Computing (2020) 102:2385–2408
https://doi.org/10.1007/s00607-020-00833-6

REGULAR PAPER

On opportunistic software reuse

Niko Mäkitalo[1] · Antero Taivalsaari[2] · Arto Kiviluoto[1] · Tommi Mikkonen[1] · Rafael Capilla[3]

Received: 27 February 2020 / Accepted: 2 July 2020 / Published online: 10 July 2020

Abstract
The availability of open source assets for almost all imaginable domains has led the software industry to *opportunistic design*—an approach in which people develop new software systems in an ad hoc fashion by reusing and combining components that were not designed to be used together. In this paper we investigate this emerging approach. We demonstrate the approach with an industrial example in which *Node.js* modules and various subsystems are used in an opportunistic way. Furthermore, to study opportunistic reuse as a phenomenon, we present the results of three contextual interviews and a survey with reuse practitioners to understand to what extent opportunistic reuse offers improvements over traditional systematic reuse approaches.

Keywords Software reuse · Software engineering · Opportunistic design · Opportunistic reuse · Software architecture · Code snippet

Mathematics Subject Classification 68-04

✉ Tommi Mikkonen
tommi.mikkonen@helsinki.fi

Niko Mäkitalo
niko.makitalo@helsinki.fi

Antero Taivalsaari
antero.taivalsaari@nokia-bell-labs.com

Arto Kiviluoto
arto.kiviluoto@alumni.helsinki.fi

Rafael Capilla
rafael.capilla@urjc.es

[1] Department of Computer Science, University of Helsinki, Helsinki, Finland

[2] Nokia Bell Labs, Tampere, Finland

[3] Department of Computer Science, Rey Juan Carlos University, Madrid, Spain

1 Introduction

Ever since software reuse became a software engineering discipline in the early 1970s, there has been a call for systematic software reuse practices. The key idea in reuse is that parts of a computer program written at one time can be used in the construction of other programs later [15]. Reusability is typically considered as one of the key "ilities" or major software quality factors [15]. However, it has turned out that for various reasons achieving the benefits of reuse and reusable software is difficult [40].

Today, software reuse takes place in many ways. The evolution of software artifacts from mere source code to modern forms of reuse as services and microservices [4] has changed the way developers integrate functionality into their systems. Moreover, the systematization offered by software product line engineering practices for building and customize reusable core assets into a common product line architecture [9] is a popular development strategy nowadays. However, the current popularity of today's open source repositories has brought a shift in which developers search and reuse code modules seemingly without any systematic reuse method and in which software designs are created and modified opportunistically based on available third-party components.

Opportunistic design is an approach in which people develop new software systems by reusing and combining components that were not designed to be used together [18, 34]. This emergent pattern places focus on large scale reuse and developer convenience, with the developers "trawling" for most suitable open source components and modules online. The availability of open source assets for almost all imaginable domains has led to software systems in which the visible application code—written by the application developers themselves—forms only the "tip of the iceberg" compared to the reused bulk that remains mostly unknown to the developers [44]. While often immensely productive in short term, this approach implies that reuse takes place in a rather ad hoc, mix-and-match fashion.

This paper examines opportunistic design and new forms of software reuse, and presents an industry system where we discuss opportunistic reuse. Furthermore, in order to gather more evidence, we conducted a number of interviews and ran a survey to investigate how widely this practice has spread in the software industry and to understand how developers apply new forms of reuse when creating real-world software systems. The work is an extended version of a short insights article recently published in IEEE Software [34] and a position paper that focused on practical challenges of opportunistic design [44]. While the goal of the earlier short papers was to increase the awareness of opportunistic reuse practices, the goal of this extended version is to provide a coherent view of the phenomenon and to truly connect it to academic background and previous work.

The rest of this paper is structured as follows. In Sect. 2, we provide the necessary background and motivation for the paper. In Sect. 3, we focus on the fundamentals of software reuse in light of the new, emerging forms of reuse. In Sect. 4, we present a motivating example following the "tip of the iceberg" development model that is typical today. In Sect. 5, we describe our research approach. In Sect. 6, we provide three contextual interviews from practitioners as confirming evidence of opportunistic reuse practices. In Sect. 7, we present a survey regarding emerging practices of

opportunistic reuse, with data collected from the industry. In Sect. 8, we provide an extended discussion of our findings and their implications for practitioners. In Sect. 9, we present threats to the validity of this work. In Sect. 10, we address related work. Finally, in Sect. 11 we discuss the conclusions. Sections 2–4 are based on our previous papers, whereas Sects. 5-11 are new material that includes our results.

2 Background and motivation

Originally introduced in the NATO Software Engineering Conference in 1968 [35], software reuse started becoming a popular research topic in the 1980s [2,31,32]. Since then, software reuse as research topic blossomed, reaching maturity in the 1990s, after the introduction and adoption of software product line engineering (SPLE) practices and Microsoft's component technologies.

Compared to systematic reuse methodologies [25,31] proposed more than twenty years ago, today people routinely trawl for ready-made solutions for specific problems online and try to discover libraries and code snippets to be included in applications with little consideration or knowledge about their technical quality. The availability of reusable open source assets has changed the nature of system development profoundly, with developers rarely performing reuse in a planned or institutionalized way such as in product line development [8]. Also, with the popularity of service-based components [3,49], microservices, and open source repositories, the systematization of reuse techniques adopted by SPLs have become gradually less formal and more opportunistic. Today, developers tend to reuse software trawling and scraping the Internet for most suitable or most easily available components at the time when they need them. Quite often, the developer has no idea of what code or how much code they are actually reusing, since the included components may dynamically pull in hundreds or even thousands of additional subcomponents; it may simply be infeasible to analyze all the dependencies and follow their evolution over time.

This approach is all about combining unrelated, often previously unknown hardware and software artifacts by joining them with "duct tape and glue code" [18]. Opportunistic reuse is common in spite of components potentially having unknown safety-related characteristics or having been developed by unknown developers using unknown methodologies [52]. Although it is widely admitted that opportunistic designs are not automatically optimal and such designs may require significant architectural adjustments to fulfill functional and non-functional requirements [41], developers have embraced this approach in droves. Component selection is often based simply on popularity ratings or recommendations from other developers [34].

In contrast with classical forms of reuse aimed to produce high-quality components [16,48], reuse of assets that have not been originally designed with reuse in mind can lead to serious quality-related issues. Apart from potential quality issues, the scale of opportunistic design and ad hoc reuse has received surprisingly little attention among researchers, but nowadays is becoming a more relevant topic of study for many research groups.

3 Opportunistic reuse as a new trend

According to Krueger's classic survey paper on software reuse [27], a successful software reuse technique must fulfill the following four "truisms":

1. For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.
2. For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.
3. To select an artifact for reuse, you must know what it does.
4. To reuse a software artifact effectively, you must be able to "find it" faster than you could "build it".

In relation to Krueger's observations, the situation has changed radically within a short period of time. In the 1980s, studies showed that a considerable amount of time in software development was spent on designing routines and structures that are almost identical with constructs used in other programs. In 1984, Jones reported that on average only 15% of code is truly unique, novel and specific to individual applications; the remaining 85% appeared to be common, generic, and concerned with making the program to cooperate with the surrounding execution environment [23]. Other studies in the 1980s reported potential reuse rates between 10 and 60% [2,31,32].

Although the potential for software reuse was high in the 1980s and early 1990s, actual reuse rates remained low. Those days developers actually preferred writing their own code, and took pride on doing as much as possible from scratch. In fact, they were effectively expected or forced to do so, since third-party components were not widely available or easy to find. Furthermore, before the widespread adoption of open source software development, components were rarely available for free or with license terms favoring large-scale use, apart from standard operating system and language libraries.

Today, many companies and individuals are actually proud about the amount the third-party code in their products. For instance, well-known car manufacturers, such as Bentley and Volvo, proudly boast in their advertisements and reviews about the large amount of software in their cars, as if it was categorically a good thing [51]. For example, the 2018 version of the Bentley Continental GT is said to contain "*93 processors, feeding more than a 100 million lines of code through eight kilometers of wiring*".[1] Thus, software reuse has finally happened in massive scale, but it has happened in an entirely different way than envisioned by the software engineering community.

As opposed to the situation in the 1980s and 1990s when the amount of reused software formed only a fraction of the entire software systems, the situation is now decidedly the contrary. While opportunistic designs promise short development times and rapid deployment, developers are becoming accustomed to designs that they do not fully understand, and yet using them even in domains that require high attention to security and safety. As mentioned before, opportunistic reuse and the "tip-of-the-iceberg" development model has fundamentally reshaped the way software systems are constructed. Instead of systematically written, organized software libraries available

[1] http://edition.cnn.com/style/article/bentley-continental-gt/index.html.

for purchase, there is a cornucopia of free, overlapping open source components of varying quality available for nearly all imaginable tasks and domains. This is in striking contrast with the situation in the late 1990s when software reuse was declared dead.[2]

The success of reuse in general, and opportunistic design in particular, in the past years can be attributed largely to second and fourth points raised by Krueger [27]. The Web and its search engines have made it easy to search for potentially applicable software components and code snippets online. In addition, the availability of software in open source form has made it easy to experiment with potentially applicable components without any significant financial commitment ahead of time. Yet, contrary to Krueger's [27] third point—"to select an artifact for reuse, you must know what it does"—it can be argued that the overall understanding of the reused software has decreased over the years. After all, the key premise in "classic" software reuse is that there are systematically designed systems with stable, well-documented interfaces. The sheer volume of open source software makes it difficult to analyze and compare technologies in detail, let alone fully understand the abstractions exposed by them, especially in light of the highly varying quality of documentation that is characteristic of many open source projects.

To summarize, the basic problem in opportunistic design is that it does not follow any systematic, abstraction-driven approach. Instead, as characterized by Hartmann et al. [18], developers end up creating significant systems by hacking, mashing and gluing together disparate, continually evolving components that were not designed to work together. Developers publishing such components often have no formal training in creating high-quality software components, and the developers performing opportunistic, ad hoc reuse might not have any professional skills for selecting and combining such components. From our observations from the existing literature and experience in reusing software, we jointly synthesized the following list of challenges:

- What are the current or most popular reuse practices, specifically those concerned with non-systematic approaches?
- How do developers search and reuse code in today's software projects?
- How do developers integrate the reused code into their existing systems and architectures?
- What checks do reuse practitioners perform before they integrate the reusable code snippets into their own system?

We will answer the challenges above in the following sections.

4 An industrial example

Our industrial example is based on an IoT development project in which the goal was to design and implement a scalable cloud back-end for an IoT system for multiple use cases. In particular, some of the use cases required that large amounts of measurement data could be collected—generated by various types of data-intensive measurement devices—and then streamed in continuously to our back-end at high data rates. Besides streaming data, real-time data analytic was needed. This feature consisted of analyzing

[2] https://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html.

and processing the incoming data in near real-time, and then generating visualizations, actuation requests, alerts and other actions with minimal latency. Inside the back-end, query mechanisms were introduced for accessing collected data (time series), with various query parameters and query types (e.g., from a certain sensor and within the given time range). Finally, a notification mechanism was introduced to provide alerts when data values reached certain predefined criteria.

4.1 Architecture design

The baseline architecture of our system was largely prescribed by the domain of IoT systems, in essence forming an end-to-end data pipeline. In this pipeline, the data flow from measurement devices via a cloud backend to applications. Within the pipeline, the cloud introduces numerous functions, including in particular data acquisition, analytics, storage, and access and notification mechanisms.

In order to make it easier to add new domain-specific functionality, components and APIs on top of the base system, a microservice based architecture [36] was selected as a generic solution for plugging in additional components without interfering with the rest of the system.

From the very beginning of the development, it was obvious that we would have neither interest nor resources to develop the complete system from scratch. Therefore, we decided to rely extensively on available third-party open source software for areas that were not at the focus of the development, such as streaming data acquisition and real-time analytics functionality, both of which require complex solutions.

In addition, our system included a lot of "bread-and-butter" cloud back-end functionality such as user identity management (user accounts and access permissions), device management, logging and monitoring capabilities, and some administrative tools for managing the overall system. Concrete key subsystems include *NGINX*[3] for security perimeter, *Apache Kafka*[4] for data acquisition, *Apache Storm*[5] for real-time data analytics, and *Graphite*[6] and *Icinga*[7] for logging and monitoring.

We also wanted to have a flexible, scalable cloud deployment model that was not physically tied to any particular machines, data centers or vendors. To meet these requirements, we used Docker[8] and Docker Swarm.[9] The deployment model had to include the ability to easily deploy multiple instances of the entire cloud environment onto different types of cloud environments, including OpenStack[10] in particular.

For the implementation of microservices, our developers tapped extensively into the rich NPM module ecosystem; about half of the microservices are written on top of Node.js. In microservice implementations, the number of NPM modules (transitive

[3] https://nginx.org/.

[4] https://kafka.apache.org/.

[5] http://storm.apache.org/.

[6] https://graphiteapp.org/.

[7] https://www.icinga.com/.

[8] https://www.docker.com/.

[9] https://docs.docker.com/engine/swarm/.

[10] https://www.openstack.org/.

closure of all the NPM modules pulled in by each microservice) varies from a few dozen to over a thousand per microservice. Cumulatively, the total number of different NPM modules (excluding duplicates) used by the system exceeds two thousand. While many of those NPM modules are very simple, such as *uuid*, there are also significantly more complex ones such as *core-js*, *shelljs*, or *redux*. Overall, we estimate that only about 3-5% of the source code of the system was written by our developers, while the vast majority comes from third-party open source components. The exact percentage is not easy to measure given the transitive and overlapping dependencies in the dynamically loaded modules and the fact that dependencies may change in different versions of the components.

4.2 Analysis on reuse and its effectiveness

For software developers who are familiar with modern cloud backend technologies, the selections of key components and subsystems are relatively easy to justify. These selections meet Krueger's software reuse "truisms" (see Sect. 3) with flying colors: all the key subsystems mentioned above are so complex that developing similar functionality from scratch would have been prohibitively expensive (Krueger's Principle 2). These technologies made the implementation of the system considerably faster, thus reducing the cognitive distance between the original concept and final implementation (Principle 1). These systems are so well documented that the "finding them" and "knowing what they do" principles were relatively easy to fulfill (Principles 3 and 4).

In contrast, the technology selections for the Node.js based microservices are more difficult to justify. According to the developers, many of the NPM modules were chosen based on module popularity ratings alone. In the end, because of the extensive transitive dependencies in NPM modules, the system contains literally hundreds of NPM modules that were not explicitly chosen by the developers but which were pulled in because of the dependencies. We have characterized software reuse in the implementation of the example in light of Krueger's software reuse principles a bit further in Table 1.

The motivating industrial example presented above is by no means unique in its very high reuse ratio. There are various factors to contribute to such rates, such as routinely using containers and virtual machines. Libraries that are included in designs represent only a fraction of the features needed, and standardized deployment pipelines as well as other infrastructure enable deployment in the large. A further influencing factor in this example is that the team was largely self-organized and free to select components that were considered well suited for their project, but also motivated to study components on their own right. Links to systems, subsystems and modules were discovered via different sources, in particular from the Stack Overflow, NPM repository, as well as from developers' personal contacts and favourite web sites. Although strict scrutiny was applied to the selected components, this technology selection process contributed to high reuse rates, simply because more widely used, popular and mature libraries and frameworks tend to have a large number of dependencies that pull in hundreds of additional components.

Table 1 Evaluation of the industrial example in view of Krueger's truisms/principles

Krueger's truism	Observations in the industrial example
1 An effective reuse technique reduces the cognitive distance between the concept and its implementation	The selection of the key components and subsystems mentioned above sped up development considerably, and allowed the limited development resources to be used on implementing domain-specific functionalities
2 For a software reuse technique to be effective, it must be easier to reuse the artifacts than building software from scratch	Considering the scale of the system being built and the effort it takes to build such software, it would have been impossible to build the same system from scratch with the same resources and in the same timeframe. The use of new software technologies such as containers facilitated the development of a more scalable solution
3 To select an artifact for reuse, you must know what it does	This is a pain point for opportunistic reuse. In selecting the main implementation technologies, developers resorted primarily to their past experience, choosing technologies that were fairly well understood ahead of time. However, in choosing the NPM modules that were used for implementing the microservices, the developers had to rely extensively on 3rd party recommendations, ultimately picking numerous components that were unfamiliar at the beginning of the project. Moreover, the external dependencies of those components were originally unknown
4 To reuse a software artifact effectively, you must be able to find it faster than you could build it	In the era of Internet, it can be remarkably easy to search possible implementation technologies and components. In our project, we noticed that developers were often overwhelmed by the amount of possible overlapping component offerings especially in the NPM ecosystem, though. While finding components can be easy, choosing the *right* ones has not become easier. This is an area where developer's experience plays a significant role

5 Research method

In this section we describe the research approach that we followed. As we planned to investigate opportunistic reuse as a phenomenon, we decided to carry out an exploratory case study as described in [10,50] to find out similarities and differences among different opportunistic reuse practices. To carry out our study we first defined the following research questions derived from the challenges stated in Sect. 3.

RQ1. *How does opportunistic reuse appear in practice?*
Rationale: In this research question we want to investigate which are the main reasons by which opportunistic reuse appears.
RQ2. *How do the developers select the reusable assets when using an opportunistic reuse strategy?*
Rationale: This question analyzes how developers reuse software when they do not follow a classic, systematic and rational process for component selection and development.

RQ3. *What is the expected architectural impact arising from the practice of opportunistic reuse?*
Rationale: Finally, our last question analyzes what are the expected benefits of opportunistic reuse compared to traditional, systematic approaches.

Basically, the first two challenges described in Sect. 3 are addressed by research questions RQ1 and RQ2, while the third challenge is addressed by RQ3. Our fourth challenge is partially addressed in RQ3 but also by one of the questions we describe in our only survey and direct interviews as well. To address the aforementioned research questions we first collected some initial insight from three direct interviews with practitioners and developers in our universities (Sect. 6) and after we carried out an online survey (Sect. 7) with international practitioners involved in reuse activities. The questions we asked the subjects in the online survey were derived from the insight gained during the interviews.

6 Evidence gathered from university projects

To further investigate the process of opportunistic design, we conducted three contextual interviews with practitioners from two research groups from *Universidad Rey Juan Carlos* (URJC, Spain) and *University of Helsinki* (Finland). Contextual interviews are a kind of think-aloud protocols [11,19] commonly used to gain insight into the environment or context in which a design will be used. Therefore, we gathered qualitative data as stated by [39] to provide additional insight on how practitioners do opportunistic reuse. We selected interviewed research groups using the following criteria: (i) Groups involved in reuse practices of open source components, (ii) Groups with at least five years of experience reusing open source software, (iii) Local colleagues or groups that we can approach easily to perform direct interviews.

In the interviews, we asked the following questions:

(i) Which repositories you prefer to use for finding reusable assets?
(ii) How do you search for reusable assets?
(iii) How do you ascertain that a reused code asset fits in your system or software project?
(iv) How do you integrate the reused code into your system?
(v) Do you check the conformity of the reused asset with the existing architecture?

First interview. In the first interview we asked two researchers involved in a research project called ElasTest,[11] an H2020 research project aimed to create an elastic platform for testing complex distributed large software systems in the cloud. Both researchers have more than 10 years of experience in developing and reusing code assets, and they are leaders of their research group as well. The key answers to the aforementioned questions are as follows.

(i, ii, iii) People from ElasTest mainly search in Google for a reusable asset in order to solve a programming issue, and then utilize Stack Overflow mentioned above to find relevant discussions. Then, they look for similar questions searching for responses

[11] https://elastest.eu/project.html.

with highly ranked answers, and review the responses manually and select the code snippet that fits best to their situation.

(iv, v) Usually both researchers stated they perform some security checks before they integrate the code into the system; afterwards they run the usual tests defined in the project. Only in those cases where they need to upgrade the version of the existing platform caused by a requirement of the code reused they check the conformity with the existing software architecture for adaptation purposes.

Second interview. Our second interview targeted researchers and developers from research groups at URJC (Spain). We interviewed two of their research leaders with more than 15 years of experience in open source software projects and mining software repositories. Both are researchers and founders of a start-up called Bitergia,[12] and they have experience in reusing software artifacts. All interviewees from research groups have a PhD in computer science, and they range between 39 and 45 years old. The answers we received for the five questions are summarized below.

(i, ii) The repositories most frequently used for finding reusable components are Stack Overflow but also in Python.org repositories. The search strategy used is Google search looking for the first non-sponsored match. If it is on Stack Overflow or a similar repository, they keep looking for reviews and ratings there. In other case they go back to the search results and look for the second non-sponsored match.

(iii) Regarding how the subjects can be sure if a code snippet is applicable: When they are not sure they use the Python interpreter, and test it before introducing the asset into the code.

(iv, v) Finally, the integration is done using simple copy-paste of the code; some adaptation is typically required, e.g., to change conflicting variable names. The subjects said that checking the conformity of the reused asset with the existing architecture is not relevant for Python3 code.

Third interview. Our third interview concerns developers of Toska group, Department of Computer Science, University of Helsinki, where total of two developers were interviewed. The goal of the group is to provide administrative tools for the department, and the age of the interviewees is between 20-30. Both interviewees also have experience working for companies outside of the university context.

(i, ii) Stack Overflow is used for finding snippets as well as links to bigger components. In addition, NPM is routinely used for smallish functions, Github for complete (sub-)systems, and Docker Hub for container-related issues. Asking or checking from Stack Overflow is done routinely. When working on unfamiliar territory, the strategy used has three steps: first, figure out what others are using; secondly, share the view with the team, and finally make a personal decision.

(iii) When selecting a module to reuse, first the developers check that the module has the correct features. If necessary, the missing features are added and a pull request is sent to the project, but only if the addition is architecturally sound. If necessary, own implementation following the baseline design of an existing component can be used, for instance if the necessary additions contradict the original design or the used licence does not match the needs, or there are support issues with the community. Once the module is running, it is possible to consider performance and other secondary issues.

[12] https://bitergia.com/.

(iv, v) Modifications are made on the need basis to systems that are familiar enough or small enough. In contrast, larger systems such as GUI libraries, databases and so on are taken as given and not modified at all, and rest of the system will be adapted to accommodate them. The process of integration also depends on which repository the code is from. For Github, for instance, a common approach is to install the entire repository and follow installation instructions. In contrast, for NPM or Dockerhub, command line interface is preferred, so that a package can be deleted as effortlessly as it was installed in the first place.

Conformance and license checks are made upon integration, although this partly depends on what kinds of reusable assets are being reused. To summarize, in all interviews about opportunistic reuse, its seems a search using Stack Overflow is a usual practice in this kind of research projects where software development tasks are carried out. In both cases the subjects stated they employ similar techniques as opportunistic reuse strategy. Nevertheless, the alignment of the code with the architecture is only performed in those cases where interoperability requirements demand this, as in other cases developers assume the code snippets reused should fit under the existing architecture.

7 Survey design

According to the guidelines defined in [38], we performed a survey to confirm our pilot study based on the contextual interviews and find evidence with practitioners about opportunistic reuse practices.

7.1 Setting up the survey

Planning the survey. We carried out an online survey with industry practitioners and researchers in the field of software engineering asking 9 questions about their experience in reusing software, with special focus on opportunistic reuse. We asked questions about: (i) code reuse practice; (ii) architectural impact during reuse; and (iii) associated benefits. The complete list of questions is presented in the Appendix. The mapping between the research questions and the specific questions asked in the survey is as follows: RQ1 is answered by questions Q1 and Q4, RQ2 is answered by Q2 and Q3, RQ3 is answered by Q6 and Q7. Finally, we asked other questions (Q5, Q8 and Q9) not specifically related to any of the research questions but important to gain additional findings.

Data collection. As a result we collected responses from 30 industry practitioners and researchers in Brazil, Denmark, Finland, Italy, Spain and Sweden. We collected the data until November 15, 2019 as we could not get additional responses. We used the following online survey[13] to collect the responses. The authors suggest to email different reuse experts or people involve in reuse practices in the past based on contact of our research network and names found randomly in the research papers we analyzed. We double-checked that all participants who answer the survey respond to at least the

[13] https://tinyurl.com/yaagvmat.

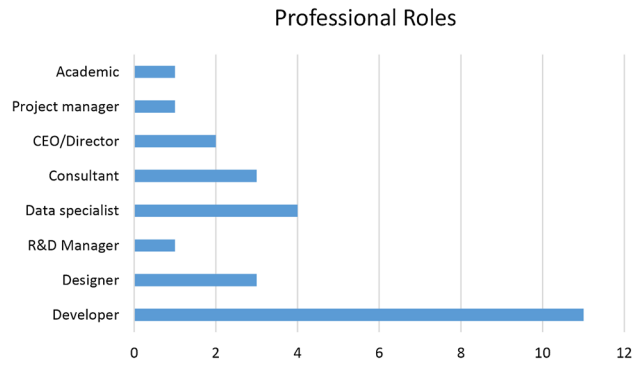

Fig. 1 Professional roles of the respondents

80% of the answers to accept the response as valid. In the following sections we discuss the results.

7.2 Survey results

The demographics of the survey are as follows. We received responses from subjects representing 13 companies and one university. The size of the companies are: seven large companies, two medium-size organizations and four small organizations. Only one respondent did not indicate his organization. The ages of the respondents range between 22 and 55 years, and the average industry experience is 15 years, ranging between 3 and 37 years. We provide diverse insights from 30 subjects located in different countries and organizations, with all of them involved in reuse practices, mainly in software companies. Figure 1 presents the distribution of the roles of the subjects, with developers, designers and data specialists being the most frequent roles.

Since we are interested in which programming languages offer more reuse opportunities and which are the most popular ones, Fig. 2 summarizes the languages that are most frequently used amongst the respondents. As we can see in the figure, Java, JavaScript, NodeJS, and other web-enabled languages such as Python, PHP and Ruby are the most popular ones for reuse purposes (29 subjects mentioned at least one of them). These languages are also those that developers found most amenable to software reuse.[14] Furthermore, we need to mention that five of our respondents answered that all or any platform are valid for reuse without indicating any specific programming language.

Regarding how the subjects identified the code they wanted to reuse, a significant percentage of the users (67%) listed Internet searches as the preferred method for finding reusable assets, while others emphasized the importance of web sites such as CodeProject,[15], Stack Overflow footnotehttps://stackoverflow.com/. GitHub[16] and

[14] Note that respondees were allowed to choose multiple languages, and we counted each mentioned language as a valid response.

[15] https://www.codeproject.com/.

[16] https://github.com/.

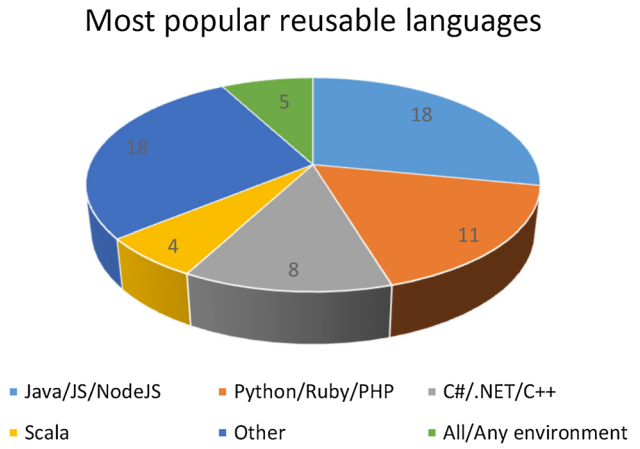

Fig. 2 Most popular programming languages for reuse

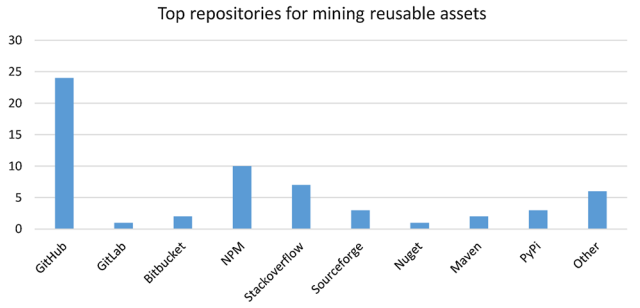

Fig. 3 Most popular repositories for mining reusable assets

Table 2 Average number of reused components per project

Ranges of reusable components per projects	Number of times
Between [0–3]	8
Between [4–20]	7
Between [21–100]	1
More than 100	5

Reddit.[17] Figure 3 lists the most popular code repositories based on the survey, with GitHub, NPM and Stack Overflow being the most frequently used ones. Only a small number of subjects responded that asking colleagues or using professional networks is a suitable way for finding reusable assets. Interestingly, reuse from previous projects developed in the companies was only highlighted by one of the subjects. One further interesting measurement is the extent of reuse per project. Table 2 shows the number of reused components per project as summarized by the subjects of our study.

[17] https://www.reddit.com/.

Table 3 Frequency of architecture updates after reuse

Frequency types	Frequencies
Always	2
During project reviews/when needed	9
Usually not/very seldom	1
Occasionally/sometimes	5

Our questions also addressed interoperability of the reusable assets within the existing system. Based on the answers, only 3% of the subjects did not consider interoperability as an important concern, while 67% of the subjects felt that interoperability is sometimes important or important for any reuse practice. However, 17% of the respondents did not fully understand our question and the expressed doubts regarding answering about the role of interoperability for reuse, since it may depend on the customer and their organization.

7.3 Impact on the architecture

When it comes to the importance and impact of code reuse in architectural design, it is difficult to summarize the received answers. The majority of the subjects considered that reuse affects the architecture, but some of them indicated that architectural design comes first and code reuse is merely a coding practice. Moreover, some of the respondents noted that larger components typically have a larger impact on architectural changes than smaller components. In particular, selected development frameworks often mandate (or effectively dictate) many of the architectural choices. In other cases, reuse may fall within the scope of design decisions made when designing the architecture, since the architecture defines the way code is distributed and reused over each service. However, when coordinating over a number of a applications and modules, code reuse becomes more cumbersome. One subject indicated that in some cases, reuse may drive architectural design. For instance, open source service running in Kubernetes as an inexpensive option against a costly commercial service.

The frequency of architecture updates after reuse is shown in Table 3. As we can see, fifteen subjects occasionally update the architecture after reuse, and six subjects indicated architecture changes to occur “always” or “during project reviews”. Only two respondents said that the architecture was rarely updated. From the six subjects that indicated the architecture must be updated after reuse, four out of six agreed (previous question) that code reuse drives architectural design; surprisingly, two out of the six respondents think that code reuse does not drive architectural design.

One subject indicated that the functionality derived from requirements should already have defined the architecture, but at the same time code reuse is used for achieving functionality. We understood from this response that all the functionality reused must be aligned with existing architecture, so the architecture would rarely be updated even when new reusable components are brought into the system.

Table 4 Summary of Benefits and Downsides of Reuse

Benefits	Downsides
Faster development	Compatibility problems may lead to technical debt, and copy-paste reuse practices to hard traceability problems
Increased productivity	Snowballing dependencies may affect code reuse
Cost reduction	Reusable software assets are often poorly maintained
Avoidance of common security risks	Security concerns may arise from open package repositories such as NPM
Stability of the initial prototypes and the reduction in the number of errors	GPL licenses are complex to understand

8 Implications for practitioners and researchers

Next, we provide an extended discussion of our key findings, which are summarized in Table 4. First, we present implications for practitioners for each of the research questions addressed, and then, a set of implications for researchers is provided.

Based on the presented sample project, the number of components that are readily available for reuse is overwhelming. There are packages and subsystems available for almost all imaginable needs, and in many cases there are de facto solutions for key areas, such as Hadoop for offline data analytics, to mention a concrete example. For several areas, there are a number of competing technologies. A good example are container technologies, an area for which there are solutions that offer slight variations between different implementations. At the level of implementation-level libraries, offerings become significantly richer; a good example are the JavaScript libraries for client-side Web development—an area that offers an abundance of relatively similar choices, with developer popularity shifting from one library to the next over the years. Given the cornucopia of choices, it is difficult to create a one-size-fits-all platform out of the available components, even if the target architectures were seemingly similar, as subtle differences in the reused components often imply that developers will have to create their own configurations and patches to make the selected libraries work together. In addition, in some domains like IoT, the situation is made more complex due to the use of complementing technologies that require different development approaches [43]. Hence, performing reuse in short term is easy and does not require that much skills. In contrast, long-lasting, strategic reuse is harder than ever and requires a lot of experience and skills, including not only software skills but also risk management experiment and business insight.

RQ1. *How does opportunistic reuse appear in practice?* The industrial example presented in Sect. 4 represents a rather typical end-to-end IoT system today, built upon open source components and readily available back-end service technologies.

The majority of microservices in this system were implemented in the JavaScript programming language, Dockerized into separate services, utilizing the Node.js runtime environment and its expansive NPM module ecosystem. Furthermore, the example demonstrates how various packages and libraries today offer ready-made solutions for many basic functionalities, such as user accounts and authentication, that are required by nearly all back-end services—In our case, as already mentioned, this includes systems such as *NGINX* for security perimeter, *Apache Kafka* for data acquisition, *Apache Storm* for real-time data analytics, and *Graphite* and *Icinga* for logging and monitoring. Furthermore, many backend-as-a-service systems have been proposed, that readily integrate such systems into a coherent hole. Reuse at such scale can significantly increase developer productivity, and for many large and complex software systems, reuse at such scale is often the only option when developing new systems from scratch. Thus, while opportunistic reuse can open security risks when components of unknown provenance are used, it also shows how fast and straightforward it can be to wire together a complex system with relatively small amount of development resources.

RQ2. *How do the developers select the reusable assets when using an opportunistic reuse strategy?* In motivating example, the selection of components, tools and techniques was left under the responsibility of the developers, who tended to choose to use latest and most fashionable tools and techniques in the implementation. At the level of individual components, the selection was based on various criteria, although the transitive and overlapping dependencies in the dynamically loaded modules, together with the fact that dependencies may change in different versions of the components, make it difficult to determine if a rational selection process was followed in developing the different subsystems. The survey confirmed the findings of the industrial example regarding the number of components. In this context, in which the amount of potentially reusable software is excessive to be understood by any individual, it is not surprising that almost all the developers use the Internet as a data source when writing code, often trusting anonymous commenters and popularity ratings more than their own colleagues. In fact, seeking advice online was a far more popular way for finding components than asking a colleague.

RQ3. *What is the expected architectural impact arising from the practice of opportunistic reuse?* As for architectural consequences, the variety of the responses in the survey was wide. Even though our gut feeling was that in most cases architecture drives reuse practices, there are exceptions in which reuse has a major impact on the architecture design, depending on how systematic the selected reuse approach is and how well it is enforced. Moreover, the fact that the majority of the respondents replied that the architecture may need updating after choosing new reusable assets basically implies that reuse indeed *does* impact the architecture or at least requires revisiting of the original architecture choices. In several cases the subjects mentioned a limited impact in the design mainly because they are focused on or driven by the "de facto" expected architecture of the system; thus they do not necessarily perceive that reused assets might affect the original design choices. This is in line with the observations in the industrial example, in which the general feeling was that its architecture was largely prescribed by the IoT domain and its generic end-to-end architecture. One particular concern worth mentioning is that some respondents expressed their cau-

tion about unknown dependencies that reused assets might bring into the architecture. Finally, software reuse is not only about the use of existing software to construct new software, but reusable assets can also be software knowledge [15]. Focusing on this knowledge introduces a wide range of research opportunities, ranging from analytical work to constructive development and risk management.

To complement the implications for practitioners discussed above, we now focus on the implications for researchers. This in particular concerns the downsides raised in Table 4.

Analytical work. One of the most obvious research topics for analytical research consists of systematic analyses of widely used open source systems, their components, and the quality and stability of their interfaces and documentation. Furthermore, as our findings imply that in many domains a generic reference architecture has emerged, systematic analyses can be extended to the most popular open source components for key domains, and recommendations of best available components for each area. An important part of these are objective reviews and measured statistics of them in real-world applications. The analysis can be enriched with the study and definition of recommended reuse patterns and combinations of most popular open source components. Moreover, open source projects could even provide information regarding the observed forms of their reuse as a part of larger systems, and hence contribute to the increased understanding of the scale of reuse—it is one thing to reuse a large-scale subsystem such as NGINX, and completely different to reuse an NPM package or an individual component. This will help mitigate challenges associated with compatibility problems and technical debt related to it, whereas technical debt tools and quality metrics can help to identify quality issues.

Tools and techniques. Numerous tools and techniques are needed for supporting opportunistic designs and associated reuse. To begin with, tools for visualizing all the "underwater" dependencies in a "tip of an iceberg" software system that relies extensively on SOUP (Software of Unknown Provenance) components are an essential element to better understand the fundamentals of different designs. This includes improved tools for static and dynamic component dependency analysis, "tree shaking" (for eliminating duplicate components), crawling to the end of dependency chains to create transitive closure of all the needed modules, and so on. Such facilities will also improve understanding regarding snowballing dependencies.

A second class of tools and techniques consists of systems that help assess the stability and maturity of reused components, e.g., how likely they will change in terms of their interfaces, and assessing how trustworthy their contributors are. The tools and techniques (e.g., dynamic, regularly updated dependency charts) should also help monitor and understand changes in widely used subsystems that are loaded on the fly from third-party sources. In addition, such tools enable assessing the quality of maintenance of reusable assets.

A third class of tools and techniques include systems that enable the development and testing of "iceberg" software systems within safe boundaries. Such sand-boxing technologies are especially important in complex systems in which software runs on multiple servers or virtual machines. For instance, with Docker Compose[18] it is

[18] https://docs.docker.com/compose/.

possible to package an entire cloud onto a single machine for testing purposes ahead of deploying the system onto an actual farm of servers or VMs. To some extent, this will also mitigate risks associated with open package repositories.

Risk management. The final engineering element that is intimately associated with opportunistic designs consists of risk management guidance and techniques that help assess the risks associated with "tip of the iceberg" systems depending on rapidly evolving third-party components [52]. Indeed, successful opportunistic reuse is heavily dependent on risk management. The use of third-party components—especially if it occurs in a fashion in which first-level reused components end up transitively pulling in layers and layers of other components—raises the risks associated with a software system considerably. While concerns regarding the issues around trust have been raised already over thirty years ago [45], the fact that unrelated software components are nowadays so commonly fused together means that software written with even the best of intentions can introduce severe problems. Today, risk management should cover issues other than code as such, in particular licence management. There are subtle differences between different licences that may yield two technically compatible systems legally incompatible [17]. Moreover, as we experienced in our research groups, training about the diversity of GPL-related and other open source licenses can help to reduce the entry barrier to understand its use.

9 Threats to validity

From the initial insight gained from our industrial example, the contextual interviews and the online survey we can confirm that opportunistic reuse is a proven trend nowadays. The diversity of software components in different repositories and variety of programming languages and technologies involved confirm the *de facto* opportunistic reuse practices in software development and especially in distributed teams. Nevertheless, the impact on opportunistic reuse in the architecture is proven only in some cases, so we need to conduct additional studies to demonstrate a stronger connection between ad hoc reuse and its impact in the design in terms of adaptation effort.

Regarding external validity and the general applicability of the results, we are aware that the sample of reuse practitioners is not very large and that the number and diversity of the countries of the interviewed subjects is small. Since the target group for our research consisted of reuse experts, they are more likely to favor software reuse than developers in general. However, it seems a good starting to confirm with different developers and designers from several countries that opportunistic reuse is an increasing trend due to the popularity of open source repositories. To confirm our initial results we need a broader study including practitioners from more countries.

Construct validity shows whether the experimental settings reflect the theory. Since our survey interviewed subjects from different roles, and some of the authors of this work have been involved in systematic reuse practices, we reduced the bias of our initial findings. In addition, we selected the subjects randomly to avoid additional bias in our analysis. Also, we followed common guidelines described by [20] for conducting semi-structured interviews and reduce the bias from the interviewees. Nevertheless, a second experiment involving more subjects would be useful to confirm our claims.

10 Related work

Systematic software reuse has been studied decades. There are numerous papers on that topic, including the classic survey paper by Krueger [27] that we have referred to in several places in this paper earlier. In contrast, there are only a few studies that investigate opportunistic software reuse, or the transition from systematic to opportunistic reuse. Several years ago, Frakes [14] described the shift to systematic reuse in software engineering and elaborated the success factors that help reuse to be systematic. Similar concerns were also highlighted by Tracz [47]. Another study [12] highlights the emergence of systematic reuse in the 2000s to improve software development productivity and quality. The authors analyzed 15 software projects, and they found evidence of systematization because of lack of incentives. This claim for systematic reuse practices in object-oriented programming is stated in [42] where reuse in OO was mostly ad hoc in the 1990s. The authors introduce the idea of reuse contracts to make assets more reusable.

Hartmann et al. brought the topic of opportunistic reuse to public attention [18]. More recently, Fischer et al. [13] describe clone-and-own techniques (i.e. a practice "where a variant of a system is built by coping and adapting existing variants") as a suitable form of systematic reuse that has been adopted in the recent years. The authors suggest a methodology applied to clone-and-own in variability models, because clone-and-own techniques often suffer of a lack of systematic process. This type of reuse is hence close to opportunistic reuse in this sense. A recent survey by Capilla et al. [5] investigates different forms of reuse from past practices to current ones. Although the authors provide some insight on how practitioners do reuse, they do not analyze in depth which of these practices are carried out in an opportunistic way. A recent work by Ali et al. [1] presents a hybrid DevOps process that follows a systematic reuse-based development and management process aimed to reduce reuse effort and cost based on information retrieval techniques. In [21], the authors apply data mining techniques to guide developers to reuse code snippets using a set of predefined rules and they developed a prototype to predict useful API code snippets.

Early works on opportunistic reuse such as in [26] explore scrapheap development as a form where developers identify opportunities for reuse. Scrapheap reuse is low-cost reuse strategy in which large parts of applications are composed from "scraps" of software functionality retrieved from systems that have been discarded. In [22] the authors discuss a pragmatic reuse approach of third-party functionality by two start-up companies in building successfully two products, as a clear demonstration of opportunistic reuse, but the effort required to integrate new services and components into the architecture may lead to severe changes in the design. Also the approach discussed in [24] highlights the role of opportunistic reuse in a systematic way around 80 software cases, while in [28], the authors propose to systematize the selection of external modules based on the extraction of design decisions from source code to avoid design mismatches when reusing opportunistically.

In a recent work by Kulkarni [29], the author discusses the need for sourcing software without a concrete reuse plan to improve developers' productivity. The authors highlight the existing problems and consequences of opportunistic reuse practices, such as fragile structure, violations of constraints or unforeseen behavior, mainly

caused by a widespread adoption of these reuse practices. Additionally, the study provided in Paschali et al. [37] highlights reuse opportunities in 600 open source projects in different application domains, as in some of the domains it is not easy to identify the reusable assets, such as one of the relevant questions we asked to our subjects in this research.

Moreover, the approach described in [46] highlights a trendy modern form of code reuse where software is engineered in the form of microservices, as an evolution of reusable web services. Microservices enable a scalable way for building service-based systems with components of small granularity and enabling rapid development, testing and deployment. In a similar vein, the work discussed in [6] report a survey about mining reusable microservices from legacy systems. All in all, it is clear that one of the trends since arrival of web services architectures is an opportunistic way to reuse web code snippets on behalf of the popularity of web-based systems and open source software. The authors in [30] describe a process to improve open-source software reuse and capitalize the opportunistic reuse practices in open source projects. Therefore and to bring a bit more controversy between both strategies, the authors suggest that opportunistic practices can be guided by a process to maximize the adoption of reuse, similarly to the steps performed by the practitioners we interviewed.

Finally, in the study reported in [7] about reusing web code snippets opportunistically, the authors report that developers rarely test immediately the system after opportunistic reuse happens, which is in line with some of the patterns and observations from the developers we interviewed.

11 Conclusions

In this paper, we have provided empirical evidence on the role of opportunistic software reuse by presenting an industrial case study, three practitioner interviews, and the results of a more comprehensive survey focusing on software reuse. Based on the results, software reuse takes place at a very large scale in the software industry today. In fact, it appears that it is nearly impossible to write any significant software systems nowadays without reusing third-party components extensively, with the developers themselves only writing the "tip of the iceberg", while the bulk of the system comes from external sources and unfamiliar developers. This is dramatically different from software development in the 1980s and 1990s when developers still prided on writing most of the software themselves. Today, software systems tend to contain so much invisible code with so many dependencies that they are impossible to analyze by hand.

While McIlroy's original vision of software reuse [33] called for high-quality mass-produced software components to be used in a large, industrial scale, today's software reuse scene is really more about grassroots, opportunistic reuse rather than strategic, systematic reuse. This transition has been enabled by the ease of searching and publishing components on the Internet. The resulting approach raises many interesting challenges from architectural mismatches ("tail wagging the dog"), poor understanding of the actual behavior ("I decided to use this component since it had been used by so many other developers before") to all kinds of security issues arising from unknown, dynamic dependencies with other components.

Moreover, even individual code snippets seem to be easy to integrate into a bigger whole. Still, in cases where components do not fit well, changes at the architecture level have to be made, which is recognized by the developers. However, many developers do not perform the necessary security checks during the integration phase, which may create security issues in the code in particular in the long run. Thus, while reuse may not be strategic in terms of company goals, the elements of systematic approach to selecting components for reuse are emerging, consisting of recommendations, experience reports and personal experience. This in turn emphasizes the professional attitude of practicing developers while reusing components of different origins.

To conclude, we have studied opportunistic design and its implications, and presented a call for action for the research community. It is our belief that opportunistic reuse and the "tip of the iceberg" programming model has not yet received the attention it deserved, as the eventual solution will be developer education to understand the contexts in which opportunistic design is acceptable, and where more risk-aware approaches are needed. Furthermore, in general we hope that this paper raises the awareness of opportunistic reuse, and encourages people to tackle the challenges associated with this important topic. To this end, practices and software reuse principles developed in the 1980s and 1990s—especially in the area of creating modular, well-documented, stable interfaces and reusable components—provide a solid foundation to build on.

Acknowledgements Open access funding provided by University of Helsinki including Helsinki University Central Hospital. The work of N. Mäkitalo was supported by the Academy of Finland (Project 328729). The work from R. Capilla is partially funded by the Spanish research network MCIU-AEI TIN2017-90664-REDT.

Appendix: Survey on opportunistic reuse practice

In this appendix, we describe the nine questions that were presented to the subjects about opportunistic reuse practices.

1. In which environment (Java, Python, C#, PHP., etc) have you encountered opportunistic code re-use in today's software development?
2. How do you identify the code (e.g., packages, libraries, frameworks, etc.) that you want to re-use? (Mining code, Internet search, Database query).
3. What are the preferred open source repositories that you search for reusable code? (GitHub, GitLab, Bitbucket, Sourceforge, Smipple, Snipplr, Code Project, Javascript, npm, Other (please specify)).

4. To what extent are interoperability concerns a problem when integrating reusable code into the systems of your organization? (Not important, Sometimes relevant, Important concern).
5. How many reusable components do you reuse on average per project?
6. To what extent does code re-use drive your architectural design or development process?
7. Do you update your software architecture after reusing code components, snippets, libraries? (Always, Never, Occasionally, During project reviews, Other (please specific)).
8. What positive aspects do you see in today's code re-use? (reduced number of errors, faster development, productivity, Other (please specify)).
9. What are the downsides of today's (opportunistic) code re-use? (software is harder to debug, emergent features, Other (please specify)).

References

1. Ali N, Horn D, Hong JE (2020) A hybrid DevOps process supporting software reuse: a pilot project. J Softw Evol Process. https://doi.org/10.1002/smr.2248
2. Biggerstaff TJ, Richter C (1987) Reusability framework, assessment and directions. IEEE Softw 4(2):41–49
3. Bouzid A, Rennyson D (2015) The art of SaaS: a primer on the fundamentals of building and running a successful SaaS business. Xlibris
4. Capilla R, Gallina B, Cetina Englada C (2019) The new era of software reuse. J Softw Evol Process 31(8):e2221
5. Capilla R, Gallina B, Cetina Englada C, Favaro J (2019) Opportunities for software reuse in an uncertain world: from past to emerging trends. J Softw Evol Process 31(8):e2217
6. Carvalho L, Garcia A, Assunção WKG, Bonifácio R, Tizzei LP, Colanzi TE (2019) Extraction of configurable and reusable microservices from legacy systems: an exploratory study. In: Proceedings of the 23rd international systems and software product line conference, SPLC 2019, volume A, Paris, France, September 9–13, 2019, ACM, pp 6:1–6:6
7. Ciborowska A, Kraft NA, Damevski K (2018) Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the web. In: Proceedings of the 15th international conference on mining software repositories, MSR 2018, Gothenburg, Sweden, May 28–29, 2018, ACM, pp 94–97
8. Clements P, Northrop L (2002) Software product lines. Addison-Wesley, Boston
9. Dikel D, Kane D, Ornburn S, Loftus W, Wilson J (1997) Applying software product-line architecture. Computer 30(8):49–55
10. Easterbrook S, Singer J, Storey MD, Damian DE (2008) Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering, pp 285–311
11. Ericsson KA, Simon HA (1998) How to study thinking in everyday life: contrasting think-asloud protocols with descriptions and explanations of thinking. Mind Cult Act 5:176–186
12. Fichman RG, Kemerer CF (2001) Incentive compatibility and systematic software reuse. J Syst Softw 57(1):45–60
13. Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Enhancing clone-and-own with systematic reuse for developing software variants. In: 30th IEEE international conference on software maintenance and evolution, Victoria, BC, Canada, September 29–October 3, 2014, pp 391–400
14. Frakes WB, Isoda S (1994) Success factors of systematic reuse. IEEE Softw 11(5):14–19
15. Frakes WB, Kang K (2005) Software reuse research: status and future. IEEE Trans Softw Eng 31(7):529–536
16. Gao JZ, Tsao HS, Wu Y (2003) Testing and quality assurance for component-based software. Artech House Publishers

17. Hammouda I, Mikkonen T, Oksanen V, Jaaksi A (2010) Open source legality patterns: architectural design decisions motivated by legal concerns. In: Proceedings of the 14th international academic mindtrek conference: envisioning future media environments, ACM, pp 207–214
18. Hartmann B, Doorley S, Klemmer SR (2008) Hacking, mashing, gluing: understanding opportunistic design. IEEE Pervasive Comput 7(3):46–54
19. Holtzblatt K, Joses S (1995) Conducting and analyzing a contextual interview. Human–computer interaction, pp 241–253
20. Hove S, Anda B (2005) Experiences from conducting semi-structured interviews in empirical software engineering research. In: 11th IEEE international symposium in software metrics, IEEE, pp 10–23
21. Hsu S, Lin S (2011) MACs: mining API code snippets for code reuse. Expert Syst Appl 38(6):7291–7301
22. Jansen S, Brinkkemper S, Hunink I, Demir C (2008) Pragmatic and opportunistic reuse in innovative start-up companies. IEEE Softw 25(6):42–49
23. Jones TC (1984) Reusability in programming: a survey of the state of the art. IEEE Trans Softw Eng 10(5):488–494
24. Kalnina E, Kalnins A, Celms E, Sostaks A, Iraids J (2010) Generation mechanisms in graphical template language. In: MDA & MTDD 2010—proceedings of the 2nd international workshop on model-driven architecture and modeling theory-driven development, in conjunction with ENASE 2010, Athens, Greece, July 2010, SciTePress, pp 43–52
25. Kim Y, Stohr EA (1998) Software reuse: survey and research directions. J Manag Inf Syst 14(4):113–147
26. Kotonya G, Lock S, Mariani J (2008) opportunistic reuse: lessons from scrapheap software development. In: Proceedings of component-based software engineering, 11th international symposium, CBSE 2008, Karlsruhe, Germany, October 14–17, 2008, pp 302–309
27. Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183
28. Kulkarni N (2013) Systematically selecting a software module during opportunistic reuse. In: Notkin D, Cheng BHC, Pohl K (eds) 35th international conference on software engineering, ICSE'13, San Francisco, CA, USA, May 18–26, 2013, IEEE Computer Society, pp 1405–1406
29. Kulkarni N, Varma V (2017) Perils of opportunistically reusing software module. Softw Pract Exp 47(7):971–984
30. Lampropoulos A, Ampatzoglou A, Bibi S, Chatzigeorgiou A, Stamelos I (2018) REACT—a process for improving open-source software reuse. In: 11th international conference on the quality of information and communications technology, QUATIC 2018, Coimbra, Portugal, September 4–7, 2018, IEEE Computer Society, pp 251–254
31. Lanergan RG, Grasso CA (1984) Software engineering with reusable designs and code. IEEE Trans Softw Eng 10(5):498–501
32. Lentz M, Schmid HA, Wolf PF (1987) Software reuse through building blocks. IEEE Softw 4(4):34–42
33. McIlroy MD (1968) Mass produced software components. In: Naur and Randell (eds) Software engineering: report of conference sponsored by the NATO science committee, Garmisch, Germany, Oct 7–11, 1968, pp 79–85
34. Mikkonen T, Taivalsaari A (2019) Software reuse in the era of opportunistic design. IEEE Softw 36(3):105–111
35. Naur P, Randell B (1969) Software engineering: report of a conference sponsored by the NATO Science Committee (Garmisch, Germany, Oct 7–11, 1968). NATO Scientific Affairs Division, Brussels
36. Newman S (2015) Building microservices: designing fine-grained systems. O'Reilly Media
37. Paschali ME, Ampatzoglou A, Bibi S, Chatzigeorgiou A, Stamelos I (2017) Reusability of open source software across domains: a case study. J Syst Softw 134:211–227
38. Pfleeger SL, Kitchenham BA (2001) Principles of survey research: part 1: turning lemons into lemonade. ACM SIGSOFT Softw Eng Notes 26(6):16–18
39. Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164
40. Schmidt DC (1999) Why software reuse has failed and how to make it work for you. C++ Report 11(1):1999
41. Shaw M (1995) Architectural issues in software reuse: it's not just the functionality, it's the packaging. ACM ACM SIGSOFT Softw Eng Notes 20:3–6
42. Steyaert P, Lucas C, Mens K (1997) Reuse contracts: making systematic reuse a standard practice. https://www.semanticscholar.org

43. Taivalsaari A, Mikkonen T (2018) On the development of IoT systems. In: 2018 third international conference on fog and mobile edge computing (FMEC). IEEE, pp 13–19
44. Taivalsaari A, Mäkitalo N, Mikkonen T (2019) Programming the tip of the iceberg: software reuse in the 21st century. In: 2019 EUROMICRO conference on software engineering and advanced applications. IEEE
45. Thompson K (1984) Reflections on trusting trust. Commun ACM 27(8):761–763
46. Tizzei LP, dos Santos MN, Segura VCVB, Cerqueira RFG (2017) Using microservices and software product line engineering to support reuse of evolving multi-tenant SaaS. In: Cohen MB, Acher M, Fuentes L, Schall D, Bosch J, Capilla R, Bagheri E, Xiong Y, Troya J, Cortés AR, Benavides D (eds) Proceedings of the 21st international systems and software product line conference, SPLC 2017, volume A, Sevilla, Spain, September 25–29, 2017. ACM, pp 205–214
47. Tracz W (1995) Confessions of a used-program salesman: lessons learned. In: Samadzadeh MH, Kazerooni-Zand M (eds) Proceedings of the ACM SIGSOFT symposium on software reusability, SSR@ICSE 1995(April), pp 23–30 (1995) Seattle. WA, USA, ACM, pp 11–13
48. Trendowicz A, Punter T et al (2003) Quality modeling for software product lines. In: Proceedings of the 7th ECOOP workshop on quantitative approaches in object-oriented software engineering
49. Turner M, Budgen D, Brereton P (2003) Turning software into a service. Computer 36(10):38–44
50. Yin RK (2014) Case study research design and methods, 5th edn. Sage, Thousand Oaks
51. Zax D (2012) Many cars have a hundred million lines of code. MIT Technology Review
52. Zimmermann M, Staicu CA, Tenny C, Pradel M (2019) Small world with high risks: a study of security threats in the NPM Ecosystem. 28th USENIX security symposium. Santa Clara, CA, pp 995–1010