Investigating Developer Experience in Software Reuse

Rodrigo Feitosa Gonçalves
rfeitosa@cos.ufrj.br
PESC - UFRJ
Rio de Janeiro, Brazil

Cláudia Maria Lima Werner
werner@cos.ufrj.br
PESC - UFRJ
Rio de Janeiro, Brazil

Claudio Miceli de Farias
cmicelifarias@cos.ufrj.br
PESC - UFRJ
Rio de Janeiro, Brazil

ABSTRACT

Software reuse has been recognized as a key strategy for improving productivity, reducing development costs, and enhancing software quality. However, successfully implementing software reuse practices largely depends on the developer experience (DX). This study investigates the factors, barriers, and strategies influencing DX in software reuse. Through a Rapid Review (RR), we analyzed 328 studies, selecting 10 for detailed data extraction based on defined filters and the backward snowballing technique. Our findings identify 15 factors affecting DX in software reuse, categorized into technical, organizational, and human/social factors. We also uncover 7 barriers that impede developers from improving DX and identify 13 strategies to enhance it. The results highlight the critical role of comprehensive documentation, a clear understanding of software functionality, and robust reuse-compatible infrastructure as key technical factors. Organizational support, effective resource allocation, and fostering a communication, collaboration, and self-efficacy culture are essential for successful software reuse. This study's insights have significant implications for researchers and practitioners, offering practical guidance to develop more effective reuse practices and improve DX.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**.

KEYWORDS

Developer Experience, Software Reuse, Rapid Review

1 INTRODUCTION

Software reuse is an essential practice in software engineering, widely adopted to increase efficiency and reduce development costs [4]. By utilizing existing components, developers can avoid redundant work and focus on innovative aspects of projects [30]. Adopting reusable libraries, frameworks, and design patterns has become crucial for the competitiveness and sustainability of modern software organizations [42].

Although the theoretical benefits of software reuse are well established, practical implementation faces significant challenges, particularly related to the developer experience [47]. Difficulties such as the complexity of integrating reusable components [5], the lack of adequate documentation [3], the challenge of finding and evaluating suitable components, the need for ongoing maintenance and updates of reused software, and the potential mismatch between reused components and specific project requirements all negatively impact developer productivity and satisfaction [10, 47]. These challenges are essential for improving reuse practices and maximizing their benefits.

Enhancing developer satisfaction and motivation is critical for numerous organizations, given its potential to enhance productivity and bolster employee retention [12]. Recent studies have concentrated on comprehending the factors influencing developer productivity and satisfaction, encompassing aspects associated with software reuse [19, 36, 44]. The influence of these factors on developers' productivity and satisfaction varies across individual, team, organizational, and project-specific contexts [14].

Developer experience (DX), articulated by Fagerholm and Munch [12], is a broader concept perspective encompassing developers' sentiments, thoughts, and perceptions regarding their work. This definition suggests that DX is influenced by various factors such as team culture, work environment, and daily tasks. Similar to satisfaction, DX is also deeply personal [12, 14].

There is a gap in the literature and industrial practices regarding the real experience of developers in software reuse. Many approaches focus on technical aspects, neglecting the perspective of developers who often face psychological obstacles [30, 46]. Furthermore, over the years, the growing number of publications underscores that software reuse continues to be a subject of considerable interest in software engineering [30, 46]. In this context, the practical question to be addressed in this study is: **How is developer experience characterized in software reuse?** Identifying factors that influence DX - such as lack of documentation, organizational culture, and developers' professional experience with software reuse - can assist professionals in the software industry in enhancing their activities in software reuse environments. A Rapid Review (RR) was conducted to answer this study's practical question. RR studies are secondary research endeavors to uncover evidence to help solve practical problems [7, 24].

As a result, we identified 15 factors that affect DX in software reuse. These factors were classified into technical, organizational, and human/social categories. Additionally, we identified 7 barriers that hinder developers from improving the factors affecting DX in software reuse. We also identified 13 strategies to improve DX in software reuse. In this regard, the findings of these studies can raise awareness about the influence of these factors on software reuse, aiding professionals and researchers in developing more effective practices to address the impact of these factors in this context.

The remainder of this paper is organized as follows: Section 2 presents background; Section 3 presents related work; Section 4 depicts the method and protocol; Section 5 shows the results of the study; a discussion on the results and contributions of the study are presented in Section 6; finally, Section 7 presents the threats to validity, and Section 8 concludes the paper with final remarks and future work.

2 BACKGROUND

2.1 Software Reuse

Over the years, various definitions have been proposed for software reuse. For this article, we will adopt the definition provided by the IEEE Standard for Information Technology Reuse Processes and Software Life Cycle Processes [23]: *"Software reuse entails capitalizing on existing software and systems to create new products"*.

Under this broad definition, several terms coexist, including Component-Based Development (CBD), Software Product Lines (SPL), Model-Driven Development (MDD), Domain Engineering (or Domain Analysis), and Commercial Off-The-Shelf (COTS) solutions, among others [52]. Additionally, there are two primary development approaches: by-reuse (utilizing pre-existing components in development) and for-reuse (creating components intended for reuse) [32]. Reuse can also be classified based on its application scope as vertical reuse (reusing software within a specific application domain) or horizontal reuse (reusing components across multiple application domains) [15].

2.2 Developer Experience

The concept of experience can be viewed through various lenses in software engineering [12, 51]. These different variations of experiences are intertwined, but it is crucial to differentiate them to prevent misunderstandings. Various experiences associated with consuming, using, or interacting with something provided by others include user experience (UX), product experience, brand experience, and service experience. Thus, the definition of DX is shaped by the concept of UX. According to ISO 9241-210:2019 [48], UX is defined as "a person's perceptions and responses that result from the use or anticipated use of a product, system or service". Experiences specific to development include programmer experience and, finally, DX. The term "developer" refers to anyone involved in software development, while "experience" pertains to engagement rather than solely to expertise, although the two are interconnected [12]. So, we will adopt this definition of DX for this article.

Fagerholm and Münch [12] approach DX from a psychological perspective, dividing it into three distinct sub-areas or categories: cognitive (how developers perceive the development infrastructure), affective (how developers feel about their work), and conative (how developers perceive the value of their contribution).

3 RELATED WORK

As research on DX is a burgeoning field, relatively few secondary studies are available, particularly on software reuse. Our investigation has identified some literature studies exploring DX or related concepts. Fontao et al. [13] present preliminary results from a literature study in which they applied a systematic review methodology called snowballing and thematic analysis to explore factors influencing DX in Mobile Software Ecosystems (MSECO). They analyzed 11 papers and identified 20 factors associated with DX. They argue that understanding these factors in the ecosystem context is crucial for keystones that require mechanisms to ensure developers remain accountable for productivity, robustness, and niche creation.

Morales et al. [34] conducted a systematic literature review (SLR) of articles published in the last decade to identify definitions of Programmer eXperience (PX) or DX. As a result, the authors analyzed 73 articles, which highlighted the following findings: 1) The essential elements influencing PX are programmer motivation and the choice of tools they use in their work; 2) The majority of identified studies (59%) aimed to evaluate the influence of PX, UX, and usability in programming environments; 3) Most studies (70%) employed methods such as usability testing and heuristic evaluation methods; and 4) Four sets of heuristics are used to evaluate software development artifacts about programming environments, programming languages, and application programming interfaces. Unlike the previously mentioned studies, this work investigates the factors that influence DX in software reuse.

4 RESEARCH METHOD

We performed an RR during May and June of 2024 to conduct this study. RR studies are secondary research efforts primarily aimed at providing evidence to support decision-making and address problems faced by professionals in practice [7, 17]. To characterize them, an RR is a fast-tracked approach to conduct a systematic review of the scientific literature [18].

To conduct this RR, we followed the protocol model proposed by Cartaxo et al. [7]. Additionally, we considered the guidelines for conducting SLR proposed by Kitchenham and Charters [26]. The demand for an RR arose from a practical problem: technical factors, such as compatibility with existing platforms and code modularity, and social factors, such as effective team collaboration and cultural acceptance of software reuse practices, may influence DX in software reuse.

4.1 Definition of Research Questions

This RR aims to investigate factors that affect DX in software reuse. We formulated three research questions (RQ) to achieve the research objective. These RQ were defined closely with industry professionals and are presented in Table 1.

Table 1: Research Questions

ID	Research Questions	Focus
RQ1	What are the main factors that affect developer experience in software reuse?	This RQ aims to identify and describe the main factors that affect developer experience during software reuse. This includes understanding both technical and social factors.
RQ2	What barriers prevent developers from improving the factors affecting developer experience in software reuse?	This RQ aims to identify and analyze the main barriers that hinder the improvement of factors influencing developer experience in software reuse.
RQ3	What are the strategies used to improve developer experience in software reuse?	This RQ aims to identify and evaluate the current strategies to improve developer experience in software reuse.

4.2 Search Strategy

To conduct automated searches for primary studies that address the RQ defined in Table 1, we utilized the Scopus digital library[1], which consists of various relevant digital libraries [6]. Additionally, combining Scopus with snowballing procedures can mitigate the gap

[1] https://www.scopus.com/

from not using other digital libraries and provide a representative set of articles on the proposed topic [35].

A search string was defined according to the terms used in the RQ. We defined the search string, which involved crafting and testing different versions in the chosen digital library. The search string was based on terms previously used in studies related to software reuse and DX [4, 12, 14, 16]. Finally, the search string was reviewed by two experienced researchers in experimental software engineering. Table 2 presents the search string used in this study. Since our research pertains to an RR, there was no need to limit the scope to specific approaches, as our objective is to characterize DX in software reuse in general.

Table 2: Search String

Search String
(TITLE-ABS-KEY("software reuse" OR "software reusability" OR "product line" OR "component-based") AND (programmer OR developer AND (experience)))

4.3 Inclusion and Exclusion Criteria

To select studies, we defined and applied inclusion criteria (IC) and exclusion criteria (EC) to the retrieved studies (Table 3). Studies were included if they met all IC and excluded if they met at least one EC criterion.

Table 3: Selection Criteria

	ID	Description
Inclusion Criteria	IC1	The study presents evidence based on scientific experimental methods and experience reports (e.g., interviews, opinion surveys, case studies, etc.).
	IC2	The study answers at least one RQ.
Exclusion Criteria	EC1	The study does not meet at least one IC.
	EC2	The study is duplicated.
	EC3	The study is unavailable for free download or through institutional access.
	EC4	The study is not peer-reviewed or is a Master's or Doctoral thesis.
	EC5	The study is not written in English or Portuguese.
	EC6	The study brings the concept of DX as the time of experience or seniority.
	EC7	The study is not related to software reuse.

4.4 Study Selection Process

This study employed a seven-stage selection process: (1) Executing the automated search of primary studies (Search execution); (2) 1st Filter: Removal of duplicate studies (applying EC2); (3) Applying the 2nd Filter: reading title, abstract, and keywords; (4) Applying the 3rd Filter: reading introduction and conclusion; (5) Applying the 4th Filter: complete reading of the study; (6) Applying backward snowballing (BS) technique; and (7) Data extraction and synthesis.

The stages were based on the selection process defined by Cartaxo et al. [6], where each stage's output feeds into the next stage. Notably, BS was conducted by analyzing the references of the 6 studies included after stage 5 and was performed by the first researcher. In the first iteration, 196 studies were analyzed. After applying the criteria from Table 3 and implementing filters 1, 2, 3, and 4, only 3 studies were included. In the second iteration, 274 studies were analyzed (referenced in the study included in the first iteration of BS). However, no studies were included. Figure 1 provides a more detailed view of the process results.

4.5 Data Extraction and Synthesis

The first researcher conducted the extraction process, after which another, more experienced researcher verified the results. All data were systematically extracted using the Parsifal[2] tool to record the necessary information. A form containing the following fields was used to address the research questions: (1) study identifier (ID); (2) title; (3) authors; (4) year; (5) publication venue; (6) country; (7) factors affect DX in software reuse; (8) barriers preventing developers from improving these factors; and (9) strategies used to enhance DX.

To address the RQ, we utilized data synthesis. According to Kitchenham and Charters [27], two primary methods for data synthesis exist: (i) descriptive (qualitative) and (ii) quantitative. In this study, we employed the qualitative method to analyze the extracted data and answer the RQ, resulting in a descriptive data synthesis [11]. Initially, we adopted an open coding approach inspired by the initial procedure of the Grounded Theory approach. Thus, we systematically coded the data extracted from selected studies inductively (bottom-up) [8]. For this analysis, we thoroughly read the selected studies, dividing parts of the text into **coherent units** (sentences or paragraphs). Subsequently, we assigned **codes** representing key points relevant to the RQ. Following the open coding, we applied axial coding, following the Charmaz approach [8], to organize the codes into **categories**. Table 4 presents examples of the coding process for the selected studies (the resulting codes and their categories). The data extraction and synthesis results are detailed in Section 5.

5 RESULTS

The RR aimed to retrieve as many relevant studies as possible on the research topic. Therefore, no filtering based on publication year was applied, meaning all studies returned up until June 2024 were considered. The first researcher analyzed and selected the returned studies, while the second validated the selected studies. Figure 1 provides a more detailed view of the process results. Initially, 328 studies were returned. Following the selection process, 10 studies were included.

Table 5 presents a summary of data extraction from the selected studies in ascending order based on the year of publication. Each study's identification code (ID) references the respective study in the subsequent sections. Figure 2 illustrates the chronological progression of the included studies according to their year of publication. Most of these studies were published over 10 years ago, indicating that interest in and research on the topic have been more intense in past decades. This distribution reveals that most studies are dated, with a significant concentration of publications between 2003 and 2010. This fact highlights the need for new studies on this theme to update and expand existing knowledge, considering recent technological and social changes. Additionally, considering only the countries of the lead author of each study, it was found

[2]https://parsif.al/

Table 4: Illustration of the Coding Process

Coherent unit: *"Feedback and Evaluation Establishing feedback mechanisms and conducting regular evaluations of reuse practices can help identify areas for improvement, address challenges faced by developers, and refine reuse strategies to better align with organizational goals". E1*		
Preliminary Code	**Focused Code**	**Category**
Establishing feedback mechanisms	Feedback and Evaluation	Strategy

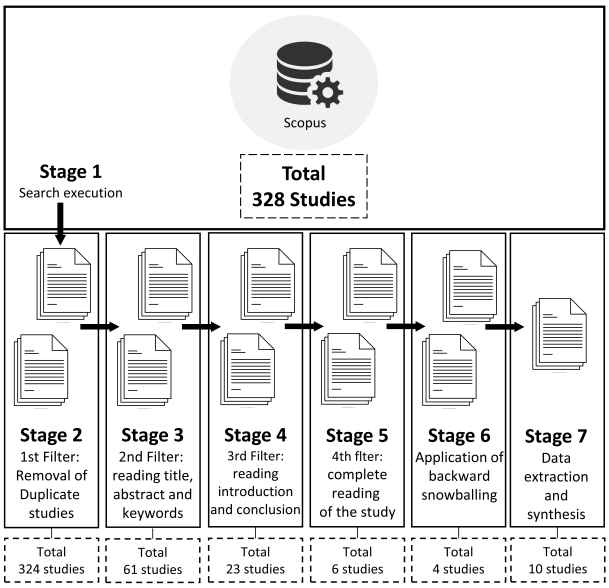

Figure 1: Results of the Study Selection Process

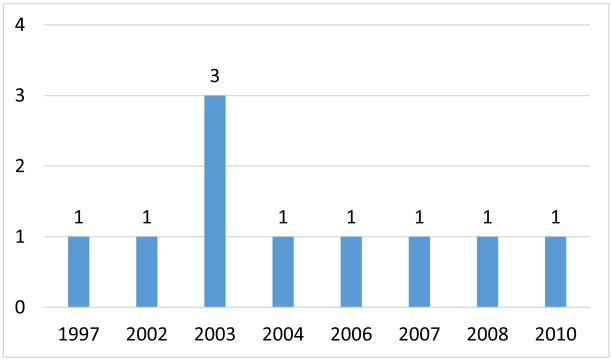

Figure 2: Number of Studies per Year

that the United States (3 studies), Norway (2 studies), and countries such as Germany, Qatar, South Korea, Scotland, Switzerland (1 study each) had publications on the subject (Figure 3).

Regarding the research method adopted by the studies, six studies (S1, S4, S5, S6, S7, and S9) conducted case studies in software development organizations, while four (S2, S3, S8, and S10) carried out field studies. The research techniques employed in these studies included surveys, used by nine studies (S1, S2, S3, S4, S6, S7, S8, S9, and S10), semi-structured interviews, applied in four studies (S2, S4, S9, and S10), and structured interviews, utilized in three studies (S5, S6, and S7). The raw data and all the steps necessary to

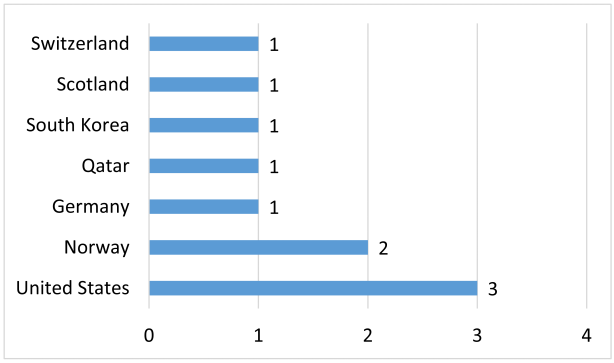

Figure 3: Number of Studies per Country

reproduce the research are detailed in the supplementary material openly available via ZENODO[3].

5.1 RQ1: What are the Main Factors that Affect Developer Experience in Software Reuse?

In the context of this research, we define "factors" as elements, characteristics, or actions that affect DX in software reuse. Thus, we identified 15 factors that affect DX in software reuse, classified into three main categories: technical, organizational, and human/social factors. Table 6 presents the identified factors. In the following sections, we present more details about each factor identified in the studies.

5.1.1 Technical Factors. **Documentation** (S4, S6, S7, and S10) is crucial for developers when reusing software components. It ensures flexibility and usability by providing detailed and up-to-date information. However, developers often find documentation incomplete or outdated, relying instead on informal sources like personal experience and local experts. Well-documented interfaces (APIs) simplify integration, reducing the need to fully understand the component's inner workings (S10).

Understanding Software Functionality, Interactions, and Architecture (S2, S6, and S9) significantly affects DX in software reuse. Developers struggle with deciphering framework behavior due to complex source code, abstract classes, and dispersed functionality S2. Understanding class interactions is also challenging, as they are not visible in the source code or documentation, complicating the prediction of modification impacts. Additionally, developers find it challenging to grasp architectural motivations and modify elements without compromising integrity, further hindered by insufficient and outdated documentation (S2, S6, and S9).

[3]https://doi.org/10.5281/zenodo.12685772

Table 5: List of Selected Studies

ID	Title	Author	Country	Origin	Year
S01	An Empirical Study of Software Reuse with Special Attention to Ada	Lee et al. [28]	South Korea	BS	1997
S02	Defining the Problems of Framework Reuse	Kirk and Wood [25]	Scotland	SCOPUS	2002
S03	Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices	Rothenberge et al. [40]	United States	BS	2003
S04	Barriers to adoption of software reuse A qualitative study	Sherif and Vinze [43]	Qatar	BS	2003
S05	Project-Level Reuse Factors: Drivers for Variation within Software Development Environments	Rothenberger [41]	United States	SCOPUS	2003
S06	A Study of Developer Attitude to Component Reuse in Three IT Companies	Li et al. [29]	Norway	SCOPUS	2004
S07	An Empirical Study of Developers Views on Software Reuse in Statoil ASA	Slyngstad et al. [45]	Norway	SCOPUS	2006
S08	A multi-level analysis of factors affecting software developers' intention to reuse software assets: An empirical investigation	Mellarkod et al. [31]	United States	SCOPUS	2007
S09	Code Reuse in Open Source Software	Haefliger et al. [49]	Switzerland	BS	2008
S10	Code reuse in open source software development: Quantitative evidence, drivers, and impediments	Sojer and Henkel [47]	Germany	SCOPUS	2010

Table 6: Factors Affecting Developer Experience in Software Reuse

Factors	Study ID
Technical	
Documentation	S4, S6, S7 and S10
Understanding Software Functionality, Interactions, and Architecture	S2, S6 and S9
Availability of Reuse-Compatible Infrastructure	S3 and S8
License	S9
Programming language	S9
Quality Attributes (Non-Functional Requirements)	S7
Organizational	
Project Attributes	S3 and S5
Client Influence	S5
Perceptions About Resource Allocations	S8
Project Culture	S5
Requirements (Re)negotiation	S6
Human/Social	
Developer Reuse Experience	S1, S4, S5, S6 and S8
Communication and Collaboration	S4
Community Knowledge	S10
Self-Efficacy	S8

Availability of Reuse-Compatible Infrastructure (S3 and S8). A formal, structured process for reuse helps standardize practices, making reuse efforts repeatable and predictable. The presence of methodologies, frameworks, and standardized processes is essential for the success of software reuse and influences developers' perception of the usefulness of software assets. A well-established reuse-compatible infrastructure enhances DX by providing clear guidelines and reducing uncertainty (S3 and S8).

License (S9). License issues significantly impact DX in software reuse. Legal issues arise when the licenses of potentially reusable code are incompatible with the developer's project, complicating or preventing reuse. This creates additional hurdles for developers, who must navigate and comply with various licensing requirements, affecting software reuse's ease and feasibility (S9).

The **Programming Language** (S9) used in a project significantly influences DX in software reuse. For example, certain languages may make it challenging to integrate popular libraries, limiting the ease of reuse. Project architectures may not be sufficiently modular to facilitate the seamless integration of reusable code. Moreover, compatibility issues between programming languages can further complicate the inclusion of code from different sources.

Quality Attributes (Non-Functional Requirements) (S7). Developers' confidence in the quality attributes of reusable components significantly impacts their software reuse experience. Uncertainty arises when developers are unsure whether components have been adequately tested to meet their quality requirements before integration. Improving the specification and publication of quality attributes and consistent component testing is crucial to enhancing developers' confidence and facilitating smoother integration of reusable components into projects.

5.1.2 Organizational Factors. **Project Attributes** (S3 and S5), such as commonality in requirements, design, and code (S3), significantly influence software reuse success and DX. They enable efficient component reuse, streamline integration within the same framework, leverage previous project solutions, and ensure compatibility with the project domain. Optimizing these attributes enhances developers' ability to effectively reuse components, improving overall software development efficiency and satisfaction (S3 and S5).

Client Influence (S5) factors that affect DX in software reuse include budget and time constraints, perceived value of reuse, and fear of interconnectivity. Limited budgets and tight deadlines restrict the implementation of reusable components. If clients do not see the benefits of reuse, they may not support it, leading to resistance in resource allocation. Concerns about cascading failures from reused components can discourage reuse, increasing the developer's workload and reducing potential benefits.

Perceptions About Resource Allocations (S8). Developers' perceptions regarding the appropriateness of resource allocations for software reuse play a crucial role in influencing their perceived usefulness of software assets. This factor is critical in shaping the overall DX in software reuse. In software reuse, perceptions of resource allocation encompass the degree to which an individual perceives that organizational resources are sufficient for developing and integrating reusable assets.

Project Culture (S5) reflects the different degrees of emphasis placed or incentives offered (tangible or intangible) to an organization. This analysis underscores the importance of a reuse-oriented project culture in enhancing developer engagement and optimizing the reuse process, ultimately leading to better software development outcomes.

Requirements (Re)negotiation (S6) is a critical but often inefficient aspect of software reuse that significantly affects DX. The challenges stem from a lack of access to source code, inadequate vendor support, and insufficient engineering expertise to modify integrated components. Even in-house reusable components often fail to meet all project requirements, necessitating (re)negotiation.

5.1.3 Human/Social Factors. **Developer Reuse Experience**. Experienced individuals, or reuse champions, are crucial for driving initiatives, providing training, and facilitating communication (S4

and S1). Key factors include recognizing reuse patterns, understanding the company's reuse model, and knowing component availability (S5). Understanding the reuse model is a long process, partially achievable through training and informal communication (S6). Thus, as developers gain experience, they recognize the value of software assets, finding them more accessible and valuable to adopt. Reuse-related experience positively influences the perceived usefulness and ease of reusing software assets, enhancing DX (S8).

Effective **Communication and Collaboration** (S4) between different teams and stakeholders are essential for successful software reuse. Encouraging information sharing and involving multiple teams in design reviews help overcome barriers. Improved communication and teamwork foster a more cohesive development environment, enhancing the overall DX and reuse initiatives.

Community Knowledge (S6). Developers often rely on their project's community, such as mailing lists, for direct and efficient information about reusable knowledge and code. This community knowledge is crucial for finding solutions, sharing best practices, and accessing reusable components. Active participation in these communities helps developers stay informed and supported, enhancing their ability to effectively reuse software assets and improving their overall experience.

Self-efficacy (S8). Developers indicate that their confidence in their skills for developing and reusing assets influences their perception of the usefulness and ease of implementing reuse technology. When developers believe they have the necessary skills, they report that reusable assets improve their job performance. Therefore, self-efficacy is positively related to perceptions of the usefulness and ease of reusing software assets.

5.2 RQ2: What Barriers Prevent Developers from Improving the Factors Affecting Developer Experience in Software Reuse?

In the paradigm of this study, "barriers" are obstacles that hinder or prevent an individual from taking action to achieve his/her goal. They can be physical, psychological, social, or economic. We identified 7 barriers that prevent developers from improving factors affecting DX in software reuse. Table 7 presents the identified barriers. We provide more details about each barrier identified in the studies.

Table 7: Barriers Preventing Improvements in Developer Experience

Barrier	Study ID
Lack of Top Management Support	S1, S3, S4, S5, S8 and S10
Insufficient Training and Education	S3, S4, S5 and S8
Resistance to Change	S1, S3, S4 and S8
Lack of Access to Reusable Components	S1 and S9
Perceived Lack of Benefits	S1
Ineffective Incentive Schemes	S8
Inefficiency in Reuse	S9

Lack of Top Management Support (S1, S3, S4, S5, S8 and S10). Limited resources, time constraints, and insufficient support hinder developers from enhancing reuse capabilities (S1 and S3). Without this support, reuse efforts are significantly hindered (S3). Management may hesitate to invest in reuse due to high initial costs and lack of immediate returns, compounded by the absence of measures to assess benefits and costs (S4). Some organizations report reuse success in a few projects, but many struggle to replicate this consistently. This inconsistency suggests that while an organization may have high reuse capability at the organizational level, project-level success factors can vary, affecting overall reuse success (S5 and S8).

Insufficient Training and Education (S3, S4, S5, and S8). Developers often lack the necessary training to effectively create and utilize reusable components, and inadequate mechanisms for knowledge sharing within organizations can prevent them from learning best practices (S3). Thus, training programs and education on reuse concepts are often insufficient, preventing developers from engaging in reuse activities effectively (S4). Acquiring a thorough understanding of reuse is a long process that cannot be fully achieved through training alone (S5). Furthermore, inadequate training on the benefits and processes of software reuse hinders developers from understanding its value and impact on their work (S8).

Resistance to Change (S1, S3, S4, and S8). In complex problem domains, developers may find identifying and extracting reusable components challenging, particularly when domain knowledge is limited or fragmented within the team (S1). An organizational culture resistant to change can impede reuse efforts, as developers and managers may be accustomed to their existing workflows and hesitant to adopt new practices (S3).

Developers may resist reuse due to the belief that it inhibits creativity, a phenomenon known as the "Not Invented Here" syndrome. This resistance is also driven by a lack of trust in the quality and performance of reusable components created by others (S4). Individual attitudes and beliefs further contribute to resistance towards software reuse, making it difficult for developers to embrace reusable assets (S8).

Lack of Access to Reusable Components (S1 and S9) is a barrier to improving DX in software reuse. Developers often need help finding sufficient high-quality reusable components within their organizations or repositories, hindering their ability to leverage reuse effectively (S1). Additionally, only some reusable resources are often available that meet the specific needs of a developer's project, making it challenging to find suitable code to reuse (S9). Enhancing access to a well-maintained and comprehensive repository of reusable components facilitates effective software reuse and improves DX.

Perceived Lack of Benefits (S1) is a barrier to improving DX in software reuse. If developers do not see immediate benefits, such as productivity gains or quality improvements, they may be less motivated to invest time and effort in enhancing their reuse practices. Without clear, tangible advantages, developers might prioritize other tasks over reuse initiatives.

Ineffective Incentive Schemes (S8). The lack of incentives or recognition for reuse efforts can demotivate developers from actively engaging in reuse practices. With proper incentives, developers may prioritize reuse activities. Implementing effective incentive schemes that reward reuse contributions can encourage developers to participate more actively in reuse practices, thereby improving their overall experience and the success of reuse initiatives.

Inefficiency in Reuse (S9). Some developers perceive that finding, understanding, and adapting reusable code can be more time-consuming than writing it from scratch. This perceived inefficiency

can deter them from engaging in reuse practices. To overcome this barrier, it is essential to streamline the process of locating and integrating reusable components, making it more efficient and user-friendly (S9). By addressing these inefficiencies, developers can be encouraged to adopt reuse practices, improving their overall productivity and experience.

5.3 RQ3: What are the Strategies Used to Improve Developer Experience in Software Reuse?

In the paradigm of this study, the term "strategies" refers to actions or interactions that can be employed to overcome, manage, or respond to the phenomenon under investigation. We identified 13 strategies to improve DX in software reuse. These strategies address various aspects of software reuse, from organizational culture to technical infrastructure. Table 8 presents the identified strategy and their frequency across different studies.

Table 8: Strategies to Improve Developer Experience

Strategy	Study ID
Training and Incentives for Reuse	S1, S3, S5 and S6
Social Support for Software Reuse	S1, S8 and S9
Creating Discussion Forums	S4 and S6
Creating Standards and Tools	S1 and S10
Establishing Reuse Guidelines	S1 and S4
Cookbooks	S2
Defining Informal Communication Channels	S6
Early Release and Credible Promise	S9
Encouraging Experimentation	S1
Feedback and Evaluation	S1
Mentoring Programs	S4
Reference Manuals	S2
Top-Down Approach	S4

Training and Incentives for Reuse (S1, S3, S5 and S6). Providing developers with training programs on software reuse practices, tools, and techniques can enhance their skills and knowledge in effectively identifying, creating, and utilizing reusable components (S1). Training programs that educate developers on reuse principles, best practices, and the use of reuse repositories are essential for fostering a culture of reuse (S3).

The promotion and emphasis on reuse can vary across projects due to the client's attitude toward reuse and the project leader's experience. More experienced project leaders tend to prioritize reuse during development, indicating that reuse experience factors can moderate the internal promotion of reuse (S5 and S6).

Social Support for Software Reuse (S1, S8 and S9). Cultivating an organizational culture that values and promotes software reuse, recognizing and rewarding reuse efforts, and providing resources and support for reuse initiatives can motivate developers to engage actively in reuse practices (S1). Support from senior management, co-workers, and project managers plays a crucial role in developers' perceptions of the usefulness of software assets. Managers who encourage and support reuse and co-workers' positive perceptions and involvement are significant factors in promoting reuse (S8).

Additionally, developers with larger personal networks within the OSS (Open Source Software) community and experience in more projects have better access to reusable artifacts. This network allows them to find and integrate reusable code more efficiently, enhancing their ability to leverage software reuse (S9).

Creating Discussion Forums (S4 and S6). Promoting active communication between asset creators and users helps to identify reuse opportunities and address issues early. Informal knowledge transfer is particularly important in component reuse. Special interest groups or mailing lists dedicated to specific components or groups of similar components can facilitate this informal exchange. These forums allow component users to share knowledge, experiences, and best practices, enhancing their ability to utilize reusable assets effectively (S4 and S6).

Creating Standards and Tools. Investing in advanced tools and technologies that support software reuse can significantly enhance developers' ability to identify and reuse components efficiently. Integrated development environments (IDEs), version control systems, and repository management tools are crucial in streamlining reuse (S1). Developing standards and tools that facilitate the search and integration of software components can also reduce the costs of finding and using reusable components (S10).

Establishing Reuse Guidelines (S1 and S4). Developing clear guidelines and standards for software reuse is essential for streamlining the reuse process and guiding developers in effectively reusing components. This includes creating comprehensive documentation on reusable components, coding standards, and repository management practices (S1). Establishing clear policies and standards for creating, documenting, and using reusable assets ensures consistency and quality across the organization (S4).

Cookbooks are practical guides that present solutions to common problems within a specific framework. They combine code with natural language descriptions of the issues addressed, making them accessible and easily understood. Cookbooks are organized into "recipes," each addressing a separate problem within the framework (S2).

Defining Informal Communication Channels is essential for facilitating the flow of necessary information about reusable components among developers. Informal channels, such as casual conversations, chats, and impromptu meetings, can provide developers with quick access to valuable insights and experiences related to component usage (S6).

Early Release and Credible Promise. Adopting a development paradigm emphasizing releasing an initial functioning product version early can significantly enhance DX in software reuse. This strategy, known as delivering a "credible promise," increases the likelihood of reuse by demonstrating the project's feasibility and merit (S9).

Encouraging Experimentation. Encouraging developers to experiment with different reuse approaches, explore new technologies, and learn from successful and unsuccessful reuse experiences can foster a culture of continuous improvement in software reuse (S1). By implementing this strategy, organizations can cultivate an environment that values innovation and continuous improvement, enhancing DX and the effectiveness of software reuse.

Feedback and Evaluation. Establishing feedback mechanisms and regularly evaluating reuse practices can help to identify improvement areas, address developers' challenges, and refine reuse

strategies to better align with organizational goals (S1). Implementing structured feedback systems where developers can provide input on their reuse experiences and suggest improvements.

Mentoring Programs. Beyond formal training, mentoring programs where experienced developers guide less experienced ones can be highly effective. Mentors can provide practical insights and help new developers navigate the challenges of reuse (S4).

Reference Manuals describe the individual components of a software system, documenting each class by detailing its interface and role. These descriptions are typically organized into modules that represent the logical structure of the system. For example, JavaDoc extends this paradigm by incorporating hypertext links between related classes and using source code comments to generate descriptions, helping the documentation stay in sync with the code (S2).

Top-Down Approach. Management support is essential for successful software reuse initiatives. Organizations should promote a top-down approach, where top management actively supports and invests in reuse efforts. This includes setting strategic goals aligned with reuse and providing the necessary resources (S4).

6 DISCUSSION

The results of this study reveal various factors, barriers, and strategies that influence the developer's experience in software reuse. This discussion contextualizes these findings within existing literature and highlights implications for practice and further research.

6.1 Factors Affecting Developer Experience in Software Reuse

Our findings align with existing research emphasizing the importance of technical, organizational, and human/social factors in software reuse [2, 10]. Technical factors, such as comprehensive documentation (S4, S6, S7, and S10), understanding software functionality, interactions, and architecture (S2, S6, and S9), and availability of reuse-compatible Infrastructure (S3 and S8), are consistent with the literature indicating that clear documentation and well-structured systems are essential for effective reuse [20, 38]. Comprehensive documentation reduces the cognitive load on developers, making it easier for them to understand and integrate reusable components, thus improving their experience [14]. The influence of quality attributes (Non-Functional Requirements) (S7) and programming language (S9) on reuse effectiveness further supports previous studies highlighting the need for high-quality, language-compatible components [33]. Understanding software functionality and interactions helps developers make informed decisions about the applicability of reusable components, enhancing efficiency and satisfaction [12]. The availability of a robust, reuse-compatible infrastructure provides developers with the necessary tools and frameworks, simplifying the integration process and boosting their confidence and productivity [4].

Organizational factors, including project attributes (S3 and S5), client influence (S5), and perceptions about resource allocations (S8), underscore the role of organizational context in reuse practices. The project culture's impact on reuse (S5) and the necessity for (re)negotiation of requirements (S6) reflect findings from Amorim and Mendonça [2], which stress the importance of aligning organizational culture and project management practices with reuse goals.

Human/social factors, such as developer reuse experience (S1, S4, S5, S6, and S8), communication and collaboration (S4), and self-efficacy (S8), are pivotal in shaping reuse practices. This aligns with literature emphasizing the importance of developer skills, effective communication, and confidence in adopting new practices [30, 37]. Experienced developers are more adept at identifying and integrating reusable components, which enhances their productivity and job satisfaction [10]. High self-efficacy among developers leads to a more positive attitude towards reuse, as they feel confident in their ability to successfully implement reusable components [2].

6.2 Barriers to Improving Developer Experience in Software Reuse

The barriers identified, such as lack of top management support (S1, S3, S4, S5, S8, and S10) and insufficient training and education (S3, S4, S5, and S8), resonate with existing issues noted in the literature. Lack of management support hampers reuse efforts by limiting resources and strategic focus [10]. Insufficient training and education highlight the need for continuous learning and development to keep up with evolving reuse practices [1].

Resistance to change (S1, S3, S4, and S8) is another critical barrier, as documented by Johar et al. [22] who note that organizational and individual resistance can significantly impede the adoption of new technologies and practices. The lack of access to reusable components (S1 and S9) and perceived inefficiency in reuse (S9) highlight practical obstacles that developers face, corroborating findings from earlier studies [5].

Capilla et al. [5] observe that the lack of access to reusable components (S1 and S9) and perceived inefficiency in reuse (S9) are practical obstacles faced by developers. Their research indicates that although many reuse techniques have been integrated into modern software engineering processes, various factors still need to improve systematic reuse. In particular, resistance to change can be seen at both individual and organizational levels, where there is reluctance to adopt new practices due to fears of failure, training costs, and the need to alter established processes.

6.3 Strategies to Improve Developer Experience in Software Reuse

The strategies identified in this study, such as providing training and incentives for reuse (S1, S3, S5, and S6), social support for software reuse (S1, S8, and S9), and creating discussion forums (S4, and S6), align with best practices suggested in the literature [21, 50]. Training and incentives are essential for building skills and motivating developers to engage in reuse. Social support mechanisms and discussion forums facilitate knowledge sharing and collaborative problem-solving, which are crucial for successful reuse [9].

Creating standards and tools (S1 and S10) and establishing reuse guidelines (S1 and S4) are consistent with recommendations from Berger [39], who advocate for structured approaches to reuse. Cookbooks (S2) and reference manuals (S2) provide practical, accessible resources that help developers effectively understand and implement reuse practices.

Mentoring programs (S4) emphasize the importance of peer support and collective learning in fostering a culture of reuse. These strategies reflect findings from Chen et al. [9] on the value of communities of practice in organizational learning. Chen et al. [9] studied the reuse relationships within open-source communities and found that developers benefit significantly from collective intelligence and collaborative environments. Their research highlights that social computing and community-driven approaches enable developers to share knowledge, discover patterns, and improve software reuse practices through collective efforts.

7 THREATS TO VALIDITY

Some threats to the validity of this RR were identified. During this research, we sought to minimize the influence of these threats and reduce their possible risks. *Descriptive validity*: to mitigate this threat, a data collection form has been designed to support data recording to answer the questions. *Theoretical validity*: the studies were analyzed and selected under the aegis of DX by Fagerholm and Münch [12]. The search string was defined inclusively to capture studies related to concepts of DX in software reuse. Due to time and resource limitations, we could not use the forward snowballing technique. Additionally, since this study is an initial research phase, we plan to consider applying this technique in future stages of the study to expand the investigation. In addition, Despite its broad coverage, Scopus alone may not capture all relevant works. Even with the BS technique, important studies may have been overlooked, highlighting the need to include additional sources like IEEE Xplore and ACM for a more comprehensive review.

Generalizability: generalization is not a massive threat once we have used a structured protocol based on Cartaxo et al. [7] and Kitchenham and Charters [26], which facilitates replication. We also make available the datasets in the supplementary material. Constructing the conclusions of only 10 studies on a relatively underexplored topic poses a significant threat to generalization. To mitigate this threat, we sought to relate the findings of this study to other works in the literature. *Interpretive validity*: To minimize the researchers' bias, when there was doubt in executing the selection process, this was discussed extensively between the two researchers, and the differences were analyzed together by a third researcher until there was a consensus. It is worth highlighting that the protocol for RR needs to consider the quality of the retrieved studies.

8 CONCLUSION

This study used an RR to identify and analyze studies related to DX in software reuse. From 328 studies retrieved, 10 were selected for data extraction after applying defined filters and the backward snowballing technique. The analysis comprehensively explains the factors, barriers, and strategies shaping DX in software reuse.

The findings highlight the importance of technical, organizational, and human/social factors in influencing DX (RQ1). Key technical factors include documentation, understanding software functionality, interactions, architecture, and the availability of reuse-compatible infrastructure. They identify barriers (RQ2) such as lack of top management support, insufficient training and education, resistance to change, and access to reusable components. Addressing these barriers is essential for creating an environment conducive to software reuse and enhancing DX. Strategies identified (RQ3) include providing training and incentives for reuse, social support for software reuse, and creating discussion forums. These strategies emphasize the importance of peer support and collective learning in enhancing DX.

This study has implications for researchers and practitioners in software reuse. Researchers can explore the dynamic interplay of these factors, barriers, and strategies to gain deeper insights into improving software reuse practices. Practitioners can use these insights to enhance DX and achieve greater efficiency, productivity, and innovation in software development.

We can identify some future work and opportunities from the results of this study, such as: i) Investigating the relationship between factors in the context of software reuse to understand their interactions within the software industry; and ii) Developing practical guidelines and recommendations based on this study's insights to enhance positive factors and mitigate negative influences on software reuse. These guidelines can help organizations implement effective reuse practices and improve DX.

ARTIFACTS AVAILABILITY

The raw data and all the steps necessary to reproduce the research are detailed in the supplementary material openly available via ZENODO[4].

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001 and CNPq.

REFERENCES

[1] Nazakat Ali, Horn Daneth, and Jang-Eui Hong. 2020. A hybrid DevOps process supporting software reuse: A pilot project. *Journal of Software: Evolution and Process* 32, 7 (2020), e2248. https://doi.org/10.1002/smr.2248

[2] Luiz Amorim and Manoel Mendonça. 2016. A method to support the adoption of reuse technology in large software organizations. In *Software Reuse: Bridging with Social-Awareness: 15th International Conference, ICSR*. 73–88. https://doi.org/10.1007/978-3-319-35122-3_6

[3] José L Barros-Justo, David N Olivieri, and Fernando Pinciroli. 2019. An exploratory study of the standard reuse practice in a medium sized software development firm. *Computer Standards & Interfaces* 61 (2019), 137–146. https://doi.org/10.1016/j.csi.2018.06.005

[4] José L. Barros-Justo, Fernando Pinciroli, Santiago Matalonga, and Nelson Martínez-Araujo. 2018. What software reuse benefits have been transferred to the industry? A systematic mapping study. *Information and Software Technology* 103 (2018), 1–21. https://doi.org/10.1016/j.infsof.2018.06.003

[5] Rafael Capilla, Barbara Gallina, Carlos Cetina, and John Favaro. 2019. Opportunities for software reuse in an uncertain world: From past to emerging trends. *Journal of software: Evolution and process* 31, 8 (2019), e2217. https://doi.org/10.1002/smr.2217

[6] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2018. The role of rapid reviews in supporting decision-making in software engineering practice. In *International Conference on Evaluation and Assessment in Software Engineering 2018*. 24–34. https://doi.org/10.1145/3210459.3210462

[7] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2020. Rapid reviews in software engineering. *Contemporary Empirical Methods in Software Engineering* (2020), 357–384. https://doi.org/10.1007/978-3-030-32489-6_13

[8] Kathy Charmaz. 2006. *Constructing grounded theory: A practical guide through qualitative analysis.* Sage Publications, Thousand Oaks.

[9] Mengwen Chen, Tao Wang, Cheng Yang, Qiang Fan, Gang Yin, and Huaimin Wang. 2016. Social Computing in Open Source Community: A Study of Software Reuse. In *Social Computing: International Conference of Young Computer Scientists,*

[4] https://doi.org/10.5281/zenodo.12685772

Engineers and Educators, ICYCSEE. Springer, 621–631. https://doi.org/10.1007/978-981-10-2053-7_55

[10] Xingru Chen, Deepika Badampudi, and Muhammad Usman. 2022. Reuse in Contemporary Software Engineering Practices–An Exploratory Case Study in A Medium-sized Company. *e-Informatica Software Engineering Journal* 16, 1 (2022). https://doi.org/10.37190/e-Inf220110

[11] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*. 275–284.

[12] Fabian Fagerholm and Jürgen Münch. 2012. Developer experience: Concept and definition. In *International conference on software and system process (ICSSP)*. 73–77. https://doi.org/10.1109/ICSSP.2012.6225984

[13] Awdren Fontão, Arilo Dias-Neto, and Davi Viana. 2017. Investigating Factors That Influence Developers' Experience in Mobile Software Ecosystems. In *International Workshop on Software Engineering for Systems-of-Systems*. 55–58. https://doi.org/10.1109/JSOS.2017.10

[14] Michaela Greiler, Margaret-Anne Storey, and Abi Noda. 2022. An actionable framework for understanding and improving developer experience. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1411–1425. https://doi.org/10.1109/TSE.2022.3175660

[15] Martin Griss, Ivar Jacobson, Chris Jette, Bob Kessler, and Doug Lea. 1995. Systematic software reuse (panel) objects and frameworks are not enough. In *Proceedings of the 1995 Symposium on Software reusability*. 17–20. https://doi.org/10.1145/223427.213969

[16] Jenny Guber and Iris Reinhartz-Berger. 2023. Privacy-Compliant Software Reuse in Early Development Phases: A Systematic Literature Review. *Information and Software Technology* (2023), 107351. https://doi.org/10.1016/j.infsof.2023.107351

[17] Michelle M Haby, Evelina Chapman, Rachel Clark, Jorge Barreto, Ludovic Reveiz, and John N Lavis. 2016. What are the best methodologies for rapid reviews of the research evidence for evidence-informed decision making in health policy and practice: a rapid review. *Health research policy and systems* 14 (2016), 1–12. https://doi.org/10.1186/s12961-016-0155-7

[18] Candyce Hamel, Alan Michaud, Micere Thuku, Becky Skidmore, Adrienne Stevens, Barbara Nussbaumer-Streit, and Chantelle Garritty. 2021. Defining rapid reviews: a systematic scoping review and thematic analysis of definitions and defining characteristics of rapid reviews. *Journal of Clinical Epidemiology* 129 (2021), 74–85. https://doi.org/10.1016/j.jclinepi.2020.09.041

[19] Catherine M Hicks, Carol S Lee, and Morgan Ramsey. 2024. Developer Thriving: four sociocognitive factors that create resilient productivity on software teams. *IEEE Software* (2024). https://doi.org/10.1109/MS.2024.3382957

[20] Stanislaw Jarzabek and Daniel Dan. 2017. Documentation reuse: Managing similar documents. In *International Conference on Information Reuse and Integration (IRI)*. 372–375. https://doi.org/10.1109/IRI.2017.52

[21] Xiaoyu Jin, Charu Khatwani, Nan Niu, Michael Wagner, and Juha Savolainen. 2016. Pragmatic software reuse in bioinformatics: How can social network information help?. In *Software Reuse: Bridging with Social-Awareness: International Conference*. Springer, 247–264. https://doi.org/10.1007/978-3-319-35122-3_17

[22] Monica Johar, Vijay Mookerjee, and Suresh Sethi. 2015. Optimal software design reuse policies: A control theoretic approach. *Information Systems Frontiers* 17 (2015), 439–453. https://doi.org/10.1007/s10796-013-9421-1

[23] Huma Hayat Khan and Muhammad Noman Malik. 2017. Software standards and software failures: a review with the perspective of varying situational contexts. *IEEE access* 5 (2017), 17501–17513. https://doi.org/10.1109/ACCESS.2017.2738622

[24] Valerie J King, Adrienne Stevens, Barbara Nussbaumer-Streit, Chris Kamel, and Chantelle Garritty. 2022. Paper 2: Performing rapid reviews. *Systematic Reviews* 11, 1 (2022), 151. https://doi.org/10.1186/s13643-022-02011-5

[25] D. Kirk, M. Roper, and M. Wood. 2002. Defining the problems of framework reuse. In *Proceedings 26th Annual International Computer Software and Applications*. 623–626. https://doi.org/10.1109/CMPSAC.2002.1045073

[26] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Evidence-Based Software Engineering (EBSE) Project.

[27] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE 2007-001. Keele University and Durham University Joint Report.

[28] Nam-Yong Lee and C.R. Litecky. 1997. An empirical study of software reuse with special attention to Ada. *IEEE Transactions on Software Engineering* 23, 9 (1997), 537–549. https://doi.org/10.1109/32.629492

[29] Jingyue Li, Reidar Conradi, Parastoo Mohagheghi, Odd Are Sæhle, Øivind Wang, Erlend Naalsund, and Ole Anders Walseth. 2004. A study of developer attitude to component reuse in three IT companies. In *Product Focused Software Process Improvement: 5th International Conference, PROFES*. 538–552. https://doi.org/10.1007/978-3-540-24659-6_39

[30] Niko Mäkitalo, Antero Taivalsaari, Arto Kiviluoto, Tommi Mikkonen, and Rafael Capilla. 2020. On opportunistic software reuse. *Computing* 102 (2020), 2385–2408. https://doi.org/10.1007/s00607-020-00833-6

[31] Vidhya Mellarkod, Radha Appan, Donald R Jones, and Karma Sherif. 2007. A multi-level analysis of factors affecting software developers' intention to reuse software assets: An empirical investigation. *Information & Management* 44, 7 (2007), 613–625. https://doi.org/10.1016/j.im.2007.03.006

[32] H. Mili, F. Mili, and A. Mili. 1995. Reusing software: issues and research directions. *IEEE Transactions on Software Engineering* 21, 6 (1995), 528–562. https://doi.org/10.1109/32.391379

[33] Sonia Montagud, Silvia Abrahão, and Emilio Insfran. 2012. A systematic review of quality attributes and measures for software product lines. *Software Quality Journal* 20 (2012), 425–486. https://doi.org/10.1007/s11219-011-9146-7

[34] Jenny Morales, Cristian Rusu, Federico Botella, and Daniela Quiñones. 2019. Programmer eXperience: A Systematic Literature Review. *IEEE Access* 7 (2019), 71079–71094. https://doi.org/10.1109/ACCESS.2019.2920124

[35] Rebeca C. Motta, Káthia M. de Oliveira, and Guilherme H. Travassos. 2019. A conceptual perspective on interoperability in context-aware software systems. *Information and Software Technology* 114 (2019), 231–257. https://doi.org/10.1016/j.infsof.2019.07.001

[36] Abi Noda, Margaret-Anne Storey, Nicole Forsgren, and Michaela Greiler. 2023. DevEx: What Actually Drives Productivity: The developer-centric approach to measuring and improving productivity. *Queue* 21, 2 (2023), 35–53. https://doi.org/10.1145/3595878

[37] Mohd Akmal Faiz Osman, Mohamad Noorman Masrek, and Khalid Abdul Wahid. 2022. Software Reuse Practices among Malaysian Freelance Developers: A Conceptual Framework. In *Proceedings*, Vol. 82. MDPI, 30. https://doi.org/10.3390/proceedings2022082030

[38] Mohamed A Oumaziz, Alan Charpentier, Jean-Rémy Falleri, and Xavier Blanc. 2017. Documentation reuse: Hot or not? An empirical study. In *16th International Conference on Software Reuse*. 12–27. https://doi.org/10.1007/978-3-319-56856-0_2

[39] Iris Reinhartz-Berger. 2024. Challenges in software model reuse: cross application domain vs. cross modeling paradigm. *Empirical Software Engineering* 29, 1 (2024), 16. https://doi.org/10.1007/s10664-023-10386-9

[40] M.A. Rothenberger, K.J. Dooley, U.R. Kulkarni, and N. Nada. 2003. Strategies for software reuse: a principal component analysis of reuse practices. *IEEE Transactions on Software Engineering* 29, 9 (2003), 825–837. https://doi.org/10.1109/TSE.2003.1232287

[41] Marcus A Rothenberger. 2003. Project-Level Reuse Factors: Drivers for Variation within Software Development Environments. *Decision sciences* 34, 1 (2003), 83–106. https://doi.org/10.1111/1540-5915.02252

[42] Di Shang, Karl Lang, and Roumen Vragov. 2022. A Market-Based Approach to Facilitate the Organizational Adoption of Software Component Reuse Strategies. *Communications of the Association for Information Systems* 51, 1 (2022), 36. https://doi.org/10.17705/1CAIS.05140

[43] Karma Sherif and Ajay Vinze. 2003. Barriers to adoption of software reuse: A qualitative study. *Information & Management* 41, 2 (2003), 159–175. https://doi.org/10.1016/S0378-7206(03)00045-4

[44] Gustavo Silva, Carla Bezerra, Anderson Uchôa, and Ivan Machado. 2023. What Factors Affect the Build Failures Correction Time? A Multi-Project Study. In *Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '23)*. 41–50. https://doi.org/10.1145/3622748.3622753

[45] Odd Petter N Slyngstad, Anita Gupta, Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre. 2006. An empirical study of developers views on software reuse in statoil asa. In *ACM/IEEE International Symposium on Empirical Software Engineering*. 242–251. https://doi.org/10.1145/1159733.1159770

[46] Dalia Sobhy, Rami Bahsoon, Leandro Minku, and Rick Kazman. 2021. Evaluation of software architectures under uncertainty: A systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–50. https://doi.org/10.1145/3464305

[47] Manuel Sojer and Joachim Henkel. 2010. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems* 11, 12 (2010), 868–901. https://doi.org/abstract=1489789

[48] IO Standardization. 2019. Part 210: Human-centred design for interactive systems. (2019).

[49] Sebastian Spaeth Stefan Haefliger, Georg von Krogh. 2008. Code Reuse in Open Source Software. *Management Science* 54, 1 (2008), 180–193. https://doi.org/10.1287/mnsc.1070.0748

[50] Hongyi Sun, Waileung Ha, Min Xie, and Jianglin Huang. 2015. Modularity's impact on the quality and productivity of embedded software development: a case study in a Hong Kong company. *Total Quality Management & Business Excellence* 26, 11-12 (2015), 1188–1201. https://doi.org/10.1080/14783363.2014.920179

[51] Sri Lakshmi Vadlamani and Olga Baysal. 2020. Studying software developer expertise and contributions in Stack Overflow and GitHub. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 312–323. https://doi.org/10.1109/ICSME46990.2020.00038

[52] Julia Varnell-Sarjeant and Anneliese Amschler Andrews. 2015. Comparing reuse strategies in different development environments. In *Advances in Computers*. Vol. 97. Elsevier, 1–47. https://doi.org/10.1016/bs.adcom.2014.10.002