dependency graph based on Eclipse JDT and listed the detailed relationships between files. Second, we checked the partitioning of file directories and clustered the coupled files within the same directory into a module. Third, we iteratively grouped the modules according to their dependency relationships and functionalities, which were partially obtained through the online documentation. Finally, an architect from Huawei Digital Technology Lab (i.e., the third author of this paper) revised the architecture based on the publicly available information and her experience.

Table 4 presents the details of these selected projects. LOC presents the number of valid lines in subject projects, which is calculated using CLOC

(https://github.com/AlDanial/cloc) in the experiments. The description presents the basic functionality of each project.

3.1.2 Selected recovery techniques

To evaluate the effectiveness of our incremental software architecture recovery technique, we selected Bunch (Mancoridis et al., 1998) and DBDP (Kong et al., 2018) to compare the recovered architecture. Bunch is a classical architecture recovery technique, and has been widely used in existing studies (Schmitt Laser et al., 2020; Campo et al., 2021). DBDP is another effective architecture recovery technique which extracts information from code and the structure of directories. DBDP is viewed as a promising method for extracting architectural information from code and directories (Pourasghar et al., 2021). DBDP is compatible with the two kinds of hill-climbing algorithms to resolve the optimization problem, i.e., the nearest and steepest ascent hill climbing (NAHC and SAHC) used by Bunch. There are two reasons for the selection of these techniques. First, these two techniques have generated promising results in existing studies (Schmitt Laser et al., 2020; Pourasghar et al., 2021). Second, it is possible for us to obtain source code for these two techniques, which helps us easily apply automation in our experiments. In our experiments, we collected the results generated by NAHC and SAHC and used the more accurate one in the comparison. The clustering algorithm is Bunch.TurboMQ and the results are presented as the median level of graph.

3.1.3 Measurements

MojoSim is widely used to measure the effectiveness of software architecture recovery techniques (Wu et al., 2005; Bittencourt and Guerrero, 2009; Bazylevych and Burtnyk, 2015; Lutellier et al., 2015, 2018). It can calculate the similarity between the recovered architecture and ground-truth architecture. A high similarity value means that the recovered architecture is accurate. It is defined by the following formula:

$$\mathrm{MoJoSim}(A,B)=\left(1-\frac{\mathrm{mno}(A,B)}{n}\right)\times 100\%, \quad (3)$$

where A indicates the architecture that is updated by our technique, B indicates the ground-truth architecture, $\mathrm{mno}(A,B)$ is the minimum number of Move and Join operations needed to convert A to B, and n is the number of architecture entities. The 100% value of $\mathrm{MoJoSim}(A,B)$ means that the updated architecture A is the same as the ground-truth architecture B.

Turbo MQ extends the basic modularization quality (Basic MQ) to support a dependency graph with edge weights. Turbo MQ can measure the quality of the recovered architecture without the ground-truth architecture. Turbo MQ evaluates the architecture based on the degree of high cohesion and low coupling, and is calculated by the following formula:

$$\text{Turbo MQ}=\sum_{i=1}^{N}\mathrm{CF}_i, \quad (4)$$

where CF_i (i.e., cluster factor of module i) is defined as

$$\mathrm{CF}_i=\frac{\mu_i}{\mu_i+0.5\sum_j(\epsilon_{ij}+\epsilon_{ji})}. \quad (5)$$

To calculate the cluster factor of modules i and j, we collected μ_i as the number of intra-relationships of module i and $\epsilon_{ij}+\epsilon_{ji}$ as the number of inter-relationships between cluster i and cluster j. A

Table 4 Subject system statistics

Project	LOC	Number of files	Description
Okhttp	53 114	325	An Android lightweight framework for network requests
Mabatis	51 044	918	A persistence layer framework to support customized SQL
Mockito	40 411	863	A simulation test framework for simple verification error production
Junit	3512	47	A regression testing framework for Java
Retrofit	19 193	235	A restful HTTP network request framework
Jadx	45 619	574	An open source tool to decompile APK files
Terrier	55 485	1122	A program for rapid development of web and desktop search engines
Clone	10 198	91	Game written in Java
Freecol	118 428	773	A turn-based strategy game; the open source version of colonization
Fastjson	117 300	1972	An open source tool for parsing and packaging JSON formatted data

higher value of Turbo MQ means that the organization of the architecture is better, i.e., more satisfying the principle of design.

3.1.4 Experimental steps

For each studied subject, we performed the following steps:

Step 1: We collected the first five released versions of the 10 projects in Table 4 from GitHub, which gave us 50 different programs in total.

Step 2: To obtain the ground-truth architecture, we implemented architecture recovery manually on the first version for each project. The manual recovery was designed according to the exist works (Garcia et al., 2013b; Kong et al., 2018). The architecture was generated based on the file-level dependency graph and the publicly available documents.

Step 3: For each studied project, we recovered the architecture with Bunch and DBDP to build the comparative experiments.

Step 4: For each selected project, we collected the code changes from its previous version. The first version of a project was marked as the initial version, so it had no changes. There were a total of 166 code blocks which were changed during the project evolution. These changes helped us recover architecture based on the previous architecture version.

Step 5: For the last four versions of the projects, we recovered the architecture with our incremental software architecture recovery technique based on the previous version, and collected all the results to analyze the effectiveness and efficiency.

In the experiments, the file-level dependency graph was obtained through Eclipse JDT. We ran the three architecture recovery techniques, i.e., ISAR, Bunch, and DBDP, on a Ryzen 3950X server with 128 GB of memory.

3.2 Result analysis

3.2.1 RQ1: improvements in effectiveness

To evaluate the effectiveness of our technique, we applied ISAR, Bunch, and DBDP on the second version of the studied projects. There are two reasons to choose the second version: the ground-truth architecture of the second version is available according to step 2 and its previous version of ground-truth architecture, i.e., the first version, is also available. Our technique, i.e., ISAR, needs the architecture of the previous version of the target project because it can establish a mapping between code-level changes and architecture-level updates. The ground-truth architecture of the second version is obtained manually. We collected the results and calculated the MojoSim scores on the basis of the related ground-truth architecture. Table 5 presents the results.

Table 5 MoJoSim scores of the studied techniques

Project	MoJoSim score		
	ISAR	Bunch	DBDP
Okhttp	0.92	0.71	0.88
Mabatis	0.95	0.73	0.89
Mockito	0.92	0.72	0.88
Junit	0.95	0.65	0.75
Retrofit	0.85	0.57	0.85
Jadx	0.76	0.55	0.64
Terrier	0.90	0.68	0.88
Clone	0.91	0.77	0.90
Freecol	0.91	0.61	0.75
Fastjson	0.92	0.62	0.82
Average	0.90	0.66	0.82

From the results in Table 5, we have the following observations:

First, ISAR obtains the highest MojoSim scores on the second version of the studied projects. This means that the architecture generated by ISAR obtains the highest value of similarity with the ground-truth architecture compared with the architectures produced by Bunch and DBDP. The overall 0.90 ISAR MojoSim score is a very promising result. We recovered the architecture based on the previous architecture and code changes, and the results take the advantage of the manual recovery. Because DBDP is implemented based on Bunch, it usually obtains a higher score than Bunch.

Second, the MojoSim scores of the Retrofit and Jadx projects are much lower than those of the other studied projects. We looked into the source code of these two projects and found that their dependencies are much more centralized than other results. These two projects have a high-weighted module in their architecture, which seriously affects other small module scores. For these two projects, all the studied techniques generated several different small modules, which makes the MojoSim scores lower than those of the other projects.

Finding 1 For the studied projects, ISAR performs the best in terms of effectiveness.

3.2.2 RQ2: effectiveness during evolution

All the studied projects have high-quality design documents, which help us obtain the ground-truth architecture of the first version. When the projects change with various requirements, the architecture may not be updated in a timely manner. Therefore, we evaluated ISAR on the last four versions of the studied projects to investigate how the effectiveness changes during evolution. Table 6 presents the results of Turbo MQ for ISAR on the projects. Because ISAR needs the previous version to generate architecture, we could not collect the results of the first version. A high Turbo MQ score means that the structure of the architecture is good; i.e., it satisfies the "high cohesion and low coupling" design principle.

From Table 6, we can see that the effectiveness of ISAR decreases obviously during evolution. The degree of the decline is acute at first and becomes stable between the last two versions. The reason for the downtrend is that our technique recovers architecture on the basis of the previous version, which results in a loss in the stock of information during evolution. However, we still have a good result because the decline can be stabilized after several released versions, so our technique can play proper roles for the projects that do not have high-quality previous design documents.

Finding 2 Although low-quality architectural documentation can obviously impact the effectiveness of ISAR, it can still obtain promising results, i.e., an average 8.21 Turbo MQ score on the last version.

Table 6 Turbo MQ scores on the projects of ISAR

Project	Turbo MQ score			
	Version 2	Version 3	Version 4	Version 5
Okhttp	12.78	10.89	8.47	8.32
Mabatis	13.01	11.66	9.65	8.88
Mockito	11.89	10.08	8.23	7.95
Junit	2.55	2.51	2.41	2.12
Retrofit	6.89	6.88	5.64	5.44
Jadx	7.11	6.84	5.78	4.78
Terrier	14.55	12.66	10.44	10.32
Clone	13.24	13.04	10.47	9.19
Freecol	20.64	16.55	14.34	12.08
Fastjson	19.87	14.52	13.94	12.99
Average	12.25	10.56	8.94	8.21

3.2.3 RQ3: efficiency improvements

We collected time consumption data of ISAR, Bunch, and DBDP on the last four versions of the studied projects. Table 7 presents the execution time of the techniques; the execution time of the three studied techniques for the four versions of selected projects are shown. From the table, we can see that the execution time of ISAR is much shorter than those of the two other techniques. This is because our technique does not start with the whole dependency graph of the target project. ISAR needs to translate only the architecture-level changes to the previous version, whereas the two other techniques need to apply the clustering algorithm several times during recovery. The complex structure of the dependency graph makes the architecture recovery time-consuming. For the small-sized projects, ISAR efficiency improvements are not obvious. In cases of recovery on large-sized projects, i.e., Freecol and Fastjson, ISAR can significantly improve the efficiency. Consequently, we claim that ISAR

Table 7 Time consumption of studied techniques

Project	Time (s)											
	Version 2			Version 3			Version 4			Version 5		
	ISAR	DBDP	Bunch	ISAR	DBDP	Bunch	ISAR	DBDP	Bunch	ISAR	DBDP	Bunch
Okhttp	8378	32 175	22 547	8641	30 445	25 963	9655	30 927	21 874	7931	29 488	22 554
Mabatis	12 456	72 866	57 412	11 456	68 543	55 478	7645	39 012	36 885	6468	58 238	54 365
Mackito	14 748	57 915	38 452	12 658	59 611	36 462	18 984	65 025	60 742	12 750	60 924	58 168
Junit	1988	3139	2151	1964	3830	2485	1648	3295	2185	2277	3934	2987
Retrofit	2834	7032	5647	2904	7397	5784	3545	8290	5620	2733	7128	5583
Jadx	12 445	51 904	41 875	14 489	55 391	39 845	15 794	55 031	41 321	10 494	50 415	40 871
Terrier	9897	63 008	52 965	8478	57 687	51 492	10 021	59 422	50 774	10 487	61 140	49 863
Clone	6291	11 008	8456	4445	9313	7458	5156	10 420	6985	5265	9772	7059
Freecol	16 854	121 856	98 635	15 687	119 743	88 919	18 900	131 695	108 990	15 805	118 097	106 983
Fastjson	19 124	149 658	99 873	18 871	139 198	97 668	18 844	133 109	90 869	19 657	121 886	110 493

can obtain the most accurate architecture in the shortest time.

Finding 3 ISAR performs the best in terms of efficiency due to the reduction of target dependencies.

In summary, ISAR can obviously improve the effectiveness of architecture recovery with the help of existing high-quality architectural documentation. If there is no available previous version architecture, architects may obtain the design through some existing recovery techniques or manual work, and then ISAR can significantly improve the efficiency of recovery for the following versions.

4 Threats to validity

1. Threats to construct validity

The metrics used in these experiments threaten the construct validity. To reduce this threat, we selected widely used measurements, i.e., MojoSim and Turbo MQ. We used Turbo MQ scores to evaluate the effectiveness of ISAR during evolution without the ground-truth architecture. In future work, we will evaluate the techniques in terms of more measurement types.

2. Threats to internal validity

The main threat to internal validity is the potential negligence in our manual recovery. The projects in our experiments have been widely used in existing works, and we obtained some detailed design documents from online communities which help us a lot during recovery. To examine the accuracy of architectures we recovered manually, we studied as many related documents as possible, and discussed the conformation of the recovered architecture with some professional architects from the industry. To reduce this threat, we will try to communicate with the related developers to obtain the detailed design document and send them the results we recovered manually.

The other threat to internal validity is the potential faults in existing tool configurations, or in our data analysis. To reduce this threat, we carefully reviewed all the configurations, code, and data analysis scripts used in the study.

3. Threats to external validity

The selected versions of projects, dependency extracting tools, and automatic architecture recovery tools used in our experiments may pose threats to external validity. The projects used in our experiments are all popular Java programs on GitHub. We obtained the source code of the first five released versions from their online repositories. We used Eclipse JDT to extract code dependencies and built a file-level dependency graph.

We selected two existing techniques in the experiments, i.e., Bunch and DBDP. Bunch is implemented based on a genetic algorithm and a hill-climbing algorithm. Because of the iterative random process, the recovered architecture is not unique for a specific project. DBDP is based on Bunch, but it applies a new dependency preprocessing approach to improve the effectiveness. To reduce these threats, we will conduct the study with more dependency extracting tools, more architecture recovery tools, and more projects.

5 Related works

In the literature, there are many software architecture recovery techniques. According to the module extraction method, the current techniques can be divided into four categories: cluster-, pattern-, graph-, and classification-based architecture recovery techniques.

Cluster-based architecture recovery techniques commonly extract the structure of modules based on some clustering algorithms, i.e., mountain climbing and genetic algorithm (Mitchell and Mancoridis, 2006; Sözer, 2019). The techniques may improve the modular clustering technology for object-oriented systems (Zhao et al., 2015), take advantage of cluster ensembles to obtain accurate architecture (Cho et al., 2019), apply collaborative clustering technology in the field of software modularization (Naseem et al., 2013), or apply meta-heuristic search techniques (Mitchell, 2003; Yang et al., 2021) during clustering. Teymourian et al. (2020) proposed a fast clustering algorithm for large-sized projects, which performs operations on the dependency matrices. Most clustering algorithms are hierarchical and search-based, and cluster-based architecture recovery techniques can present an explicit structural design, but they are usually time-consuming.

Pattern-based architecture recovery techniques commonly follow a pattern-driven approach to help understand software (Tzerpos and Holt, 2000). This kind of technique is based on some common patterns, such as the directory structure pattern, source

file pattern, leaf set pattern, and body-head pattern. Tamburri and Kazman (2018) investigated a general usage pattern to match code entities with architecture information. Monroy and Pinzger (2021) mined the pattern through interaction with stakeholders and obtained the architecture based on a conceptual model. Teymourian et al. (2020) recovered architecture descriptions based on centrality measures, which can guide the architect to distribute the entities to different architectural layers. The condition of the architecture pattern limits the usage range of this kind of technique because most of them are domain-specific.

Graph-based architecture recovery techniques usually have two ways of generating architecture. One is to construct a module dependency graph and use vectors to divide the graph into subgraphs to increase cohesion and to reduce coupling between nodes (Akthar and Rafi, 2010). The other method converts the recovery process to a graphical pattern matching process by constructing a software entity relationship graph (Sartipi, 2003). Pourasghar et al. (2021) combined a pattern-based method and a cluster-based method to build a graph-based clustering algorithm that can use the depth of relationships to calculate similarity between artifacts. The main drawbacks of graph-based techniques are that the results are usually a locally optimal solution and that the techniques work very slowly when faced with large graphs.

Classification-based architecture recovery techniques usually use Bayesian classification to divide updated software modules into subsystems to update incomplete or outdated documents in the software (Maqbool and Babri, 2007). The use of Bayesian classification focuses only on code addition and ignores deletion and modification. Link et al. (2021) applied text classification on the identification and extraction of architectural information. The effectiveness of these techniques is limited by the classifier, which is generated based on rich knowledge of architecture.

We present an incremental software architecture recovery technique driven by code changes, which uses a Bayesian classifier and an Orphan adoption classifier to make incremental software architecture updates based on original architectural documentation. Different from traditional methods, our technique takes into account the similarity of code during evolution, and we update the previous architecture from the changed spots instead of comprehensive clustering or classification.

6 Conclusions

Traditional software architecture recovery techniques do not take into account the similarity between two versions of software architecture during evolution. This kind of information loss results in excessive time consumption during architecture recovery. Therefore, in this paper, we presented an incremental software architecture recovery technique driven by code changes, i.e., ISAR. We used a Bayesian classifier and an Orphan adoption classifier to update software architecture based on features of the previous version and changed entities. We evaluated ISAR by conducting experiments on 10 projects and comparing it with two other recovery techniques. The results showed that ISAR performs the best in terms of effectiveness and efficiency. There is still room for improvement of architecture recovery effectiveness. Follow-up work can start with the method of combining multiple classifiers to further improve the effectiveness of software architecture recovery.

Contributors

Bixin LI designed the technical framework. Li WANG and Xianglong KONG implemented the approach and drafted the paper. Jiahui WANG proposed the initial idea and confirmed the correctness of the recovered architecture. Bixin LI revised and finalized the paper.

Compliance with ethics guidelines

Li WANG, Xianglong KONG, Jiahui WANG, and Bixin LI declare that they have no conflict of interest.

References

Akthar S, Rafi S, 2010. Recovery of software architecture using partitioning approach by Fiedler vector and clustering. *Comput Inform Sci*, 3(1):72-75. https://doi.org/10.5539/cis.v3n1p72

Ali S, Maqbool O, 2009. Monitoring software evolution using multiple types of changes. Int Conf on Emerging Technologies, p.410-415. https://doi.org/10.1109/ICET.2009.5353135

Andritsos P, Tzerpos V, 2005. Information-theoretic software clustering. *IEEE Trans Softw Eng*, 31(2):150-165. https://doi.org/10.1109/TSE.2005.25

Anquetil N, Lethbridge TC, 2003. Comparative study of clustering algorithms and abstract representations for software remodularisation. *IEE Proc Softw*, 150(3):185-201. https://doi.org/10.1049/ip-sen:20030581

Bazylevych R, Burtnyk R, 2015. Algorithms for software clustering and modularization. X^{th} Int Scientific and Technical Conf "Computer Sciences and Information Technologies", p.30-33. https://doi.org/10.1109/STC-CSIT.2015.7325424

Bittencourt RA, Guerrero DDS, 2009. Comparison of graph clustering algorithms for recovering software architecture module views. 13^{th} European Conf on Software Maintenance and Reengineering, p.251-254. https://doi.org/10.1109/CSMR.2009.28

Campo M, Amandi A, Biset JC, 2021. A software architecture perspective about Moodle flexibility for supporting empirical research of teaching theories. *Educ Inform Technol*, 26(1):817-842. https://doi.org/10.1007/s10639-020-10291-4

Cho C, Lee KS, Lee M, et al., 2019. Software architecture module-view recovery using cluster ensembles. *IEEE Access*, 7:72872-72884. https://doi.org/10.1109/ACCESS.2019.2920427

Garcia J, Ivkovic I, Medvidovic N, 2013a. A comparative analysis of software architecture recovery techniques. 28^{th} IEEE/ACM Int Conf on Automated Software Engineering, p.486-496. https://doi.org/10.1109/ASE.2013.6693106

Garcia J, Krka I, Mattmann C, et al., 2013b. Obtaining ground-truth software architectures. 35^{th} Int Conf on Software Engineering, p.901-910. https://doi.org/10.1109/ICSE.2013.6606639

Glukhikh MI, Itsykson VM, Tsesko VA, 2012. Using dependencies to improve precision of code analysis. *Autom Contr Comput Sci*, 46(7):338-344. https://doi.org/10.3103/S0146411612070097

Jia XY, Chen SQ, Zhou XQ, et al., 2021. Where to handle an exception? Recommending exception handling locations from a global perspective. IEEE/ACM 29^{th} Int Conf on Program Comprehension, p.369-380. https://doi.org/10.1109/ICPC52881.2021.00042

Kobayashi K, Kamimura M, Kato K, et al., 2012. Feature-gathering dependency-based software clustering using Dedication and Modularity. 28^{th} IEEE Int Conf on Software Maintenance, p.462-471. https://doi.org/10.1109/ICSM.2012.6405308

Kong XL, Li BX, Wang LL, et al., 2018. Directory-based dependency processing for software architecture recovery. *IEEE Access*, 6:52321-52335. https://doi.org/10.1109/ACCESS.2018.2870118

Kong XL, Han WN, Liao L, et al., 2020. An analysis of correctness for API recommendation: are the unmatched results useless? *Sci China Inform Sci*, 63(9):190103. https://doi.org/10.1007/s11432-019-2929-9

Lee KS, Lee CG, 2020. Identifying semantic outliers of source code artifacts and their application to software architecture recovery. *IEEE Access*, 8:212467-212477. https://doi.org/10.1109/ACCESS.2020.3040024

Lehman MM, 1996. Laws of software evolution revisited. 5^{th} European Workshop Software Process Technology, p.108-124. https://doi.org/10.1007/BFb0017737

Lima C, Assunção WK, Martinez J, et al., 2019. Product line architecture recovery with outlier filtering in software families: the Apo-Games case study. *J Braz Comput Soc*, 25(1):7. https://doi.org/10.1186/s13173-019-0088-4

Link D, Behnamghader P, Moazeni R, et al., 2019. The value of software architecture recovery for maintenance. Proc 12^{th} Innovations on Software Engineering Conf (formerly known as India Software Engineering Conf), Article 17. https://doi.org/10.1145/3299771.3299787

Link D, Srisopha K, Boehm B, 2021. Study of the utility of text classification based software architecture recovery method RELAX for maintenance. Proc 15^{th} ACM/IEEE Int Symp on Empirical Software Engineering and Measurement, Article 33. https://doi.org/10.1145/3475716.3484194

Liu X, Wang HD, Ma HY, et al., 2021. The architecture design and implementation of aircraft structural fault assistant decision system based on data analysis. *J Phys Conf Ser*, 1813:012032. https://doi.org/10.1088/1742-6596/1813/1/012032

Lutellier T, Chollak D, Garcia J, et al., 2015. Comparing software architecture recovery techniques using accurate dependencies. IEEE/ACM 37^{th} IEEE Int Conf on Software Engineering, p.69-78. https://doi.org/10.1109/ICSE.2015.136

Lutellier T, Chollak D, Garcia J, et al., 2018. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Trans Softw Eng*, 44(2):159-181. https://doi.org/10.1109/TSE.2017.2671865

Mancoridis S, Mitchell BS, Rorres C, et al., 1998. Using automatic clustering to produce high-level system organizations of source code. Proc 6^{th} Int Workshop on Program Comprehension, p.45-52. https://doi.org/10.1109/WPC.1998.693283

Mancoridis S, Mitchell BS, Chen Y, et al., 1999. Bunch: a clustering tool for the recovery and maintenance of software system structures. Proc IEEE Int Conf on Software Maintenance, p.50-59. https://doi.org/10.1109/ICSM.1999.792498

Maqbool O, Babri HA, 2004. The weighted combined algorithm: a linkage algorithm for software clustering. 8^{th} European Conf on Software Maintenance and Reengineering, p.15-24. https://doi.org/10.1109/CSMR.2004.1281402

Maqbool O, Babri HA, 2007. Bayesian learning for software architecture recovery. Int Conf on Electrical Engineering, p.1-6. https://doi.org/10.1109/ICEE.2007.4287309

Mendonça NC, Kramer J, 1998. An experiment in distributed software architecture recovery. 2^{nd} Int ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, p.106-114. https://doi.org/10.1007/3-540-68383-6_16

Mens T, Tourwe T, 2004. A survey of software refactoring. *IEEE Trans Softw Eng*, 30(2):126-139. https://doi.org/10.1109/TSE.2004.1265817

Mitchell BS, 2003. A heuristic approach to solving the software clustering problem. Int Conf on Software Maintenance, p.285-288. https://doi.org/10.1109/ICSM.2003.1235432

Mitchell BS, Mancoridis S, 2006. On the automatic modularization of software systems using the Bunch tool. *IEEE Trans Softw Eng*, 32(3):193-208. https://doi.org/10.1109/TSE.2006.31

Monroy M, Pinzger M, 2021. ARCo: architecture recovery in context. *J Xi'an Univ Arch Technol*, XIII(2):128.

Naseem R, Maqbool O, Muhammad S, 2013. Cooperative clustering for software modularization. *J Syst Softw*, 86(8):2045-2062. https://doi.org/10.1016/j.jss.2013.03.080

Pourasghar B, Izadkhah H, Isazadeh A, et al., 2021. A graph-based clustering algorithm for software systems modularization. *Inform Softw Technol*, 133:106469. https://doi.org/10.1016/j.infsof.2020.106469

Sartipi K, 2003. Software architecture recovery based on pattern matching. Int Conf on Software Maintenance, p.293-296. https://doi.org/10.1109/ICSM.2003.1235434

Schmitt Laser M, Medvidovic N, Le DM, et al., 2020. ARCADE: an extensible workbench for architecture recovery, change, and decay evaluation. Proc 28th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering, p.1546-1550. https://doi.org/10.1145/3368089.3417941

Sievi-Korte O, Richardson I, Beecham S, 2019. Software architecture design in global software development: an empirical study. *J Syst Softw*, 158:110400. https://doi.org/10.1016/j.jss.2019.110400

Silva DEU, Bittencourt RA, Calumby RT, 2019. Clustering similarity measures for architecture recovery of evolving software. Anais do VII Workshop de Visualização, Evolução E Manutenção de Software, p.45-52. https://doi.org/10.5753/vem.2019.7583

Sözer H, 2019. Evaluating the effectiveness of multi-level greedy modularity clustering for software architecture recovery. 13th European Conf on Software Architecture, p.71-87. https://doi.org/10.1007/978-3-030-29983-5_5

Tamburri DA, Kazman R, 2018. General methods for software architecture recovery: a potential approach and its evaluation. *Empir Softw Eng*, 23(3):1457-1489. https://doi.org/10.1007/s10664-017-9543-z

Teymourian N, Izadkhah H, Isazadeh A, 2020. A fast clustering algorithm for modularization of large-scale software systems. *IEEE Trans Softw Eng*, early access. https://doi.org/10.1109/TSE.2020.3022212

Tufano M, Sajnani H, Herzig K, 2019. Towards predicting the impact of software changes on building activities. IEEE/ACM 41st Int Conf on Software Engineering, p.49-52. https://doi.org/10.1109/ICSE-NIER.2019.00021

Tzerpos V, Holt RC, 2000. ACCD: an algorithm for comprehension-driven clustering. Proc 7th Working Conf on Reverse Engineering, p.258-267. https://doi.org/10.1109/WCRE.2000.891477

Wu J, Hassan AE, Holt RC, 2005. Comparison of clustering algorithms in the context of software evolution. 21st IEEE Int Conf on Software Maintenance, p.525-535. https://doi.org/10.1109/ICSM.2005.31

Yang TF, Jiang ZY, Shang YH, et al., 2021. Systematic review on next-generation web-based software architecture clustering models. *Comput Commun*, 167:63-74. https://doi.org/10.1016/j.comcom.2020.12.022

Zhang PL, Jiang YJ, Wei AJ, et al., 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. IEEE/ACM 43rd Int Conf on Software Engineering, p.50-61. https://doi.org/10.1109/ICSE43902.2021.00018

Zhao JF, Zhou JT, Yang HJ, et al., 2015. An orthogonal approach to reusable component discovery in cloud migration. *China Commun*, 12(5):134-151. https://doi.org/10.1109/CC.2015.7112036