International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 12, No.1 (Oct-2022)

https://dx.doi.org/10.12785/ijcds/120176

Characterizing Reverse Engineering Practices on Decayed Software Applications

Samir. Obaid[1], Ibrar. Arshad[2] and Muhammad Usman Abid[3]

[1]*Department of Computer Science, Capital University of Science and Technology, Islamabad, Pakistan*
[2]*Department of Computer Science, Capital University of Science and Technology, Islamabad, Pakistan*
[3]*Department of Software Engineering, Ripha International University, Islamabad, Pakistan*

Received 22 Jan. 2021, Revised 15 Jul. 2022, Accepted 23 Jul. 2022, Published 31 Oct. 2022

Abstract: Architecture reverse engineering is an approach to reproduce architectural contents once an application has deviated from its planned architecture. Unassisted understandings of an application by an individual, interviewing a person knowing the subject system, and computer-aided tools are few approaches that can produce architectural contents from a decayed software application. The former two approaches are helpful when an individual in the organization can understand the software application. Worst comes when computer-aided tools remain the only way to produce architectural contents from an application's source code. This research aims to identify architectural contents that industrial practitioners reproduce through reverse engineering, finding out the users of identified architectural contents and how the existing tools help in meeting industrial practitioners' needs. A qualitative study was performed to achieve the research objectives by choosing a homogenous sampling approach from the organizations where software applications were under gradual development for many years. Semi-structured interviews were conducted, and a coding approach was used to find out themes from transcribed data. We identified different architectural contents that practitioners produce from source code. Our findings show that practitioners use reverse engineering tools to produce architectural content from an application's source code. However, there are some architectural contents that practitioners need to reverse engineer, but no available tool produces those contents. The reverse engineering tools produce a wide range of architectural contents from source code but, contents visualization as required by practitioners is a challenge that needs to be addressed.

Keywords: software architecture, reverse engineering, code decay, architecture recovery

1. Introduction and Overview

Reverse engineering is a practice of extracting design from a finished product which can be software or a machine. In software engineering, reverse engineering is the process of obtaining design and other implementation-related information from a developed software [1], [2]. The importance of software reverse engineering was first realized when it became impossible to maintain or update software that has evolved over the years. The maintenance of legacy software is difficult because the documentation, over the years, became inconsistent with the source code, or it is difficult to understand the source code either because the length of code is too much or it is written in some old language. According to Ali [3], 50-90% of the work effort is usually spent understanding the source code whenever legacy software is to be updated. Through reverse engineering practices and tools, the effort required for this maintenance task can significantly be reduced.

Architecture reverse engineering is an approach for analyzing a system to identify its components, their interrelationship and create representations of the system in another simplified form or a higher level of abstraction [4], [5]. Architecture reverse engineering can also be explained as identifying the architectural contents of a system that are deviated from its planned architecture [6], or its architecture is never documented at all. There are several reasons for this deviation, for instance, the system was easy to develop at the early stages; therefore, no one was bothered to document architecture during further development and maintenance, the architecture documentation was considered a time-consuming activity, there exists an architecture document, but it was not updated with the upcoming changes of the system. Due to these reasons, the system source code becomes the only source to understand the system's major components and dependencies, design information, and quality aspects that different stakeholders require. The term code decay [6], [7], [8] is used if source code mismatches documented or planned architecture. The term architecture degeneration [9] is also used for systems where actual architecture mismatches the planned architecture. Parnas [10] uses the term software ageing for degraded system design and increased complexity. Once the system goes through many years of development activities, it tends to become complex,

E-mail address: samir.obaid@cust.edu.pk, ibrar.arshad@cust.edu.pk, obaid.musman@edu.pk

poorly structured and has little or no documentation. The system becomes hard to understand; any new decision becomes hard to undertake as it requires paying the extra cost of development and testing to ensure that new functionalities are added without placing any new problem. Architecture reverse engineering is helpful in this regard, as it recovers some valid architectural contents of the system. There are some tools that can reproduce architecture contents from source code information. However, it is important to know that produced architectural contents and their representations are acceptable for industrial practitioners who use reverse engineering tools. To address this challenge, the high-level goal of this research is to:

- Find out architectural contents that industrial practitioners need to produce from source code.

To meet the high-level goal of the research, we have set the following objectives:

- To find architectural contents that industrial practitioners reproduce from source code
- To find users of reproduced architectural contents
- To know how existing tools help reproduce required architectural contents?

To meet the first two research objectives, a qualitative study was performed where practitioners were interviewed to know about: how and what architecture contents they produce from source code? Furthermore, what is their aspiration for reverse engineering tools? In order to meet our third research objective, existing vendor tools were studied, and then their features were compared with survey results. Our finding showed that existing reverse engineering tools produce various architecture contents from source code information; however, there were architecture contents that need to be produced, yet none of the existing tools produces these contents from source code.
The remaining of this paper is organized as follows: Section 2 discusses existing tools. The research methodology is described in section 3. Finally, section 4 and 5 contain our findings from the qualitative study, and conclusion of work done and future direction respectively.

2. Existing tools

There are many standalone tools and plug-ins for major IDEs that recover architecture from application source code. These tools help improve comprehension of source code, check code quality against quality metrics, delta analysis to show the difference between two versions of the code, and code compliance to enforce architecture design decisions on code. Among earlier reverse engineering tools, Wong et al. [11] propose Rigi, a tool that produces flow graphs from application source code. The flow graph presents functions, functions calls, and data accesses. The produced flow graph can be much more detailed, depending upon several function calls in the application source code. To minimize the flow graph's noise or complexity, Rigi uses the cluster and filtering [6] technique. Through this technique, a user can group low-level entities (function calls) into high-level entities to obtain an abstract model that meets the user's reverse engineering goal. Rigi is a freeware with its source code, and pre-compiled downloads are available on their official website .[12] Rigi comes with extensive learning resources, user manuals, and sample programs of increasing complexity, demonstrating how Rigi can reverse-engineer software systems.
Systa et al. [13] propose Shimba, a prototype reverse engineering tool that is designed to understand java source code. Shimba analyzes java byte code and visualizes a subject system's static and dynamic components by customizing Rigi. These components include classes, interfaces, methods, constructors, variables, static initialization blocks, return types, and visibility. It also extracts relationship components from java byte code such as extension, implementation, containments, method call, and assignment. These extracted components can be visualized using Rigi dependency graphs. Shimba creates a sequence and state-chart diagrams from extracted components to visualize dynamic aspects of the subject system. A sequence diagram helps understand the relationships among different objects, whereas a state-chart diagram helps to understand the overall behavior of certain key objects. Shimba supports the model slicing technique, through which a user can filter out unnecessary details from the dynamic models to focus on particular parts of the target system. Imagix4D [14] is a reverse engineering tool that helps software developers to understand, document, and improve complex, third-party, and legacy source code. Imagix4D analyzes source code and produces different abstract views of the system, such as subsystem architecture (well-segmented subsystems with cleanly specified interfaces), package diagram, class diagram, functions, and variables dependencies control flow graphs. The tool also assesses analyzed code over a hundred different quality metrics to ensure that the software meets planned development criteria and determines where to focus testing efforts. Its delta analysis feature helps in visually reviewing the structural differences between different versions of the code. Imagix4D is available with the latest stable release since 2019, and it also has a rich user manual and technical support document.
Sangal et al. [15] propose Lattix [16] , a static analysis tool that reverse-engineers architecture from an application's source code. Architecture contents, reverse engineered by the tool, can be viewed in the form of graphs (lines and boxes) or dependency structure matrix (DSM). DSM is a simple, compact, and visual representation of system components (classes, packages) in the form of rows and columns. Like Rigi, the DSM feature of Lattix helps filter low-level architecture contents and create clusters to make a more abstract view of the system architecture. In addition, Lattix facilitates code compliance with existing architecture

by providing rules-based architecture enforcement features that enable the architect to define design rules and restrict the development team from violating those rules during the development phase. NDepend [17] is a static analysis tool that explores existing architecture from C# and Visual Basic code. Like Lattix,NDepend presents architecture details in the form of DSM that helps filter low-level components to visualize the high-level architecture of the system. In addition to DSM, the initially explored architecture can also be filtered with the help of a query language called CQLink. Through CQLink, a user can query for specific elements from the model by hiding other elements of the model, which helps a user to define their own way of presenting a lightweight architecture of the system. Like Lattix, NDepend provides rules-based architecture enforcement features that help users define design rules and then restrict developers from violation during release and eventually before committing to source control. NDepend comes with a compare build feature that can graphically visualize what has been changed between two code versions.BOUML [18] recovers UML diagrams from Java, PHP, C++, and Python source code. Along with generating UML models from the source mentioned above code, it also generates XML Metadata Interchange (XMI) that can be used by any other model generator tool to extract a model from XMI. This XMI file can also help generate diagrams other than UML and thus make it convenient to develop their own models for the representation of architectural contents. There are many plug-ins for eclipse IDE that visualize architectural details from Java source code. UML-Lab [19] supports round trip engineering, i.e., reverse engineering architecture from source code and generating code from architectural diagrams. The tool presents architecture by the UML class diagram, whereas it also facilitates creating customized models with the help of UML profiling. Another similar tool is ObjectAid [20] that creates a UML sequence and class diagram from Java source code. The tool helps automatic synchronization between code and diagrams. Any changes in the code are automatically reflected in the diagram without executing the model generation process again. AgileJ [21] creates a class diagram reverse-engineered from Java source code. Since generated diagrams can be cluttered; therefore, the tool also provides filtering ability to reduce the noise in the presented information. MaintainJ [22] visualizes light weight models from java source code with the help of a class and sequence diagram. In order to visualize a dynamic view, the tool analyzes the program call stack and presents stack traces by a UML sequence diagram. The tool can also visualize a call stack between two applications when running on two different Java virtual machines. For example, when an application calls a service (running on different JVM), the call flow across JVMs can be shown by a single sequence diagram. ModelGoon [23] visualizes component dependency by reverse engineering Java source code into package and class diagrams. The tool also produces a sequence and collaboration diagram to present a dynamic behavior of a selected codebase. Diver [24] [25] is a code analyzer that records traces of running programs and visualizes a sequence diagram of captured traces. It also helps the user to explore the code from a sequence diagram to gain the same perspective from the code that a user understands from a sequence diagram. We found a large number of reverse engineering tools that produce architectural content from an application source code. In Table I, we did our best effort to identify available reverse engineering tools and describe them. Tools come with various static and dynamic models to facilitate users of the tool with a wide range of information. In the presence of diverse reverse engineering tools, it is important to know about practitioners' needs and challenges that they face while reverse engineering architecture contents from application source code.

3. Methods

We have chosen a qualitative research method and conducted interviews to gather focused, in-depth, and qualitative textual data from respondents. As described in Fig. 1, the research process started by formulating research objective. We have already mentioned our research objective in section 1 of the paper. Following research objective, a vendor study was performed to understand existing reverse engineering tools and their features, as described in Section 2.We then selected the study sample, conducted interviews, and executed a qualitative coding process described in subsections A, B, and C.

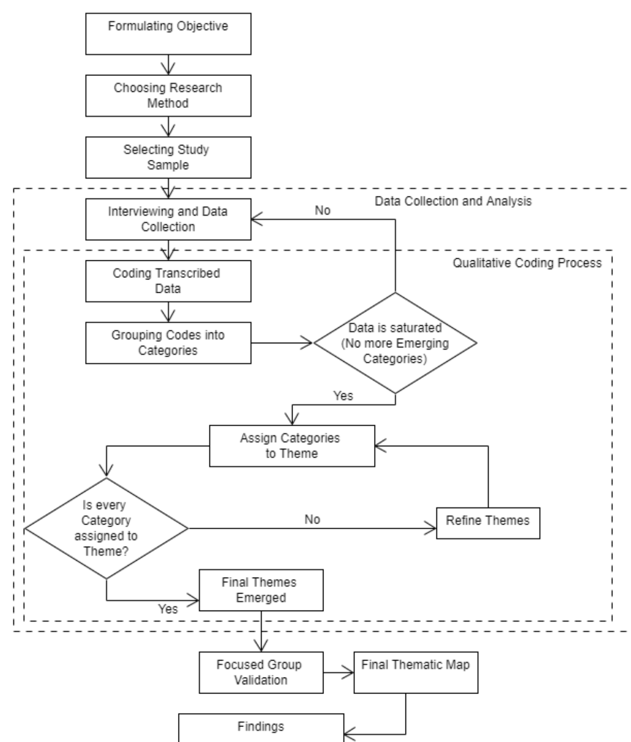


Figure 1. Methodology

TABLE I. Reverse engineering tools

Tool	Input/Languages Support	Brief description and architectural contents produced as output
Rigi [12]	C, COBOL, C++	Rigi is a reverse engineering tool that generates flow graphs from application source code. It helps in program understanding by providing an interactive graphical editor where a user can further abstractor granularize generated model diagrams.
Shimba[13]	Java	Based on Rigi, Shimba helps in program comprehension by visualizing static and dynamic models from program source code. Classes, interfaces, methods, constructors, variables, static initialization blocks, return types, and visibility.
Imagix4D [14]	C, C++ and Java	It is a reverse engineering tool that helps to visualize legacy code written in C or Java language. The high-level visualized models include subsystem architecture diagrams and UML class diagrams. More detailed models include sequence diagram, definition use operation on variables and data flow diagram.
Lattix [16]	C,C++, Java, Python,Fortran, and ADA	The significant contribution of Lattix is a generation of DSM from source code analysis. DSM is a simple, compact, and visual representation of system components (classes, packages) in the form of rows and columns. Lattix helps to understand, refactor and control architecture erosion with the help of rule-based architecture enforcement.
NDepend [17]	.Net managed code includes C# and visual basic	NDepend is a static analysis tool that can visualize C# and visual basic source code. Like Lattix, it generates DSM to visualize components and their dependencies.
JDepend [26]	Java	JDepend visualizes packages and dependencies from java source code. It traverses java classes and packages to generate quality metrics. Quality metrics include extensibility, reusability, and maintainability to manage and control package dependencies effectively.
BOUML [18]	C++, Java, PHP	BOUML can reverse engineer C++, java and PHP code and visualize them by UML diagrams. It can also be used to generate XMI from code reverse engineering. The XMI is lightweight artifacts that software architects can use to generate their own diagrams, which they better understand.
FUJABA [27]	Java	An open-source model-based reverse engineering tool that helps to generate high-level design artifacts from program source code. High-level design artifacts include the UML class diagram and activity diagram. The tool also provides various plug-ins such as Archimetrix, which helps to generate component-based software architecture from application codebase.
CPPX [28]	C, C++	A tool that extracts fact base from C++ source code. The fact base is a graph whose vertex represents classes, functions, expressions and variables. The edges represent the relationship between identifiers (functions and variables) to their declaration, methods call their targets, and objects to their types.

Continued on next page

TABLE I – *Continued from previous page*

tool	Input/Languages Support	Brief description and architectural contents produced as output
QLDX [29]	C++	QLDX is a reverse engineering tool for the visualization of software architecture from application source code. This tool produces facts based on application source code and then uses LSEdit to visualize the application architecture graphically.
IBM RSA [30]	Java	IBM RSA can be used for model-driven software engineering and reverse engineering. In addition, it can be used to generate UML structural and behavioural diagrams from application source code. Its most popular design artifacts are the UML class diagram and UML sequence diagram.
Zynamics BinNavi[31]	Binary code	BinNavi is a reverse engineering tool that was built to generate control flow-based analysis artifacts from binary code. It can be helpful to assist vulnerability researchers who look for vulnerabilities in disassembled code. The tool provides a powerful debugger to locate relevant code quickly. The tool also facilitates adding additional plug-ins to meet the specific goals of the users.
UML Lab [19]	Java	UML Lab is a lightweight modelling tool that can be used in forward engineering (from modelling to code), reverse engineering (from code to model) and round-trip engineering (synchronize code and model). The tool generates a UML diagram from the application's source code. However, it also facilitates defining your own template and reverse engineer code to those defined templates.
Diver [24]	Java	Diver is an open-source tool and eclipse plug-in that helps to analyze and visualize java source code. It can generate a UML sequence diagram from java source code and thus provide a dynamic behavioural view of the underlying codebase.
Eclipse AgileJ [21]	Java	AgileJ is an eclipse plug-in that analyzes java source code to identify classes and relations among classes. The tool also helps to identify and visualize design patterns from application source code. It also round-trip engineering feature which can automatically synchronize code changes into diagrams.
MaintainJ [22]	Java	Generates class and sequence diagram at runtime while executing any scenario. It can also visualize call stacks among applications deployed across multiple Java virtual machines.
ModelGoon [23]	Java	The tool visualizes package dependencies from java source code. It also produces high-level design artifacts such as UML class and sequence diagrams.
Eclipse ObjectAid [20]	Java	ObjectAid is an eclipse plug-in and a reverse engineering tool that can visualize java source code by UML class diagram. The tool also provides features to cleanup diagrams, such as hide variables and methods from the generated model. In this way, it helps to focus on classes and associations rather than diving into details.

The coding process stopped when we reached to data situration state and final themes emerged. A focused group session was conducted to obtain experts review on obtained themes. finally, we have discussed our findings from analysed data in section 4 of the paper.

A. Sample study

In our study, a sample size of twelve participants was chosen, which is the minimum recommended sample size for data saturation [32]. In addition, we used a homogeneous sampling [33] approach, which can be described as "gathering people in a similar situation with a similar background". The rationale behind homogeneous sample was to select only those organizations where software product has undergone few years of development practices. During this time, the application has reached a certain level of complexity, and practitioners tried reverse engineering during the development process. While searching for the required sample, we found the following similar situations with our respondents:

1) Product time in the market: The first common aspect of all respondent companies was their long product life in the market. As described in Fig. 2, the minimum product life is seven years, whereas the maximum product life is seventeen years. Respondents told us that a significant amount of code was written years ago. Due to this, developers who have written that code hardly memorizes it. New developers who have joined the team find it difficult to understand the code without their colleagues' assistance. The worst comes when old developers have left the company, and new developers try to understand the code with unaided browsing.

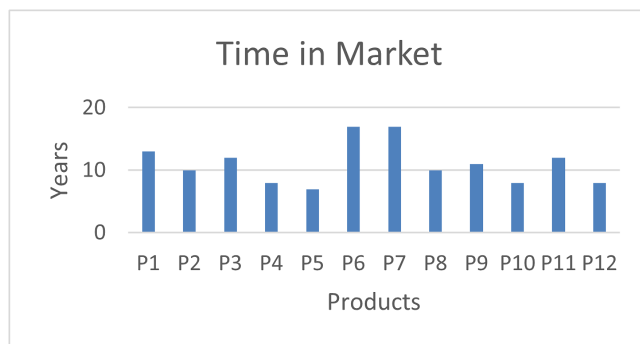


Figure 2. Products time in the market

2) Team size: The second common aspect among respondent companies was their team size. Fig. 3 describes team size, which ranged from 65 members to 180 members, and it included: developers, quality assurance (QA) engineers, technical support team, architects, team leads, technical report writers, and company's high-ups who may also be non-technical personals. Respondents told us that there are varying interests of team members which depend upon their role. We have thoroughly discussed those interests in our findings. For instance, if two development groups worked concurrently on different tasks, they must know about execution flow, input and output of developed components, and code organization. It was not feasible to assist other groups with developed code, especially when there were rapid release cycles.

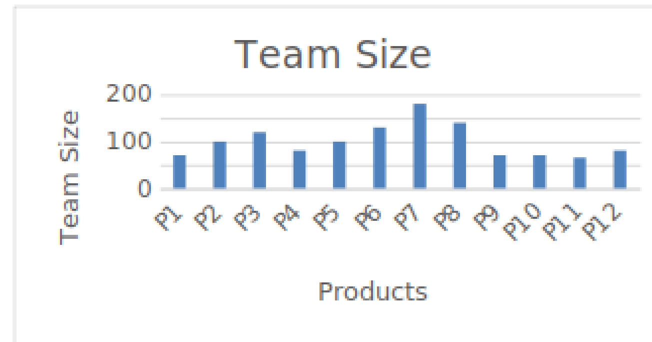


Figure 3. Team Size with respect to the number of people

3) Product size: The third common aspect among respondent organizations was lines of code (LOC). As described in Fig. 4, the code size ranged from 0.8 million LOC to 4.0 million LOC. As mentioned by respondents, having a huge code has several challenges: it is difficult to understand, communicate, and properly document it.

B. Interview and data collection

The interviewer visited each respondent in person for an interview session, and data were recorded in audio and hand-written form. In addition, follow-up interviews were conducted in order to confirm and clarify data where required. Our in-depth interview had many questions for each

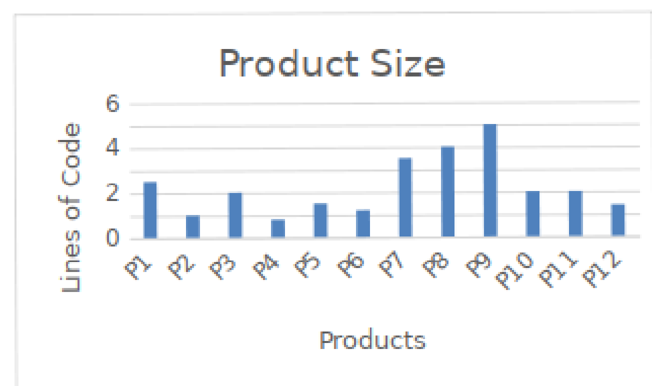


Figure 4. Product size with respect to lines of code

respondent; however, we planned some starting questions for each interview. For instance:

1) What are the architectural contents that you reproduce from the source code?
2) Who are users of reproduced architecture content?
3) How are existing tools helpful in reproducing required architectural contents?

Each interview lasted from 40 to 60 minutes, and data were transcribed on paper or audio form after having the respondent's consent. We have also conducted four follow-up interviews with four respondents after having confusion in transcribed data. Data were transcribed and then analyzed to find common patterns and to place forward general observation.

C. Qualitative coding process

1) Content Analysis: We have selected conceptual analysis as a content analysis approach, i.e., the focus was on the existence of concepts and frequency of occurrence of those concepts. According to Carley [34], several steps must be followed before conducting the content analysis. Table II briefly describes those steps, along with the decision taken by the author at each step. At the end of the analysis, reverse engineering practices were categorized, and they were used to find major architectural contents produced by the practitioners from source code and stakeholders of those architectural contents.
2) Identifying Codes: Coding as a method of organizing transcribed data is widely used in qualitative research [34], [35]. We have chosen a top-down coding approach (aka theoretical coding approach) through which we just coded those concepts that were relevant to our concerns as expressed in research questions. The coding task was executed by two researchers where one researcher identified codes from transcribed data and the other reviewed the codes to omit chances of errors. While highlighting codes, we have adopted the Boyatzis [36] approach in which a researcher briefly describes coded concepts and how a researcher highlights codes from transcribed data and the corresponding excerpt of transcribed data. The coding process was stopped after few iterations, and identified concepts were: sequencing GUI snippets, configuration details, roles, and access control, static and dynamic execution flow, APIs with required and provided services, code annotation, modules re-organization, visualization of ERD, data tables, and data rows presentation. Once codes were identified, then they were organized into themes as described in Fig. 5.
3) Themes: Themes are outlines (also known as patterns) that group together common codes to address research questions [35], [**?**]. Once gone through codes, we observed some obvious themes that emerged, and codes fit into them. For example, some frequent codes such as user stories, code snippets, and GUI snippets are grouped to set up a Scenarios theme. We identified a total of five themes at the end of this process and described them in Fig. 5. Themes are user stories, program execution flow, service APIs, code organization, and database view. Each theme further contained a sub-theme as described by rounded corner boxes in Fig. 5. For example, static and dynamic execution flow practices are categorized into a theme which is named program execution flow. Service APIs consolidate all practices where practitioners try to produce service details from the codebase. The database view contains practices: ERD generation from database schema, table's view and associations, and finding redundant tables and fields. User Stories consolidates sequencing GUI snippets, configuration details, and access control policies. Finally, codebase organization practice consolidates code annotation and modules re-organization.

D. Focused group validation

Focused group validation is conducted in order to obtain feedback from community members on the accuracy, the validity and the appropriateness of the research findings [37]. We have chosen a small group of seven participants who contributed to open discussion on our proposed thematic map. Four participants had a doctoral degree in the computing field. Three of them are currently working as permanent faculty members in computer science, and one, working as an application architect in an IT organization. Three participants had master degrees in computing and are working as senior software engineers in IT organizations. One of the participants has played the role of a moderator who helped in the smooth execution of the session and ensured discussion on each question. The session lasted for two hours, and participants highly endorsed the proposed.

E. Findings

To answer our first research question, Table III describes architectural contents, challenges faced by practitioners to produce those contents, and strategies adopted by industrial practitioners to generate these contents. The following section describes what respondents demanded under each architectural content.

1) Program execution flow describes exaction sequence, either control flow or data flow, once the system receives input. We observed that all respondents were reproducing program execution flow details from the application source code. For instance, one of our respondents mentioned that 'We document screenshot of GUI and back-end code because our fellow developers need them to understand program execution flow by GUI snippets with code behind'. After asking about this activity's practice, the respondent mentioned recording input events and execution flow (both code and GUI screens) against that input event. Team members used this execution sequence

TABLE II. Steps for conducting content analysis [34]

Tool	Output artifacts
Decide the level of analysis	We have decided to code for both single and set of words that appear in the text (transcribed data from respondents)
How many concepts to code for	We have decided to code for relevant concepts as they appear in the text)
Code for existence or frequency of concepts	Since we are not sure about the significance of frequency in this research topic, therefore, we have decided to code for the existence of the concept even if it appears multiple times.
Distinguish between concepts	We have decided to highlight the concepts exactly as they appear in the text. If two concepts with different phrases appeared, then there is a chance to distinguish them based on their relation to reverse engineering practices
Rules for coding the text	One researcher has highlighted the concepts and revised them by another researcher,
Code the text	We have chosen the manual coding technique by using paper and a highlighter. Computer-aided tools are not used due to lack of practice.
Analysis of result	We have analyzed the data to determine respondents' expectations from reverse engineering tools and architecture contents that they reproduce from source code. We also compared their expectations with available reverse engineering tools.

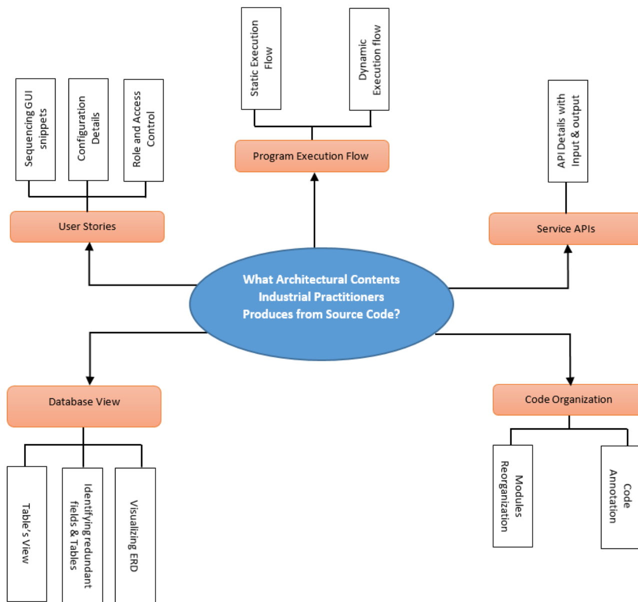


Figure 5. Thematic Map to Present Architectural Contents that Practitioners Produce from Source Code

to understand the execution sequence with no or minimum assistance from other fellows. Respondent also mentioned that sometimes they had to demonstrate an application to the company's high-ups (not from the IT domain). With the help of this document, the high-ups understood how their aspirations are realized in a developed system. Another respondent mentioned that; 'program execution flows are maintained with the help of boxes and arrows to assist team members and 'execution flows in the business layer are important for us, and we document it from code'. This respondent showed the same intent as the previous one, i.e., helping developers understand program execution flow against the input event without any assistance and communicating work done with high-ups. From the content analysis, we observed that screenshots of GUI and code and lines and boxes diagrams were two common approaches by respondents to capture execution flow details. Since respondents were capturing system-wide execution sequence instead of a particular input event, therefore, we attributed this activity as static execution flow details of the program. Respondents also showed their interest in extracting dynamic execution flow information from program source code. For instance, a respondent mentioned that 'We maintain a log of program execution traces to find the exact point where fault exists'. Another respondent mentioned that 'We record input data and execution sequence to observe where does fault exist'. When we asked about the purpose, respondents mentioned that they mostly failed to reproduce the client's reported bugs. Therefore, they have developed a tool that logs client-side configurations, input events, and failure occurrences. Once received a log, then another version of the tool is used on the developer end, which sets the same configuration, provides input data, and records code execution sequence to reach fault position. Another respondent followed the same practice by mentioning that we have an in-house developed tool that records input events and program execution trace against event'. It helps them to figure out specific input events that caused failure and program execution trace until failure occurs. Since respondents were recording execution sequences for a particular input(s), therefore, we attributed this activity as dynamic execution flow

TABLE III. Architectural contents reverse engineered by practitioners

Architectural content	Characteristics of Architectural Contents	Challenges Faced by Practitioners	Strategies Adopted by Industrial Practitioners to Generate these Contents
Program Execution Flow	Capturing input event to program, identification of program execution path from source code, finding dependencies among modules and business flow	'Execution flows in the business layer are important for us, and we document it from code . . . ' 'New developer uses to record input event and execution sequence on a paper . . . '	tatic Execution Flow 'program execution flows are maintained with the help of boxes and arrows to assist team members"We keep execution sequence along with code screenshots . . . 'Dynamic Execution Flow 'We maintain a log of program execution traces to find the exact point where fault exists."We maintain a log for execution traces of program to investigate the exact point where the system gets crashed."We observe input data, methods sequence, and the point where application crash."We have an in-house developed tool that records input events and program execution trace against an event.'
Service APIs	Contents related to documenting purpose, input, and output of reusable program libraries and web services	'development team frequently modify already developed APIs to generate a new type of record . . . '	API Details with Input and Output'Frequently document the purpose of an API, with input and output."We have to frequently re-document the purpose of APIs because there are frequent changes by the development team . . . '
Database View	The understanding database schema, identification of redundancy (both tables and fields)	database schema evolves with time, and it became hard to work on report generation, adding new records and understand the entire schema, especially by new team members. . . '	Visualizing ERD 'inspect schema to generate ERD from it."We revise our databases to find out common fields across tables'Table's View 'We have developed an in-house tool named that takes a table name as input and displays all associated table'
User Stories	Contents related to documenting and maintaining user stories from the source code information	'QA team cannot manage their work due to a lack of knowledge about interfaces and APIs. We, therefore, work with the development team to reproduce user stories . . . "Sometimes we need to create models to share our user stories with HR and technical support team to let them use it . . . '	Capturing and Sequencing of GUI Snippets'for each use case, we document a sequence of GUI snippets along with backend code. It helps both non-technical and technical personals to understand the use case.'Documenting Configuration Details'Keep configuration details required to run an application.'Documenting Role and Access Control 'We maintain role table to keep access and authorization on user stories'
Code Organization	Re-modularization of existing code, the Understanding purpose of code, and identification of reusable code from the codebase	'after gradual development, some code segments become complex and contain reusable chunks. . . "It mostly happens that the developer didn't bother to annotate comments on code or comments are vague, such situation becomes very problematic to other developers'	Code Annotation'we often write the comments on the top of the store procedures and in the code so that we may get instant knowledge about it."write details about complex code alongside to guide other developers when they modify code.'Modules Reorganization 'inspect our code to segregate code chunks into modules"we are manually inspecting our code to find reusable code chunks and then reorganize the code.'

details of the program.

2) Service APIs: Services APIs refer to services that an application provides in the form of web services. Analysis of respondent's data showed that respondents were documenting API details which purpose was manifold. For example, one of our respondents stated that 'we maintain List of API, with their purpose, input and output'. After asking about this activity's practice, the respondent mentioned that they were maintaining API details in a spreadsheet to let new developers understand the purpose of an API and input data that APIs take and output data that it produces. Another respondent mentioned that they 'frequently document the purpose of an API, with input and output'. While asking about the purpose, the respondent mentioned that their testing team required APIs details to implement or modify unit tests to validate those APIs. Since the development team frequently changed those services, they were required to re-document the purpose of APIs, inputs are taken, and output. Some other respondents were annotating APIs details in a codebase, and they were using integrated utilities with IDE to produce API details.

3) Database View: Database view refers to entity types, entity fields, and relation and type of relationships among entities.
We found many respondents were reproducing high-level models from the developed database schema. As described in Table III, one of our respondents mentioned that 'database schema evolves with time. It became hard to work on report generation, adding new records, and understanding the entire schema, especially by new team members. One of our respondents mentioned that 'We have developed an in-house tool named *** that takes a table name as input and displays all associated tables'. After asking for purpose, the respondent said they had planned to write a short and optimal query for report generation. For a particular table, the tool helped them list down table fields, a relation of a selected table with other tables based on its primary key, relation with other tables on a foreign key that it possessed. Another respondent mentioned that 'we revise our databases to find out common fields across tables'. The respondent told us that their database schema grew with time and, even after careful development, some data fields become redundant across multiple tables. Thus, they had to revise the entire schema to find redundant fields in the database schema. We also found many other respondents showing their aspiration to reproduce entity-relationship Diagram (ERD) from a database schema. For instance, a respondent mentioned that: 'our team like to have a tool that could be connected to the database, and it produces ERD', 'I would like to have a diagram which would help me in explaining the DB to my fellows' and 'ERD generator from schema would be helpful tool'. Instead of spending time on ERD design, practitioners spent time developing database schema and report generations. The schema started growing with gradual development activities, and then, a lightweight representation like ERD was required to understand and discuss the database with fellows.

4) User Stories: We found eight out of twelve respondents tried to produce user stories from application source code.
One of our respondents expressed the challenge as the 'QA team is unable to manage their work due to a lack of knowledge about interfaces and APIs. We, therefore, work with the development team to reproduce user stories. Respondents were using different tactics to reproduce user stories. For instance, a respondent mentioned that: 'for each use case, we document sequence of GUI snippets and back-end code'. Another respondent mentioned that 'we communicate with the client on email and give them demos based on the system's GUI screenshots. Sequencing GUI snippets of an application was more convenient than any other way to communicate and document user stories. In addition to the GUI sequence, some respondents were found to document the configuration details (environment setup) required before starting the user story. Examples of configuration details, as mentioned by respondents, were: configuring endpoints in distributed systems, loading data to memory, services availability, etc. The third important aspect that respondents were documenting was roles and access control policies that helped them to reserve information about different users' roles and authorizations with each role.

5) Code Organization: Code organization refers to remodularizing code, annotating existing code, and identifying reusable components from the codebase. Once code undergoes a few years of development, it becomes complex and hinders new developers' ability to understand it easily. For instance, a respondent mentioned that 'after gradual development, some code segments become complex and contain reusable chunks…'. Therefore, they were inspecting existing code to find out redundant components and code complexity that raised due to continuous development. Another respondent mentioned that 'It mostly happens that the developer didn't bother to annotate comments on code or comments are vague, such situation becomes very problematic to other developers'. While explaining the statement, the respondent mentioned that sometimes it becomes difficult to understand and modify the code script written years ago. Therefore, they execute an activity to annotate, explain code scripts, and map code with specification documents. Doing so helps new developers to understand the code and further work on it. Another respondent mentioned that 'we were having a tangled code of different features. So, we

identified tangled code, modularized the codebase, and defined a kind of sub-MVC project within a large project'. The respondent mentioned that they had violated design rules during gradual development for few years. Thus, code becomes complex, and re-modularization was the only solution. Now, any new development requires peer review by experienced developers to avoid design violations. The same situation was faced by another respondent who mentioned that 'we were reviewing 5 years old testing application because test cases about different features of an application were tangled among different test suites. Therefore, their team revised an application to segregate tangled test cases into their appropriate test suites.

F. Who are users of reproduced architectural contents?

Since users of architectural contents are actual stakeholders and therefore, it is important to know about users of reproduced architectural contents. From the content analysis, we observed that major users of architectural contents are the development and testing team. In addition to that, organization high-ups (who also have non-technical personals) and end-users also have an interest in produced architecture content. Table IV describes architectural contents reproduced from source code, users of reproduced architectural contents, and purpose/goal that the user wants to achieve after having required architectural contents.

G. How are Existing Tools Helpful in Reproducing Required Architectural Contents?

Although existing tools reproduce various architectural contents from source code, however, there are practitioners' needs to be addressed or gaps to be fulfilled by existing tools. In the following section, we describe features of tools conformance to practitioners' requirements, future work or opportunities for vendors of reverse engineering tools, and limitations in the selection, adoption, and use of existing reverse engineering tools.

1) Existing Reverse Engineering Tools That Meet Needs of Industrial Practitioners: We observed that most reverse engineering tools focus on producing execution flow from the source code of an application. Function dependencies [8], activity flow, dependency graphs [10], [11], [13], [14], [20], sequence diagram [10], [11], [15], [16], and flow graph [8], [13] are kind of execution flow details which are reproduced by existing reverse engineering tools. We found that our respondents were also reproducing execution flow details from source code, and thus this feature of existing tools has a high tendency toward respondents' aspirations.
   One of the respondent's aspirations from reverse engineering tools was to produce ERD from a database schema. There are database reverse engineering tools such as SchemaCrawler [28] , MySQL Workbench [29] , and SQLDatabaseStudio [30] are a few to name. SchemaCrawler is an open-source database schema discovery and comprehension tool that allows generating diagrams from SQL code. MySQL Workbench generates ERD from a physical schema that a user can use to understand, update and push back changes to the schema through forwarding engineering. Our respondents also expressed their need for producing services APIs details from the source code of an application. There are integrated development environments (IDE) that provide a view of services by displaying service title, purpose, and contract (input data and output). Two important respondents' aspirations, i.e., tools to work as a plug-in with IDE and produce architectural contents in their desired format, can make these tools useful for industrial practitioners.
2) Future Work for Reverse Engineering Tools to Address Needs of Industrial Practitioners: Our respondents talked about some architectural contents and were also reproducing them from source code, yet current reverse engineering tools do not reproduce these contents. In the context of database view, our respondents were interested in viewing individual table details: table fields, meta-data, relation to and from one table to another based on the primary and foreign key. Similarly, respondents were interested in examining database schema to highlight redundant table fields which were added due to gradual schema development by different developers. Although there are different database reverse engineering tools, none of them addressed these specific concerns. Developing a reverse engineering tool that could address these concerns can be possible future work for tool developers.
   Some of our respondents have developed tools to capture input system events with input data (if available) and runtime execution flow against the event. Respondents were also capturing GUI and code snippets to document execution flows. However, tools are still required to address this challenge. Respondents have also expressed their aspiration that such tools need to work as a plug-in with IDE rather than to migrate the entire code to some new environment.
   Respondents were producing system usage scenarios from source code information. They were capturing screenshots along with back-end code snippets. The purpose of these scenarios was to maintain the technical documentation for their clients, helping their developers understand system usage by observing the GUI sequence and back-end code against each GUI frame. Practitioners were also reproducing user stories in the form of sketches and natural language expressions to let non-technical people in the organization understand the purpose of the system. Unfortunately, we didn't find any reverse engineering tool that produces user story details from source code information. We believe that developing such a scenario generator tool could have potential use for

TABLE IV. Users of produced architecture contents

Architectural Contents	Users	Aspiration
User stories	Testers	To understand system features in order to develop test cases.
	Developers	To help the developer understand user stories and the source code behind each user story.
	Organizational High-ups	To know about development progress and how the system meets users' needs.
	End users	To understand how to meet their goals by understanding the flow of the system's feature.
Database view	Developers	Understand tables, the relationship among tables, and write down an optimal query to fetch reports from the database.
Execution Flow	Developers	To find out input event, then control flow and data flow against input event that could better help them to understand fault in the code
Program APIs	Developers	To understand system features in order to develop test cases.
	Testers	To know about development progress and how the system meets users' needs.
Code Organization	Developers	To understand code complexity and then refactor (for example, modularize) the code.

practitioners.

Our respondents inspected the existing codebase to highlight complex code segments due to gradual development activities and code evolution. Tools are required to reorganize complex code by identifying reusable code segments. Our respondents mentioned that they comment on top of code segments, i.e., application code and stored procedures. The objective was to let other people in the team understand the purpose of the code easily. Although it was recommended for each developer to keep working on this activity while developing code, they still have to repeat these tasks months after development. Reverse engineering tools that could help practitioners in producing such documents could have potential use for practitioners.

Similarly, respondents were found inspecting existing code to identify tangled code, complex code bocks, and non-reachable code. Such complexities appeared due to gradual development and code evolution by multiple developers and a developer's premature decision at the early stages of development. Due to these and many other reasons, the code becomes messy and complex and hence, needs to be further modularized. Tools can be developed that could help the development team to highlight code complexities during or after development.

3) Limitations in the Selection, Adoption, and Use of Existing Reverse Engineering Tools: Apart from the architectural contents that current reverse engineering tools reproduce, some other factors influence selection, adoption, and use of reverse engineering tools. We observed that development teams were reluctant to use any off-the-shelf reverse engineering tool unless there is no other way to reproduce architecture contents from source code. The development teams were willing to put effort into unaided browsing or to contact knowledgeable persons rather than exporting code to any reverse engineering tool to produce architecture contents. We suggest that future reverse engineering tools be pluggable with the integrated development environment (IDE) to let practitioners produce architecture content without exporting their code.

Existing reverse engineering tools provide an abstract and lightweight representation of the system either by using UML (accepted as de-facto standard) or tools' own modelling notations. Such models would have good empirical results on the system's understandability in a closed environment; however, after analyzing respondents' data, we found that respondents are reluctant to use any modelling notations or language that requires expertise to use. As one of our respondents mentioned that "architectural contents are reverse engineered from code, but we do not draw UML diagrams". Similarly, another respondent mentioned that "we sketch lines and boxes to document execution flow". Respondents drew sketches (having their own notations) on board or paper, and then they were made part of the document. We suggest that tools enable users to define their modeling notations, maybe because users already use these notations in their organizations.

We observed that existing reverse engineering tools produce abstract and lightweight system models such as data flow diagrams, package diagrams, class diagrams, etc. These models would be having a high impact on understandability and communication at

the early stages of development when code is not developed. However, once an application is developed, these models may not be sufficient for developers to understand the developed system. Once code is available, then the most desirable model for understanding the system is code itself. As our respondents mentioned that, 'execution flows are maintained with the help of code, and GUI snippets,' and 'code is reorganized by inspection of code itself' etc. therefore, we suggest that any reverse engineering tool to be developed in the future should provide close resemblance with code for the ease of users.

H. Validity Threats

This research aimed to identify architectural contents that industrial practitioners use to produce from source code and their expectations from automated tools to produce these contents from source code. To identify the automated tools and the architectural contents, we performed a literature review. Digital libraries like google scholar, IEEE, ACM, Spring link, etc., as well as some grey literature, were searched for our research purpose. However, there is a chance that we missed some tools due to their unavailability in these libraries. Also, as we included the tools and architectural contents in the English language, therefore, we might have missed some literature that might be available in some languages other than English.

We have chosen the sample size of twelve participants in our study. However, the number was not fixed in advance. Rather, we have stopped the data collection process when it is observed that there are repeating codes and categories in transcribed data.

We selected 12 interviewees from different organizations with different expertise, experience, and job responsibilities for our study. Since all these individuals have international projects, thus all of them have experience with a multicultural environment. Therefore, our interview supports the generalization of the result.

Another validity threat to the study was the language barrier as all interviewees have different first languages. This issue was addressed by conducting the English language interview since all interviewees were proficient in the English language. However, there may still be a chance of misinterpretation of some sentences.

Before the start of the interview, all the interviewees were briefed about the interview process and their responsibilities, along with the general terminologies. Thus, before starting the actual interview, all the interviewees and interviewers were on the same page in all these aspects.

I. Conclusions

In this research, our efforts were to identify architectural contents that industrial practitioners use to produce from source code, and they also expect automated tools to produce these contents from source code. To meet this goal, we have set our objective as: identify architecture contents that practitioners reproduce from source code, identify users of produced architectural contents, and find out how existing tools help meet the needs of industrial practitioners. To meet our first two research objectives, a qualitative study is performed to select respondents from a homogeneous group. Respondents had experienced producing architecture contents from source code when code had become complex due to continuous development from the last few years. From our analysis, we found that major contents that practitioners produce are: user stories, database views, execution flow, Program APIs, and code organization. We found that major stakeholders of produced architectural content are the development team, testing team, organizational high-ups, and end-users. To address our third research objective, we have studied existing tools and presented their purpose, input artifacts that tools take, and output models that they produce. Then, we compared the tools feature set with our analysis results. We also found architectural contents that existing reverse engineering tools produce, and they are also expected by industrial practitioners, such as program execution flow, ERD from the database schema, etc. However, practitioners require various other architectural contents, but existing tools do not support them. We also observed that existing reverse engineering tools provide architectural contents in modeling notations that are either UML or vendor's defined notations. However, a tool must be flexible in letting its users define their own notations convenient for them to understand easily. It must also be considered that merely providing modelling notations to present high-level design or architecture may not be sufficient for users to understand the system. Once there is code in hand, then the high-level design may also include (or at least provide a mapping to) the code because the codebase is much familiar stuff for practitioners once an application is developed. Any reverse engineering tool to be developed in the future may also focus on practitioners' ease in the development environment. As we stated in the previous section, practitioners are reluctant to export their code to any off-the-shelf tool to reverse engineer architectural content. A tool that could be integrated with IDE could be having various advantages to practitioners, such as the ease with respect to use and efforts.

References

[1] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 47–60.

[2] G. Canfora and M. Di Penta, "New frontiers of reverse engineering," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 326–341.

[3] M. R. Ali, "Why teach reverse engineering?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–4, 2005.

[4] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

[5] D. R. Harris, H. B. Reubenstein, and A. S. Yeh, “Reverse engineering to the architectural level,” in *1995 17th International Conference on Software Engineering*. IEEE, 1995, pp. 186–186.

[6] L. Hochstein and M. Lindvall, “Combating architectural degeneration: a survey,” *Information and Software Technology*, vol. 47, no. 10, pp. 643–656, 2005.

[7] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[8] C. Stringfellow, C. Amory, D. Potnuri, A. Andrews, and M. Georg, “Comparison of software architecture reverse engineering methods,” *Information and Software Technology*, vol. 48, no. 7, pp. 484–497, 2006.

[9] M. Lindvall, R. Tesoriero, and P. Costa, “Avoiding architectural degeneration: An evaluation process for software architecture,” in *Proceedings Eighth IEEE Symposium on Software Metrics*. IEEE, 2002, pp. 77–86.

[10] D. L. Parnas, “Software aging,” in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 279–287.

[11] K. Wong, S. R. Tilley, H. A. Muller, and M.-A. Storey, “Structural redocumentation: A case study,” *IEEE Software*, vol. 12, no. 1, pp. 46–54, 1995.

[12] Rigi, “A visual tool to understand legacy system.” [Online]. Available: http://www.rigi.cs.uvic.ca/

[13] T. Systä, K. Koskimies, and H. Müller, “Shimba—an environment for reverse engineering java software systems,” *Software: Practice and Experience*, vol. 31, no. 4, pp. 371–394, 2001.

[14] Imagix4D, “Reverse engineering tools.” [Online]. Available: https://www.imagix.com/

[15] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, “Using dependency models to manage complex software architecture,” in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005, pp. 167–176.

[16] L. Inc, “Ldm tool.” [Online]. Available: https://www.lattix.com/

[17] P.Smacchia, “code quality with ndepend.” [Online]. Available: https://www.ndepend.com/

[18] B. Pages, “Bouml a free uml tool box.” [Online]. Available: https://www.bouml.fr/

[19] Y. Gmb, “Uml lab.” [Online]. Available: https://www.uml-lab.com/en/uml-lab/

[20] U.ObjectAid, “Objectaid uml explorer.” [Online]. Available: https://marketplace.eclipse.org/content/objectaid-uml-explorer

[21] “Agilej structureviews.” [Online]. Available: https://marketplace.eclipse.org/content/agilej-structureviews

[22] C. Kothaplli, “Reverse engineer java.” [Online]. Available: http://www.maintainj.com/

[23] “Modelgoon uml4java.” [Online]. Available: https://marketplace.eclipse.org/content/modelgoon-uml4java

[24] H. Cai and R. Santelices, “Diver: Precise dynamic impact analysis using dependence-based trace pruning,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 343–348.

[25] “Diver.” [Online]. Available: https://marketplace.eclipse.org/content/diver-dynamic-interactive-views-reverse-engineering

[26] “jdepend.” [Online]. Available: https://github.com/clarkware/jdepend

[27] U. Nickel, J. Niere, and A. Zündorf, “The fujaba environment,” in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 742–745.

[28] “Cppx.” [Online]. Available: https://www.swag.uwaterloo.ca/cppx/

[29] “Software architechture group.” [Online]. Available: https://www.swag.uwaterloo.ca/qldx/index.html

[30] “Ibmrational software architect.” [Online]. Available: https://www.ibm.com/products/rational-software-architect-designer

[31] “zynamics.com - binnavi.” [Online]. Available: https://www.zynamics.com/binnavi.html

[32] V. Braun and V. Clarke, “(mis) conceptualising themes, thematic analysis, and other problems with fugard and potts’(2015) sample-size tool for thematic analysis,” *International Journal of social research methodology*, vol. 19, no. 6, pp. 739–743, 2016.

[33] D. Silverman, *Doing qualitative research: A practical handbook*. Sage, 2013.

[34] K. Carley, “Coding choices for textual analysis: A comparison of content analysis and map analysis,” *Sociological methodology*, pp. 75–126, 1993.

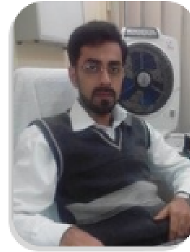
[35] J. Saldaña, *The coding manual for qualitative researchers*. sage, 2021.

[36] R. E. Boyatzis, *Transforming qualitative information: Thematic analysis and code development*. sage, 1998.

[37] N. Gibson and H. O’Connor, “A step-by-step guide to qualitative data analysis,” *A journal of aboriginal and indigenous community health*, vol. 1, no. 1.

Samir Obaid Samir Obaid is lecturer in computer science department at Capital University of Science and Technology, Islamabad. Previously, he was working as software engineer at software development organization and his responsibilities were: test planning, test design and automation. He did MS in computer science from ARID university Rawalpindi, Pakistan. His research interests are model driven engineering, model-based testing, and software reengineering.

Ibrar Arshad Ibrar Arshad is currently associated with the Department of Computer Science at Capital University of Science and Technology, Pakistan. He received his Master's degree in 2012 from Blekinge Institute of Technology, Sweden. His area of research is software requirements, software engineering practices in small and medium-scale organizations.



Muhammad Usman Abid Muhammad Usman Abid did MS in software engineering from Riphah International University Islamabad,Pakistan. Currently, he is working as a lead software engineer at reputed software development organization in Pakistan. His research interests are information systems development, empirical software engineering and software reengineering.