International Science and Technology Journal
المجلة الدولية للعلوم والتقنية
Volume 39 العدد
Part 1 المجلد
المجلة الدولية للعلوم والتقنية
International Science and Technology Journal
ISTJ

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

Received	**2026/07/23**	**تم استلام الورقة العلمية في**
Accepted	**2026/08/17**	**تم قبول الورقة العلمية في**
Published	**2026/08/18**	**تم نشر الورقة العلمية في**

Documenting and Classifying Reusable Software Components: A Case Study in Libya

Mohamed Ali Hagal[1] and Esra Fathi Elkhraaz[2]
Department of Software Engineering, Faculty of Information Technology, University of Benghazi, Benghazi, Libya
Mohamed.hagal@uob.edu.ly, Asraa.fathi@uob.edu.ly
1- https://orcid.org/0009-0001-3011-0062, 2- https://orcid.org/0009-0004-9182-7229

Abstract
Reusing existing software components is usually cheaper than writing new code, but developers can only reuse what they can find and trust. It is often finding a suitable component that is the real difficulty, not reuse itself. This paper describes a method for documenting and classifying reusable software components so that they are easier to search, compare and reuse. In this method, each component is first described using a short three-part form (metadata, functionality, interface) and this information is then used directly to classify the component along five fields: operating system, programming language, platform, role and purpose, which let a developer filter a repository down to the components that actually fit their needs. Then, this method is illustrated with a case study which involves choosing a component for a hospital patient-record system. In this illustration, three candidate components matched the basic requirements, and the documented functionality information was enough to identify the one component with the needed medical-history feature without downloading or testing any of them first. We also surveyed a wider group of Libyan developers who watched a recorded demonstration of the tool. Most of them reported a positive perception of its effectiveness and ease of use. Future work should test this method with more developers on more tasks and compare it against existing ways of searching for components.

Keywords: software reuse, component documentation, component classification, software repository, case study.

International Science and Technology Journal
المجلة الدولية للعلوم والتقنية
Volume 39 العدد
Part 1 المجلد
المجلة الدولية للعلوم والتقنية
International Science and Technology Journal
ISTJ

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

توثيق وتصنيف مكونات البرمجيات القابلة لإعادة الاستخدام: دراسة حالة في ليبيا

محمد علي حجل، إسراء فتحي الخراز
قسم هندسة البرمجيات، كلية تقنية المعلومات، جامعة بنغازي، بنغازي، ليبيا
Mohamed.hagal@uob.edu.ly, Asraa.fathi@uob.edu.ly

الملخص

تُعد إعادة استخدام مكونات البرمجيات الجاهزة أقل تكلفة عادة من كتابة كود جديد، إلا أن المطورين لا يمكنهم إعادة استخدام سوى ما يستطيعون إيجاده والوثوق به، وغالبًا ما تكون صعوبة إيجاد المكوّن المناسب هي العائق الحقيقي وليس إعادة الاستخدام بحد ذاتها. تقترح هذه الورقة طريقة لتوثيق وتصنيف مكونات البرمجيات القابلة لإعادة الاستخدام لتسهيل البحث عنها ومقارنتها وإعادة استخدامها. يُوصَّف كل مكوّن أولًا عبر نموذج قصير من ثلاثة أجزاء (البيانات الوصفية، الوظيفة، الواجهة)، ثم تُستخدم هذه المعلومات مباشرة لتصنيف المكوّن ضمن خمسة حقول: نظام التشغيل، لغة البرمجة، المنصة، الدور، والغرض. تُوضّح الورقة هذه الطريقة من خلال دراسة حالة: اختيار مكوّن لنظام إدارة مستشفى، حيث طابقت ثلاثة مكونات مرشحة المتطلبات الأساسية، وكانت معلومات الوظيفة الموثقة كافية لتحديد المكوّن المناسب دون الحاجة لتنزيل أو اختبار أي منها. يوصى بتوسيع اختبار الطريقة على عدد أكبر من المطورين والمهام مستقبلًا.

الكلمات المفتاحية: إعادة استخدام البرامج، توثيق المكونات، تصنيف المكونات، مستودع البرامج

I. Introduction

A software component is an encapsulated, independently deployable unit of functionality that is accessible through a defined interface [1]. Since McIlroy first proposed the idea of component-based construction in 1968, the reason for reusing such units instead of writing new code has stayed the same: building a system from existing, working parts is normally cheaper and faster than writing every part again. However, the size of the problem has changed.

http://www.doi.org/10.62341/istj-vol39-1-zm34

Developers today do not lack suitable components, but if anything, there are too many of them spread across package managers, internal repositories and old codebases, most of which with unclear or inconsistent descriptions. Thus, the real question is no longer whether reusable components exist, but whether a developer can find them quickly enough and trust them enough to reuse rather than rewrite.

This is what makes reuse one of the most important problems in software engineering today. There is no doubt that software reuse can bring real gains in productivity, development speed and cost [2]. Yet, as other researchers point out, developers repeatedly struggle with the practical side of reuse' that is, knowing which components to use and how to find them without too much effort [3], [4]. This difficulty usually comes from poor documentation and weak classification of reusable components, and together these two problems reduce discoverability and add extra cost and time [5].

In other words, this difficulty depends on two things happening together. First, a component has to be documented well enough that someone other than its author can judge whether it fits a new project; i,e, which operating systems it runs on, what it actually does, how it fails, and how it is called. Second, once many components are documented, they must be organized so that a search narrows the field instead of returning everything that loosely matches a keyword. Petersen [6] makes essentially this point by claiming that choosing the right component is central to a project's success, yet the selection step is exactly where existing tools are weakest. Moreover, our reading of the literature covered in Section II is that these two needs are usually treated as separate research problems. That is to say, it is often the case that one study proposes a documentation template, while another proposes a classification method and the two are rarely combined, built and then tested with real developers on a real task.

This gap is not only visible in the literature, but it also comes up whenever we talk to developers about how they actually find components as could be deduced from the survey used in this study. In addition, a component can be technically reusable and still go unused, simply because no one can tell, in reasonable time, what it does or whether it will run in their environment. In the Libyan

http://www.doi.org/10.62341/istj-vol39-1-zm34

software development context specifically, to the best of our knowledge, no earlier study has measured how common this problem is or tested a documentation-and-classification tool with local developers even though the same challenges (i.e., informal documentation habits, and limited repository tools) are unlikely to be unique to one country. This motivated us to build a tool that treats documentation and classification as one connected pipeline, not two separate tasks and to test it directly with developers instead of leaving it as a theoretical idea.

This paper describes a method with two parts. First, a short documentation form (metadata, functionality, interface) that a developer fills in once for each component. Second, a classification scheme with five fields: operating system, programming language, platform, role, and purpose which is filled in automatically from the documentation and used to filter a repository of components. We built this method as a working tool and tested it by showing, through a case study, how it helps a developer choose between several similar components. Additionally, a video demonstrating it was utilized to collect the perceptions of a group of Libyan developers of it and its expected practicality, usefulness and impact.

Therefore, this paper is scoped to the method itself, a single illustrative case study and a demonstration-based perceptual survey completed by a wider group of Libyan developers. It is worth mentioning that sections IV and V report these two evaluation tracks separately and do not pool them into a single sample.

Research Questions

This paper is guided by three questions: RQ1: 2. What is the proposed approach to reduce the problem of poor documentation and classification of software components? RQ2: How is this approach implemented, illustrated step by step? RQ3: How is such an approach perceived by Libyan developers? Section III addresses RQ1 by describing the method; Section IV addresses RQ2 through a worked case study; Section V addresses RQ3 through a developer survey.

The rest of the paper is organized as follows. Section II reviews related documentation and classification research. Section III describes the method. Section IV presents a case study of the

http://www.doi.org/10.62341/istj-vol39-1-zm34

method in use. Section V reports a perceptual survey of the tool completed after a recorded demonstration. Section VI concludes and outlines future work.

II. Related Work

Software reuse is widely reported to reduce development cost and time while improving quality. Developers can draw on several sources of reusable assets, such as legacy systems, commercial off-the-shelf products, software libraries and purpose-built components [7]. Even so, Mahmood and Zayer [8] found that weak organizational knowledge and poorly maintained repositories remain persistent obstacles to reuse in practice, a finding which is echoed by later work on the absence of well-kept repositories [9], [10].

A second group of studies looks specifically at documentation. David [11] argues that good documentation preserves both the quality and the maintainability of a component. Venigalla and Chimalakonda [12] studied open-source projects and found that documentation which is missing, inconsistent or out of date slows contributors down. Additinally, Aghajani et al. [13], in a large study of 878 documentation artifacts found something more specific: roughly half of the documentation problems they found were not about wrong information, but about correct information that simply could not be located, which is a findability problem, not an accuracy problem. This distinction matters for this paper, because it is exactly the gap between documentation existing and documentation being easy to find that motivates treating documentation and classification as one pipeline rather than two separate concerns. Furthermore, Theunisen et al. [14] add that fast-moving development teams often rely on informal chat messages instead of structured documentation, which could work for the team at the time but leaves nothing durable for a future developer to search.

A third group of studies addresses classification directly, and this is where the present work differs most clearly from earlier proposals (see Table I). Zafar et al. [15] and Srinivas et al. [16] each propose ways to group or compare components by clustering by function, language and platform in one case, and by employing a similarity measure in the other, but neither study connects classification to any

http://www.doi.org/10.62341/istj-vol39-1-zm34

specific documentation input, and neither reports a working tool tested with real users. Similarly, Kannan and Ranjani [17] classify components using design metrics such as coupling and cohesion processed through a genetic-algorithm method; however, the technique in this proposal is more mathematically advanced than the approach used here, but that complexity also makes it harder for an ordinary developer to use and the study does not test the method on a real selection task. In the same sense, Korra [18] applies the Apriori association-rule algorithm which is originally built for retail 'basket analysis' to recommend components. This kind of rule is good at spotting items that are frequently used together, but is a poor fit for component dependencies, which are directional and often strictly incompatible rather than simply 'related.' Mittermeir and Pozewaunig [19] classify components by matching their input and output behavior, which could work well for simple, well-defined components but does not directly support the operating-system, platform, and role-based questions that a developer typically needs to filter on first.

Beyond these academic classification schemes, developers today most often search for reusable components through general-purpose package registries and code hosts, such as GitHub, npm, Maven Central, NuGet and PyPI. While these platforms are usually reported useful [21], their retrieval model differs from the one proposed here in a specific way: search in those repositories is driven by free-text keyword matching against a README, package description or commit history, and results are typically ranked by download counts, stars or textual relevance rather than by whether a component actually meets an environment or functional requirement. A keyword search for a "patient record component," for instance, surfaces anything mentioning those words, compatible or not, leaving the burden of checking operating-system, platform and functional fit to the developer reading through results one at a time. Nevertheless, the method in this paper differs by requiring operating-system and platform compatibility as mandatory, structured fields before a component is even returned, and by exposing role and purpose as separate filterable attributes rather than leaving them buried in free text. However, it is worth mentioning that a head-to-head empirical comparison against any of these

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

platforms has not been done since doing so would require the same components documented on both sides, which is beyond the scope of the single case study reported here; hence, this remains a conceptual contrast rather than a measured one, and should be read as such.

Table 1. Comparison of Prior Classification Approaches with The Proposed Approach

Study	Documentation model	Classification basis	Empirically validated	Main limitation
Zafar et al. [15]	None specified	Clustering by function, language, platform	No	Classification not linked to any documentation input
Srinivas et al. [16]	None specified	Similarity measure between components	No	No implementation or case study reported
Kannan & Ranjani [17]	None specified	Design metrics via genetic-algorithm method	No	High complexity for everyday practitioners
Korra [18]	None specified	Apriori association-rule mining	No	Rule model not suited to directional component dependencies
Mittermeir & Pozewaunig [19]	Behavioral (I/O) only	Input–output transformation matching	No	Weak fit for OS/platform/role-based filtering
This paper	Metadata, functionality, interface	5 attributes: OS, language, platform, role, purpose	Illustrated via a worked case study	Shown on one case study; not yet tested at scale

Taken together, these studies agree that both documentation and classification matter for reuse, but they also show a consistent gap:

Documenting and Classifying Reusable Software Components: A Case Study in Libya

http://www.doi.org/10.62341/istj-vol39-1-zm34

most classification methods are tested only in theory or on small toy examples, never on a real task performed by a real developer. Kalantari et al. [20] make the same observation about the field as a whole. The approach described in the next section is our direct response to that gap: documentation and classification are combined into one pipeline, built as a working tool and tested with actual developers.

III. Proposed Methodology

This study introduces an approach that solves three related problems at once: poor documentation, weak classification and slow retrieval of reusable software components.

The methodology has two layers. The theoretical layer decides two things: what information gets written down about a component, and which of that information is later used to classify it. The practical layer turns that plan into a working tool with three steps: Input, Processing and Output. In simple terms: a component is documented once, the tool classifies it automatically from that documentation, and it then becomes searchable inside the repository. Fig. 1 shows this whole idea in one picture where documentation feeds classification, and classification, in turn, feeds the repository.

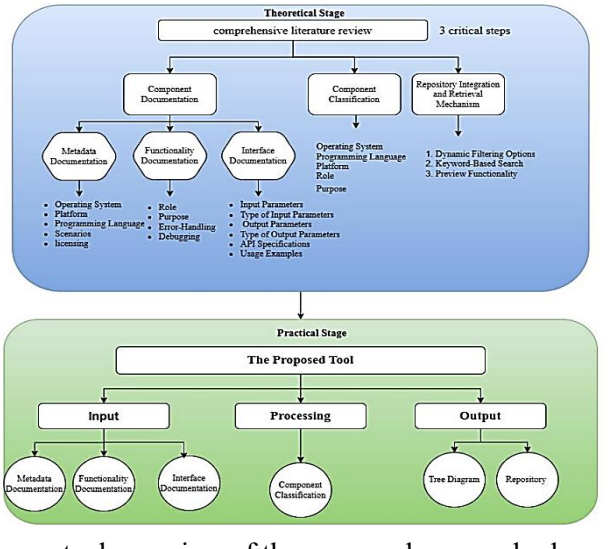

Fig. 1. Conceptual overview of the proposed approach: documentation feeds classification, which feeds repository retrieval.

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

A. Design Rationale

Two choices shape everything else in this design, so they are worth explaining before describing the steps themselves.

First, classification always comes after documentation and never before it, and never separately from it. A component cannot be labelled by operating system or by role until someone has already written down, in its documentation, what operating system it supports and what role it plays. This means a component simply cannot enter the system without being documented first, which closes the exact gap Aghajani et al. [13] describe between information existing and information being findable.

Second, we chose five classification attributes: operating system, programming language, platform, role, and purpose. We picked these five because they are the same things developers already think about when judging whether a component is useful to them, and because they each appear individually in earlier studies [15], [19] but no earlier study put all five together in one searchable tool. We did not use clustering or machine-learning methods of the kind used in [15]–[18], because those methods compare components to each other, whereas what a developer actually needs is to compare components to their own requirements, a simpler and more direct problem.

B. Component Documentation Schema

Initially, every component is documented once using a fixed template. Table 2 shows this template, which is organized into three parts:

- Metadata: documenting the operating systems the component supports, the platform or framework it needs, the programming language it is written in, example situations where it is used and its licensing terms.
- Functionality: documenting the component's role, its purpose, how it handles errors and how it can be debugged.
- Interface : documenting its input parameters, its output parameters, an example of how to use it and any API details needed to connect it to other software.

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

Fig. 2 shows the actual form developers fill in for these three parts. The form uses short fields, not open paragraphs, so it stays quick to complete. Thus, matters because a documentation step that takes too long simply will not get used.

Table 2. Framework for Documenting Software Components

Document Metadata	Document Component Function	Document Interface
Operating System (OS)	Role	Input Parameters
Platform	Purpose	Type of Input
Programming Language	Error Handling	Output Parameters
Scenario	Debugging	Type of Output
Licensing		Usage Example
		API

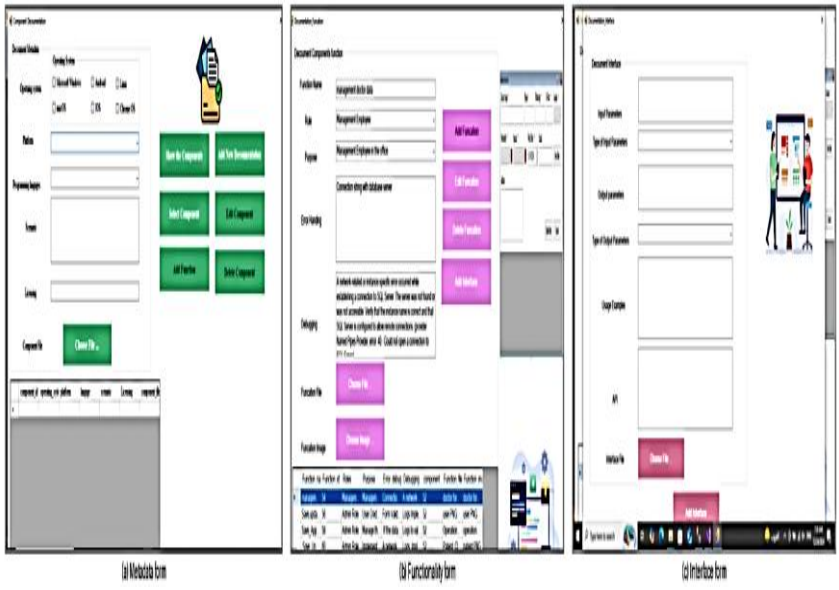

Fig. 2. Component documentation form used to capture metadata, functionality and interface information for a single component.

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

C. Component Classification Scheme

Once a component is documented, the tool classifies it automatically using the same information just entered and there shall be no extra typing or separate classification step. The five classification attributes are treated in two groups. Operating system and platform are mandatory; that is, a search cannot even run without them because a component that does not run on the right operating system or platform is useless, no matter how good it is otherwise. Programming language, role, and purpose are optional refinements which are applied once the mandatory filters have already ruled out anything incompatible.

Fig. 3 shows the Component Classification interface where a developer enters their own requirements (for example, "Windows" and ".NET") and the tool matches these against every documented component's attributes. This order which removes anything incompatible first, then narrows down by function is what keeps the search fast, and it is exactly what the case study in Section IV walks through step by step. In this study, the word 'classification' is used to mean structured, multi-attribute matching derived directly from documentation fields, not a statistical or learned model. That is to say, a component either matches a given set of attributes or it does not; no similarity score or ranking is computed and matching components are returned as an unordered filtered list.

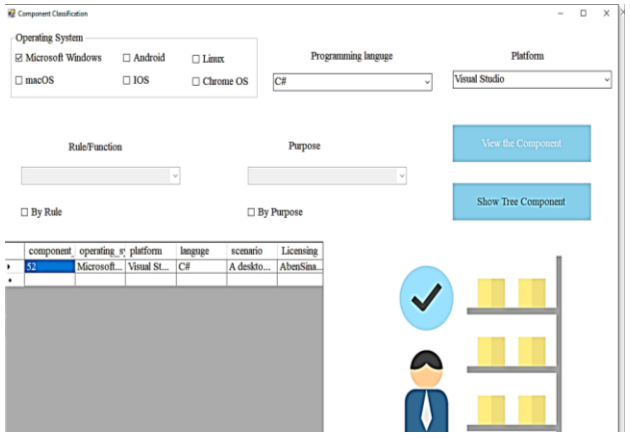

Fig. 3. Component Classification interface, where a developer enters mandatory and optional search attributes.

Documenting and Classifying Reusable Software Components: A Case Study in Libya

http://www.doi.org/10.62341/istj-vol39-1-zm34

D. Repository Architecture and Retrieval

Fig. 4 shows how the underlying data is structured: each component is linked to one metadata record, one functionality record and one interface record, and is separately tagged with its five classification attributes. Indeed, keeping the classification tags separate from the full documentation text is what lets the tool filter instantly instead of having to re-read every component's full description each time someone searches.

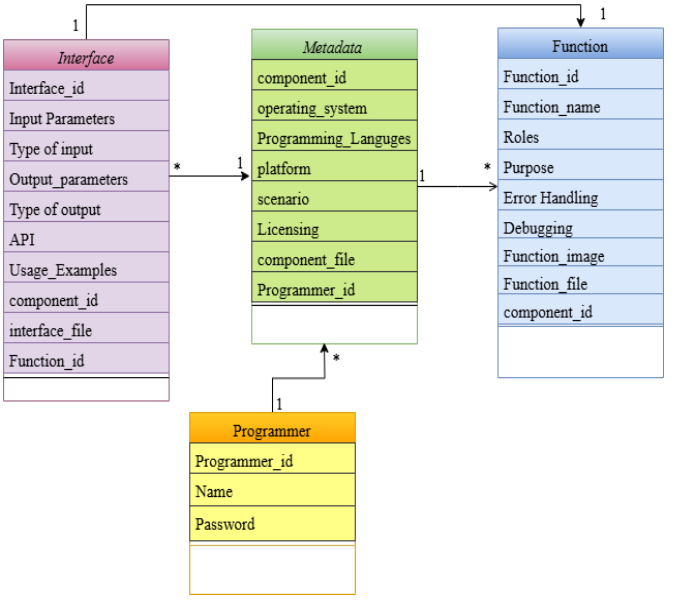

Fig. 4. Class diagram of the repository schema linking component documentation to classification attributes.

All documented and classified components are stored in one place and accessed through the Repository Form, shown in Fig. 5. This repository is a single, easy-to-use screen where a developer can search, compare and choose the component that best fits their project. Selecting a component from the results opens a tree-style preview showing its metadata, purpose, and debugging notes, so a developer can judge whether it fits before downloading or importing anything. In fact, this preview step is what the case study in Section IV relies on to compare three candidate components side by side.

Documenting and Classifying Reusable Software Components: A Case Study in Libya

http://www.doi.org/10.62341/istj-vol39-1-zm34

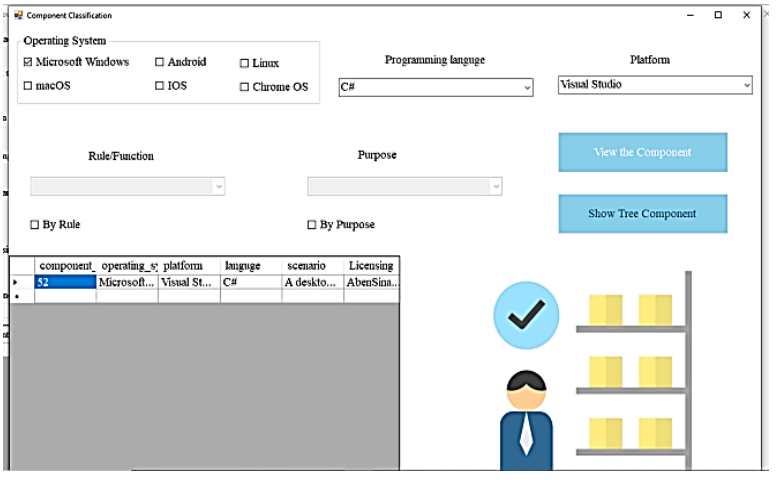

Fig. 5. Repository Form showing the filtered list of matching components.

E. System Implementation

The tool runs in three steps that match the theoretical layer described above.

- Input: this is where a logged-in user fills in the metadata, functionality, and interface forms for one component (Section III-B). Indeed, requiring login is not about security for its own sake; it is so that every entry can be traced to the person who wrote it and so the shared repository cannot be edited by just anyone.
- Processing: this is where the tool reads the values just entered and maps them directly onto the five classification attributes (Section III-C). There is no separate manual tagging step because classification is simply an automatic read of what the documenter already wrote.
- Output: here, the tool runs the developer's search and returns the filtered repository table plus the tree-diagram preview (Section III-D).

It is important to add that because classification is generated automatically from documentation, the only place a mistake can enter the system is at the documentation step itself, which is also the easiest place to catch and fix it. Section IV illustrates this tool with

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

a case study: a hospital patient-record component search, walked through step by step from search to final choice.

In short, this approach gives developers a practical way to document, classify and retrieve reusable components, and the case study in Section IV shows this working on a real task rather than only in theory.

IV. Case Study: Component Integration into a Hospital Management System

This is an illustrative case study and not a controlled experiment. It demonstrates how the method works on one real task, rather than measuring performance against an alternative. This section walks through a full example of the method in use where a hospital administrator searching for a component to extend an existing patient-management system. It is reported in detail because it exercises every part of the method described in Section III, involving mandatory filtering, optional refinement, and tree-diagram comparison, on a task with a clearly right and wrong answer.

Developers at a tech company in Benghazi called “Tahdith Wa Altahawal Alrqami” were seeked to test this tool on their project “ Spitar” over two weeks. Participation in this case study was voluntary and a written consent covering participation and data protection was obtained from the organization before the trial began. Unfortunately, no separate ethics-committee review was sought for this exploratory case, which we note as a limitation in Section VI.

A. Scenario and Requirements

An administrator of the organization where the tool was tested needed a component to record patient diagnosis data so that doctors could track cases over time. Three requirements were fixed in advance: the component had to run on Microsoft Windows, it had to integrate with Visual Studio, and, functionally, it needed both a section for entering personal patient information and a dedicated section for medical history since a component that only stored demographic data would not meet the real clinical need. These three requirements map directly onto the classification scheme in Section III-C: operating system and platform are the two mandatory filters

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

and stores medical history, not just demographics which is a purpose-level detail that only becomes visible once a component's documented functionality record is inspected.

B. Search Process

The administrator logged into the tool, opened the component classification window and entered Windows and Visual Studio as the mandatory filters (Fig. 6). This single step is exactly what the mandatory-filter design in Section III-C is meant to achieve. In other words, instead of browsing a repository in random order, the search space was reduced immediately to only the components guaranteed to run in the target environment before any functional comparison began.

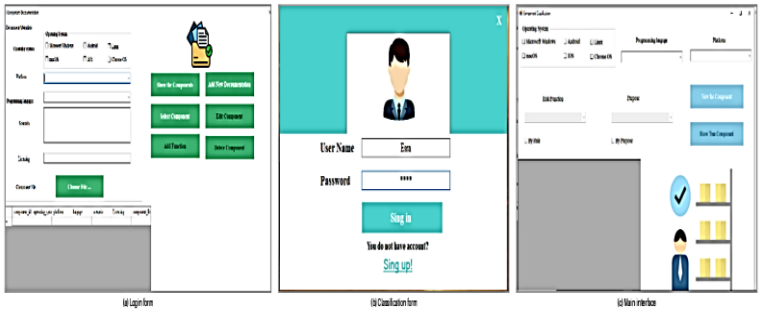

Fig. 6. Case study search process: (a) authentication, (b) mandatory-filter classification form (Windows, Visual Studio), (c) resulting repository view.

C. Comparative Analysis of Retrieved Components

Next, the query returned three components, all compatible with Windows and Visual Studio, which the administrator then compared using the tree-diagram preview described in Section III-D (Fig. 7). Component 1 recorded general patient information but had no dedicated field for ongoing medical history. Component 3 supported patient registration but also offered only limited history tracking. Component 2 was the only one of the three with a dedicated section for recording diagnosis data and tracking medical history over time, which presents the exact purpose-level requirement identified in Section IV-A. Hence, because the three components were otherwise

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

identical on the mandatory environmental filters, the deciding factor was entirely a purpose-attribute difference, visible only in each component's documented functionality record, something that would not show up from a filename, popularity or platform tag alone.

Fig. 7. Tree-diagram comparison of the three retrieved components where Component 2 was selected for its dedicated medical-history support.

D. Outcome and Interpretation

The administrator selected Component 2 without downloading or test-integrating any of the three candidates since the preview gave enough functional detail to rule out Components 1 and 3 on inspection. Two points about this outcome are worth stating. First, the mandatory-filter step did the coarse work (removing anything incompatible with Windows/Visual Studio) and the tree-diagram preview did the fine work (telling three compatible components apart on a purpose-level detail); neither step alone would have been enough, which is the practical reason for combining documentation-derived classification with a comparison view rather than relying on either alone. Second, this is one trial, not a controlled experiment and the current study did not time an equivalent search performed

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

without the tool, so this case study shows the mechanism by which the method helps, not a measured effect size.

In a follow-up interview of the experiment, the organization developers described specifically what stood out for them, that is, the mandatory filters ruled out incompatible components immediately and the tree-diagram preview let them see the medical-history difference between the three candidates without installing any of them, which they said would normally have taken far longer to discover by trial and error. Overall, they have expressed a favourable attitude towards the proposed tool highlighting its ability to work on different operating systems and its ease of installation. In their feedback; however, they suggested developing it even further to apply to larger components. A letter recommending the tool by the testing company can be accessed in appendix B.

V. Perceptual Survey of the Demonstrated Tool

Beyond the case study in Section IV, the tool was evaluated more broadly through a structured online survey built and distributed using Google Forms. The survey link was shared alongside a recorded video demonstration of the tool on social media, reaching software developers and computing students in the Libyan community where 89 responses were received before the form was closed. This track is entirely separate from the hands-on trial in Section IV: respondents here watched a demonstration rather than using the tool themselves.

A. Survey Instrument

The survey contained 25 items developed by the researchers specifically for this study rather than adapted from a previously published one. The questions were reviewed by two university instructors at the faculty of IT, University of Benghazi before distribution for checking content validity and giving their feedback about the clarity and comprehensibility of the items. They suggested some changes which the authors discussed and applied to the questions before the final version is agreed on. The full item wording, translated from the original Arabic, is reproduced in Appendix A. Items 1-3 recorded respondent qualification, years of programming experience and gender using nominal or banded-

http://www.doi.org/10.62341/istj-vol39-1-zm34

range options. Items 4-14 asked about respondents' existing familiarity with, use of, and attitudes toward software reuse in general, independent of the proposed tool, using a mix of response formats rather than a single uniform scale: 3- to 4-point ordinal options for familiarity (Item 4) and frequency of use (Item 5); binary yes/no or yes/no/not-sure options for belief and habit questions (Items 7, 9, 10, 11); multiple-selection checkboxes for perceived benefits (Item 6) and challenges (Item 8); and one open-ended item (Item 13). Items 15-24 asked respondents to rate the tool after watching the demonstration video, mostly using 4- or 5-point ordinal rating scales on effectiveness, suitability, ease of use and expected impact; Item 15, for example, ranges from "very effective" to "not effective at all," and is worded explicitly around the viewing: "after watching the video attached with this survey, in your view, how effective is the proposed tool at improving efficiency and productivity...". Finally, item 25 was a closing open-ended question. Because response formats vary by item including binary, 3-point, and 4- or 5-point ordinal, this is described as a mixed-format questionnaire rather than a single Likert scale, and report results as frequencies and percentages per item rather than a composite score. Items 6 and 8 allowed multiple selections, which is why their percentages do not sum to 100%.

B. Participant Profile

Most respondents (59.6%) held a bachelor's degree and had one to five years of programming experience (69.7%), and the sample was 59.6% female and 40.4% male (Items 1-3). We report this profile because it matters for interpretation: the sample skews toward early- and mid-career developers, so the results below should be read with that in mind rather than generalized to senior architects. The recruitment method which involved a public link shared on social media alongside the demonstration video means the sample is one of convenience rather than a probability sample, and no formal inclusion or exclusion criteria beyond self-selection were applied; this is a limitation which we acknowledge here.

C. Current Reuse Practices and Beliefs

Only 14.6% of respondents reported full familiarity with software reuse as a concept, 53.9% reported intermediate knowledge, and

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

23.6% reported little knowledge or none (Item 4). Correspondingly, 32.6% reported using ready-made components regularly and 58.4% occasionally (Item 5). Asked what benefits they associate with reuse (Item 6, multi-select; Fig. 8), respondents most often selected faster development (58.4%), improved quality (42.7%), reduced defects (36%), and increased efficiency (39.3%), whereas when asked about challenges (Item 8, multi-select), the most common were incompatibility with the current system (56.2%), poor documentation (22.5%), and maintenance difficulties (32.5%).

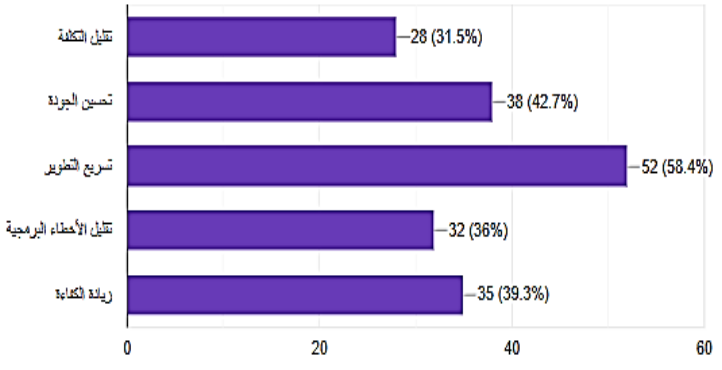

Fig. 8. Benefits of software reuse selected by survey respondents (Item 6)

Two results stand out together. First, 85.4% of respondents agreed that well-documented components are easier to reuse (Item 9) which is a direct, if self-reported, confirmation of this paper's starting premise. Second, and in tension with that figure, 44.9% said they still prefer writing new code over using ready-made components (Item 10). Read together, these two figures suggest that developers already believe documentation matters, but that belief has not fully translated into reuse behavior, which is consistent with the argument in Section I that the missing piece is a working way to find suitable components quickly, not awareness of the value of reuse.

D. Perceptions of the Demonstrated Tool

After watching the demonstration (Item 15; Fig. 9), 46.4% of respondents perceived the tool as effective and 34.5% as highly effective, for a combined 80.9% reporting positive perceived

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

effectiveness. Asked whether it offered appropriate solutions to reuse challenges (Item 16), 50.6% said it offered partial solutions and 37.3% considered it highly suitable. On ease of use relative to traditional tools (Item 17), 15.1% rated it very easy and 46.5% easy (61.6% combined), while 34.9% rated it moderately difficult.

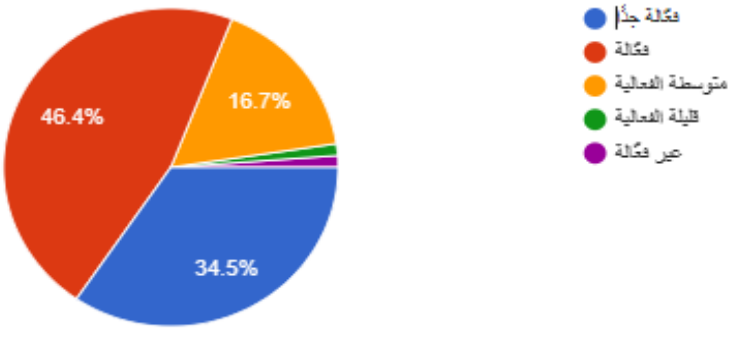

Fig. 9. Respondents' rating of the tool's perceived effectiveness after watching the demonstration (Item 15)

On expected impact, 43.5% expected a significant positive effect on software quality and 24.7% a very high effect (Item 18); 39.5% strongly believed the tool would reduce development time and 50% agreed it would (Item 19, 89.5% combined); 41.2% expected a significant improvement in team collaboration and 43.5% a moderate improvement (Item 21, 84.7% combined). On adoption intent (Item 20), 44.2% were highly certain they would use the tool in future projects and 43% said probably, leaving 9.3% neutral or negative. On perceived value relative to cost (Item 23), 51.8% rated it good value and 25.9% highly valuable. Once again, it should be noted that these are self-reported impressions formed after a =video demonstration, not measured outcomes from hands-on use and should be read as such throughout.

E. Limitations of This Survey Track

This track still has real limits, which we state directly rather than leave implicit. The sample is one of convenience, recruited through a public social-media link rather than a defined sampling frame, so response and non-response bias cannot be ruled out. The subgroup

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

comparisons in Table 5 are exploratory and underpowered for the smallest experience band (n = 6 - 7), so the absence of a significant difference there should not be read as strong evidence of no effect. Most importantly, every result in this section describes a reaction to watching a recorded demonstration, not to using the tool. Only the case study in Section IV is this paper's only evidence of hands-on use, and the two should not be conflated when reading the results together.

F. Item-Level Statistics, Reliability and Subgroup Analysis

Following access to the individual-level response file (n = 89), we report three analyses that could not be computed earlier from aggregate percentages alone. Response rates on Items 15-24 ranged from 83 to 86 out of 89 (93-97%); the remaining 3-6 respondents per item left that item blank, which we treat as missing rather than a negative response, and all summary statistics below use only non-missing cases (pairwise deletion) unless stated otherwise.

Table 4. Mean and Standard Deviation for Items 15-24

Item	Construct	n	Mean	SD	Scale
15	Perceived effectiveness	84	4.12	0.81	1-5
16	Suitability of solutions	85	3.25	0.69	1-4
17	Ease of use	86	3.56	0.64	1-4
18	Quality impact	85	3.88	0.84	1-5
19	Time Reduction	86	3.27	0.71	1-4
20	Adoption intent	86	4.27	0.82	1-5
21	Collaboration	85	4.22	0.81	1-5
22	Sustainability contribution	84	3.94	0.84	1-5
23	Value vs. cost	85	4.01	0.75	1-5
24	Change in outlook	83	3.82	1.01	1-5

Mean scores cluster toward the positive end of each scale (e.g., 4.12/5 for perceived effectiveness, 4.27/5 for adoption intent),

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

consistent with the percentage breakdowns reported in Section V-D. Item 24 (change in outlook) has both the lowest mean and the highest standard deviation (3.82, SD=1.01) which indicates more disagreement among respondents about whether the tool would change their approach than about its effectiveness or ease of use specifically.

Internal consistency across the ten tool-evaluation items (15-24) was high: Cronbach's alpha = 0.889 (n=79 after list-wise deletion of incomplete cases) which indicates that these items reliably measure a common underlying construct; that is, overall favorability toward the tool rather than ten unrelated judgments. This is well above the conventional 0.70 threshold for acceptable reliability.

We tested whether perceived effectiveness (Item 15), adoption intent (Item 20), and perceived effect on collaboration (Item 21) varied by respondent experience (three bands: 1-5, 5-10, 10-20 years), qualification (grouped into Bachelor's, Postgraduate, and Diploma), and, for Item 15, gender. One-way ANOVA (and an independent-samples t-test for gender) found no statistically significant differences on any comparison (all $p > .10$; Table 5).

Table 5. Subgroup Comparisons (One-Way Anova / T-Test)

Outcome item	Grouping variable	Test statistic
Item 15 (perceived effectiveness)	Experience	F=1.82, p=.169
Item 15 (perceived effectiveness)	Qualification	F=0.95, p=.390
Item 15 (perceived effectiveness)	Gender	t=-0.53, p=.596
Item 20 (adoption intent)	Experience	F=1.58, p=.213
Item 20 (adoption intent)	Qualification	F=0.06, p=.947
Item 21 (collaboration)	Experience	F=2.28, p=.108

Given that the 10-20-year experience band contains only 6-7

Documenting and Classifying Reusable Software Components:
A Case Study in Libya
http://www.doi.org/10.62341/istj-vol39-1-zm34

respondents, these null results should be read as suggestive of broadly consistent perceptions across subgroups rather than as a well-powered test of subgroup differences.**G. Reproducibility Protocol**

This subsection describes the steps taken for this survey track in enough detail that another researcher could repeat the same procedure and compare their results against ours, even without access to our raw data file.

- Participants: any software developer or computing student reachable through the public social-media link was eligible; no formal inclusion or exclusion criteria beyond self-selection were applied, and no target sample size was fixed in advance. Data collection stopped when the form was closed, yielding n=89.
- Data collection: the questionnaire was built in Google Forms and distributed as a public link alongside a recorded video demonstration of the tool, shared on social media. Responses were collected over several days in early January 2025 and recorded automatically by Google Forms with a timestamp per submission; no interviews or moderated sessions were involved in this track.
- Instrument: the 25-item questionnaire reproduced in full in Appendix A were used without revision after the review described in Section V-A. Items 1-14 used single-choice nominal or ordinal scales (2 to 4 response categories, described in Section V-A); Items 15-24 used single-choice ordinal scales of 4 or 5 response categories; Items 6 and 8 allowed multiple selections; Items 13 and 25 were open-ended and are not analyzed statistically in this paper.
- Analysis: Items 1-14 and Items 6/8 (multi-select) are reported as frequencies and percentages, with multi-select items reported as the percentage of respondents selecting each option (so percentages do not sum to 100%). Items 15-24 were additionally recoded to numeric scales (1 = least positive response, 4 or 5 = most positive, matching each item's response set) to compute the mean and standard deviation reported in Table 4. Cronbach's alpha was computed on these ten recoded items as k/(k-1)

http://www.doi.org/10.62341/istj-vol39-1-zm34

multiplied by one minus the ratio of the sum of item variances to the variance of summed scores, using pairwise-complete item data and listwise-complete cases for the alpha calculation itself. Subgroup comparisons (Table 5) used one-way ANOVA for the three-level experience variable and the three-level qualification grouping (Bachelor's, Postgraduate, Diploma/other collapsed from six original categories), and an independent-samples t-test for the two-level gender variable, all at a two-tailed significance threshold of .05.

- Reproduction steps: (1) recode each of Items 15-24 to its numeric scale as described above; (2) compute per-item mean and standard deviation on non-missing responses; (3) compute Cronbach's alpha on the ten recoded items; (4) group respondents by Item 2 (experience) and by Item 1 (qualification, collapsed as above) and run one-way ANOVA against each outcome item, and a t-test by Item 3 (gender) for Item 15. We used Python (pandas for data handling, scipy.stats for ANOVA and the t-test) to perform these steps; any statistical package that supports the same tests (e.g., SPSS, R) should reproduce equivalent results from the same recoding scheme.

VI. Conclusions

This paper described a method for documenting and classifying reusable software components which involved a short three-part documentation form (metadata, functionality, interface), used directly to fill in five classification fields (operating system, programming language, platform, role, purpose) that a developer can filter on when searching a repository.

The case study in Section IV shows this method working on a real task where a hospital administrator was able to go from three components that all matched the basic environment requirements to one correct choice using only the documented information without downloading or testing any of them first. This shows that the method is workable and easy to follow in at least one realistic case. The paper also tried to collect the perceptions of a wide group of Libyan developers of this tool by collecting their feedback using a survey after watching a video demonstrating its working. The data collected

http://www.doi.org/10.62341/istj-vol39-1-zm34

included information on how they perceived reuse, documentation and classifications of reusable components in general alongside their thoughts of the expected impact of the proposed tool. This paper has clear limits, which we state directly. The method is demonstrated on a single case study, one search task, and has not been tested across many components or many kinds of search; it is also not compared against how developers search for components today, so we cannot claim it is faster or better in measured terms, and we did not measure how long the same search would take without the tool.

Future work should test this method with more developers on more tasks, compare it against existing ways of searching for components and extend the documentation and classification fields to cover whole application modules rather than single components. Additionally, a broader survey-based evaluation covering participant recruitment, demographics, current reuse practices and perceived-effectiveness ratings across a larger group of developers shall be collected from a larger grouper of developers using the tool themselves rather than watching a video of its workings.

References

[1] J. H. Gerard, "Software components," IEEE Software, 2018.

[2] K. Srinivas, "Benefits and challenges of software reuse," 2011.

[3] Jalender et al., "Practical challenges in software component reuse," 2011.

[4] M. Basit et al., "Developer difficulties in locating and integrating reusable components," 2023.

[5] R. Straughn, "The cost of poor documentation and classification in software reuse," 2023.

[6] K. Petersen, "On the selection of software components for reuse: challenges and considerations," 2022.

[7] A. B. Al-Badareen et al., "Software reuse: A review and future directions," J. Software Eng. Appl., 2021.

[8] S. Mahmood and N. Zayer, "Organizational challenges to effective software reuse," 2014.

[9] T. Muhammad et al., "Repository-based approaches to software component reuse," 2014.

[10] K. R. Chythanya et al., "Repository management for reusable software components," 2022.

[11] R. David, "The role of documentation in software maintenance," 2020.

[12] A. S. M. Venigalla and S. Chimalakonda, "Understanding documentation practices in open-source software projects," 2021.

[13] E. Aghajani, C. Nagy, O. L. Vega-Marquez, L. Linares-Vasquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in Proc. 41st Int. Conf. Software Eng. (ICSE), 2019.

[14] T. Theunisen et al., "Documentation practices in continuous software development environments," 2022.

[15] S. Zafar et al., "A clustering approach to classify reusable software components," 2015.

[16] K. Srinivas et al., "Similarity-based classification of reusable software components," 2018.

[17] S. Kannan and R. Ranjani, "A genetic algorithm-based approach for classification of reusable software components," 2012.

[18] S. Korra, "Application of the Apriori algorithm in component classification and recommendation," 2022.

[19] R. Mittermeir and H. Pozewaunig, "Classification and retrieval of reusable components using characteristic input-output transformations," 2006.

[20] F. Kalantari et al., "Frameworks for reusable component classification: a review," 2021.

[21] J. Zeng, D. Han, Y. Zhu, Y. Wang, and F. Weng, “A Survey of Third-Party Library Security Research in Application Software,” Apr. 2024, doi: 10.48550/arxiv.2404.17955.